UNIVERSITY OF DHAKA

Effects of Version Control System on the Efficiency of Refused Bequest Code Smell Identification Algorithm

by

BSSE0425 - Rashed Rubby Riyadh

A thesis submitted in partial fulfillment for the
degree of Bachelor of Science in Software Engineering

in the

Institute of Information Technology

May 2016

Effects of Version Control System on the Efficiency of Refused Bequest Code Smell Identification Algorithm

Rashed Rubby Riyadh

Approved:

Signature Date

_____ _____

Supervisor: Dr. Kazi Muheymin-Us-Sakib

Associate Professor & Director

Signature Date

_____ _____

Dr. Md. Shariful Islam

Associate Professor

Signature Date

_____ _____

Mr. Alim Ul Gias

Lecturer

Signature Date

_____ _____

Mr. Amit Seal Ami

Lecturer

Abstract

Refused bequest is a code smell that originates from introduction of inappropriate class hierarchies where subclasses do not specialize or use the functionalities provided by the parent class. The existing refused bequest code smell identification algorithms use either static analysis or a combination of static and dynamic analysis based technique. However, those algorithms do not consider version control history. If the parts that have been changed from a previous version can be considered for refused bequest identification, the identification time may be reduced.

An approach is proposed here employing the version control history to reduce identification time in refused bequest code smell identification. The proposed approach has four components namely Change Collector, Source Code Parser, Error Injector and Smell Calculator. Change Collector identifies source codes that have been changed from the previous version. Source Code Parser then parses the source codes to identify metrics such as inherited method, overridden method, super classs method invocation etc. After parsing the source codes, Error Injector introduces intentional errors in the non-overridden inherited methods of the subclasses. Smell Calculator then calculates smell for each of the subclasses based on metrics identified in Source Code Parser and Error Injector.

The comparative analysis of the results shows a significant improvement in execution time. The proposed approach is found to be 16.2% more time efficient than the existing approach for the sample projects. This is because the approach incorporates version control history in the identification process. Moreover, the proposed approach is able to identify code smells as accurately as the existing approach.

Acknowledgements

I would like to express my heartiest gratitude to Dr. Kazi Muheymin-Us-Sakib for his support and timely guidance during the thesis compilation. He has been meticulous and fastidious to bring the best out of me throughout the thesis.

Contents

List of Figures

List of Tables

To my parents,

Who have always been the source of great motivation and inspiration.

Chapter 1

Introduction

Refused bequest is one of the dominant code smells that originates from improper introduction of inheritance hierarchy in a software system. That is, subclasses do not take the advantage of reusing parent methods because generalization does not exist at all. Generally, software engineers introduce inheritance hierarchy among classes to generalize behaviors or methods so that duplication of source code can be eliminated. However, improper use of inheritance violates that philosophy and classes contain source codes that these classes do not utilize at all. Moreover, code base of any software system evolves with time. It is possible that some new hierarchies may develop whereas some hierarchies may not exist anymore. So, identification of refused bequest code smell from only the evolved parts will increase the efficiency of whole identification process.

1.1 Overview

The identification time of refused bequest algorithms depends on the amount of information it has to process. If refused bequest identification algorithm is applied to the whole source codes rather to the evolved parts, the efficiency of the process may be decreased. Because, refused bequest algorithms generally work

by parsing the software source codes to calculate various software metrics such as number of inherited methods, number of overridden methods, number of subclasses in the hierarchy etc., and by running the source codes to identify client usage of the hierarchy. If these steps are applied to more source codes, refused bequest identification algorithm will take more time to identify the hierarchies. Moreover, refused bequest identification tools should provide continuous feedback in a reasonable time. Because, users need to change the hierarchies based on the provided feedback, and run the tool again to see that the code smells do not remain any more. If those tools provide slow feedback, users will lose their interest in using those tools.

1.2 Issues in State-of-the-Art Refused Bequest Code Smell Identification

Generally, two approaches are used in refused bequest code smell identification - Static code analysis based approach and Combination of Static and Dynamic analysis based approach. The necessity to analyze a hierarchys clients to identify original intention of a generalization has been emphasized in [2]. A suite of metrics which quantify the uniformity of clients calls with respect to the services provided by a hierarchy is proposed in this paper. These metrics are then used to characterize class hierarchy and to detect design anomalies. A detection strategy combining appropriate code metrics and definition of threshold is proposed by R. Marinescu [3]. In this paper, code metrics are identified by static analysis of the system source code and these code metrics are compared with thershold values to identify design flaws or code smells. Another approach employing logic metaprogramming to identify inappropriate interfaces is proposed by T. Tourwe et al. [4]. In this paper, all direct subclasses of a super class are identified and then all

possible subset of these identified classes are used to identify inappropriate interfaces. A combination of static and dynamic analysis to identify refused bequest is proposed by Elvis Ligu et al. [1]. In this paper, introduction of intentional error in subclasses non-overridden inherited methods is used to identify clients usage of super class methods. Although all the above papers provide different approach to identify refused bequest code smell, benefits provided by version control systems are not employed to limit search space for the code smell identification.

1.3 Research Questions

Now-a-days, version history keeping is an inherent characteristic of any software project. It provides the ability to identify recently evolved parts. If only evolved parts are considered for identifying the refused bequest code smell, search space will be reduced. In this research, an approach to increase the efficiency of the algorithm proposed by Elvis Ligu et. al. [1] will be proposed. Because, it is one of the prominent algorithms that can accurately identify refused bequest code smells from source codes. Also, a new tool will be created by employing the proposed approach. Thus lead to the primary research question:

> How to increase the efficiency of the refused bequest identification algorithm using version control history?

More specifically, the research question can be divided into the followings:

1. How to identify changed parts of the source codes for refused bequest identification?
2. How to combine the changed results with the existing results?

1.4 Rationale for the Research

In the context of object-oriented systems, the notion of inheritance has been recognized as a key feature claimed to reduce the cost and effort in software maintenance. However, inheritance is not a panacea, especially if it is applied incorrectly in cases where other forms of relationships would be more appropriate. So, assist developers in verifying whether the inheritance relationships they establish are appropriate, will make the inheritance relationships more meaningful. Also, meaningful relationships will help the developers to understand the code easily and to modify code for adding new features. Thus, it will reduce cost in maintaining and extending the code base. Moreover, refused bequest identification tools should provide feedback about the affected inheritance relationships as soon as possible. Because, software developers may write more codes employing the affected relationships by the time the feedback arrives. They then have to refactor those codes in order to make those relationships appropriate. Thus, increasing efficiency of the existing algorithms will help the developers to refactor less codes, that is, reduce their time and effort in developing the software. Above all, Bangladesh is digitized at a great pace to fulfill the Vision 2021. As a result, more and more services are being developed to support the various sectors of the Government. If these services are not inherently maintainable, the Government has to spend a lot for the maintenance. Also, more requirements will come up in future to add new functionalities with the existing services. If the code bases are not understandable and extensible, the Government has to develop the services from the scratch. Thus, more effort, cost and time will be needed to achieve the Vision 2021.

1.5 Organization of the Thesis

This section gives an overview of the remaining chapters of this thesis. The chapters are organised as follows

Chapter 2 : The concept of code smells along with a detailed overview of refused bequest code smell is provided.

Chapter 3 : To the best of author's knowledge, no existing literature incorporates Version Control System history with refused bequest code smell algorithms. This chapter shows the existing researches in refused bequest code smell identification.

Chapter 4 : The methodology of proposed approach is discussed in detail in this chapter.

Chapter 5 : The implementation of the framework and comparative result analysis is presented here.

Chapter 6 : It is the concluding chapter which contains a discussion of the proposed approach, threats to validate the proposed approach and some future directions.

Chapter 2

Background Study

Legacy software systems contain design anomalies and architectural problems that make these systems hard to maintain, change or extend. These abnormalities are first termed as "Code Smells" by Kent Beck [5]. This chapter provides a brief description of code smells along with its different types. As, this thesis is going to incorporate version control history in refused bequest code smell algorithm, this smell is also introduced at the end of the chapter.

2.1 Concept of Code Smells

Martin Fowler describes code smell as a surface indication that usually corresponds to a deeper problem in the system [5]. Sahin et al. represent code smell as design situations that can affect the maintenance and evolution of software [6]. Elvis Ligu et al. describe code smell as design principle violations or architectural anomalies present in the code [1].

Code smells comes from finding the scope of refactoring in a software system. Refactoring is a technique that changes the internal structure of system without modifying the external behavior. However, software developer at first needs to

identify or indicate where to perform the refactoring. A Code smell is a sign that there might be a problem in source code that could be removed by performing one or more refactorings [7].

Code smells are not the same as syntax errors or other compiler warnings - compilers can find those - but rather indications of bad program design or bad programming practices that could pose a problem when the program needs to be changed, e.g. ported to a new platform or adding of new functionality [7].

2.2 Types of Code Smells

Code smells can be broadly divided into two types which are given below.

1. Source of the smells
2. Detection Complexity of the smells

The above mentioned types of code smells are described in the following section.

2.2.1 Source of the Smells

The smells can be further categorized into three types according to where these smells occur.

Production Code Smells

Production code smells are those that are found in the software production codes. These code smells tend to make the software systems complex, rigid and inflexible. Thus, these software systems become hard to understand, modify, extend and maintain.

Martin Fowler has enlisted 22 code in his book [5]. Those are - Duplicated Code, Long Method, Large Class, Long Parameter List, Divergent Change, Shotgun Surgery, Feature Envy, Data Clumps, Primitive Obsession, Switch Statements, Parallel Inheritance Hierarchies, Lazy Class, Speculative Generality, Temporary Field, Message Chains, Middle Man, Inappropriate Intimacy, Alternative Classes with Different Interfaces, Incomplete Library Class, Data Class, Refused Bequest, Comments.

Production code smells can be further categorized into below types:

- **Bloaters**

  Bloaters are code, methods and classes that have increased to such gargantuan proportions that they are hard to work with. Usually these smells do not crop up right away, rather they accumulate over time as the program evolves and especially when nobody makes an effort to eradicate them.

  Long Method, Large Class, Primitive Obsession, Long Parameter List, Data Clumps are the smells that fall under this category.

- **Object-Orientation Abusers**

  These are incomplete or incorrect application of object-oriented programming principles.

  Switch Statements, Temporary Field, Refused Bequest, Alternative Classes with Different Interfaces are the smells that constitute this category.

- **Change Preventers**

  These smells mean that if something need to be changed in one place in code, many changes have to be made in other places too. Program development becomes much more complicated and expensive as a result.

  The smells that fall under this category are- Shotgun Surgery, Divergent Change, Parallel Inheritance Hierarchy.

- **Dispensables**

  A dispensable is something pointless and unneeded whose absence would make the code cleaner, more efficient and easier to understand.

  Comments, Duplicate Code, Lazy Class, Data Class, Dead Code, Speculative Generality are this kind of code smells.

- **Couplers**

  All the smells in this group contribute to excessive coupling between classes or show what happens if coupling is replaced by excessive delegation.

  Feature Envy, Inappropriate Intimacy, Message Chains, Middle Man smells fall under this category.

Test Code Smells

van Deursen et al. describes a special category of code smells: smells that found in test code [8]. In Extreme Programming (XP) or Test Driven Development (TDD), a large part of total code base is test code. So, it is necessary to make this code easier to read and change, like production code. The test smells and the refactorings to clean them up are different from those found in "regular" code. Some of the code smells that are special to test codes are Eager Test, Lazy Test, Resource Optimism, Test Run War, Assertion Roulette etc.

Others

Embedded code smells are design anomalies in a software system that are embedded in server side code [9]. Among various other code smells, separation of concern, software modularity and compliance with coding standards are mainly present in embedded code. These smells usually make the applications hard to maintain, modify or reuse. As a consequence, it may increase maintenance effort and cost to an unbearable level. Also, these smells are hard to detect and fix, because codes are embedded in and generated from server side code.

2.2.2 Detection Complexity of the Smells

Code smells can also be divided into two types according to detection complexity.

Primitive Code Smells

Code smells that can be directly observed from code are known as primitive code smells. For example, long method, switch statement, comments etc. code smells can be easily identified by observing the code. Thus, these smells require limited efforts in identifying them manually.

Derived Code Smells

Code smells that can only be derived from facts extracted from codes are known as derived code smells. For example, refused bequest, feature envy, parallel inheritance etc. code smells cannot be directly observed and have to be identified by calculating different code metrics such as, cohesion, coupling, method call sequence, clients code usage pattern etc. Thus, identification of these smells require greater human efforts and the identification process is given more priority over the primitive code smells.

2.3 Refused Bequest Code Smells

Refused bequest code smell is a derived code smell where subclasses do not specialize the inherited methods by overriding it. This type of code smell is about subclasses that make none or little use of the methods and fields they get from their superclass by inheritance [7]. The Refused Bequest code smell concerns an inheritance hierarchy where a subclass does not support the interface inherited from its parent class [5]. More precisely, this smell is present if the functionality inherited by the subclass is not utilized by its clients nor specialized by means of overriding. In other words, the relation between the superclass and the subclass does not constitute an is-a relationship.

2.3.1 An Example of Refused Bequest Code Smells

Refused bequest code smell example is shown in Figure 2.1.

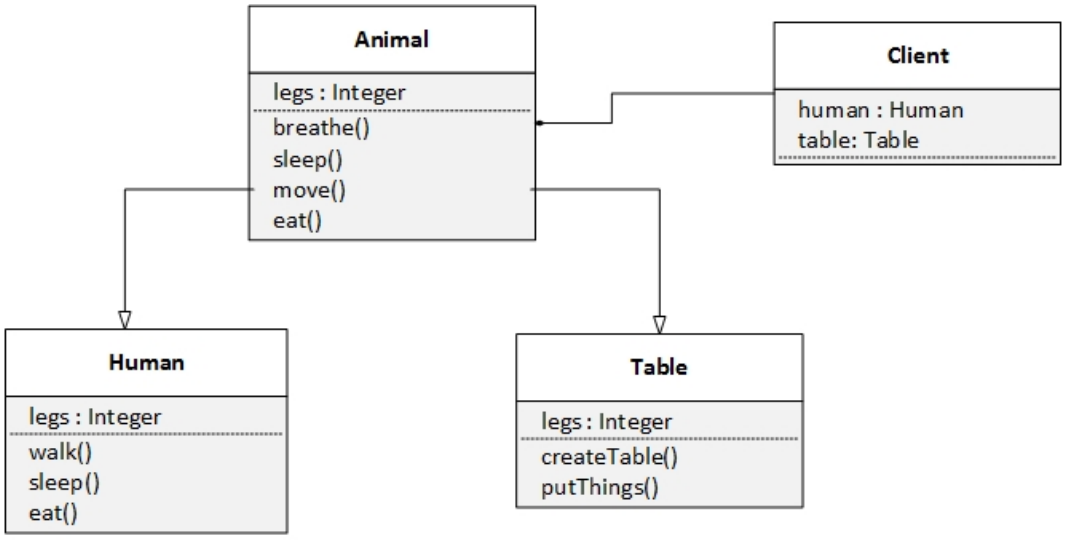

FIGURE 2.1: Example of Refused Bequest Code Smells

The example contains three classes namely Animal, Human and Table. These classes establish an inheritance hierarchy where Animal is super class of Human and Table. Here the justification of the hierarchy is that both Human and Table have attribute *legs*. Subclass Human specializes or overrides the default implementation of sleep() and eat() methods provided by super class. However, Table does not specialize any of the inherited methods. Now, consider Client which creates objects of the subclasses Human and Table. When Client uses the object of Table, it will generally invoke createTable() and keepThings() methods. However, inherited methods of Table will never be invoked because these methods do not conform to the hierarchy. That is, subclass Table refuses the methods provided by parent or super class Animal.

2.3.2 Treatment of Refused Bequest Code Smells

Matrin Fowler has proposed two treatment strategies for refused bequest code smells in his book [5]. Those are given below.

- If inheritance makes no sense and the subclass really does have nothing in common with the super class, eliminate inheritance in favor of Replace Inheritance with Delegation. That is, create a field and put a super class object in it, delegate methods to the super class object, and get rid of inheritance.
- If inheritance is appropriate, get rid of unneeded fields and methods in the subclass. Extract all fields and methods needed by the subclass from the parent class, put them in a new subclass, and set both classes to inherit from it.

2.4 Summary

A discussion of code smells is presented in the chapter. Although code smells have been introduced less than two decades ago, it has got importance in the software industry as well as in the academia. According to Cherubini and colleagues survey of 427 developers at Microsoft, developers consider refactoring as important as or more important than understanding code and producing documentation [10]. In the following chapter, some refused bequest code smell identification literature is presented.

Chapter 3

Literature Review of Refused Bequest Code Smell Identification

The frequency of refused bequest code smell is relatively low in a software system [5]. However, many researchers have investigated it individually or with other code smells. In the literature, several refused bequest identification techniques have been proposed. These techniques either use static code analysis or a combination of static and dynamic code analysis.

3.1 Static Code Analysis Based Detection Technique

Stefan Slinger proposed a detection strategy employing the static code analysis of the source code [7]. In this approach, a parser analyzes the Java source code files and produces abstract syntax trees, containing all relevant structural information. Making use of the source code (instead of byte code) even makes it possible to analyze code that is not ready for compilation yet, or that still has a few bugs in it. Whenever a syntax error is found during the parsing process, the abstract

syntax tree node(s) involved are marked as being malformed by setting a flag constant on the node(s). Depending on the seriousness of the error, some source code may not be represented in the abstract syntax tree. When the parser is done, an analyzer (visitor) traverses the abstract syntax trees, collects smell aspects and stores them in a repository. For refused bequest identification, smell aspects such as information on parent classes, methods, fields, and the methods and fields that are used by a class are needed. A Grok script is then executed on the smell aspect or fact repository to identify the classes that contain refused bequest code smells.

A detection strategy combining appropriate code metrics and definition of threshold is proposed by Radu Marinescu [3]. In this paper, a novel mechanism for formulating metrics-based rules is proposed that can capture deviations from good design principles and heuristics. Using detection strategies, an engineer can directly localize classes or methods affected by a particular design flaw, rather than having to infer the real design problem from a large set of abnormal metric values. Two mechanisms are identified that are important in use of metric for detection strategy, namely filtering and composition. Filtering is the process of reducing initial dataset. Composition can be considered as glue between different metrics. After defining the code metrics, those metrics are identified by static analysis of the system source code and these code metrics are compared with thershold values to identify design flaws or code smells.

The above detection techniques which rely only on static analyses cannot reveal dynamic information which is crucial for determining whether an inheritance relationship is appropriate or not. For example, polymorphic method invocations which totally justify the use of generalization can only be detected by dynamic analysis.

3.2 Combination of Static and Dynamic Analysis Based Detection Technique

The necessity to analyze a hierarchys clients to identify original intention of a generalization has been emphasized in [2]. A bi-dimensional characterization of class hierarchies namely the code reuse dimension and the interface reuse dimension is introduced in this paper. Also, a simple suite of metrics that contains Total Uniformity, Total Non-uniformity and Partial Uniformity is defined. Then, a prototypical tool called MEMBRAIN is used to perform data flow analysis on source codes. In order to approximate the aforementioned uniformity metrics, the intra-procedural static class analysis (SCA) have been implemented using MEMBRAIN. This data flow analysis determines at particular program points the set of classes for an object. In other words, it determines for any reference variable, at a particular program point, the possible set of classes of the instance to which that reference points. Based on this information computed for all the potential callers of a method M and based on the ResponsibleFor set of the same method, uniformity metrics for that method can be easily computed.

Amin Milani Fard et al. proposed an approach that uses a metric-based algorithm, and combines static with dynamic analysis to detect these smells in JavaScript code [11]. It employs a common heuristic based approach that uses code metrics and threshold. Due to the dynamic nature of JavaScript, static code analysis alone will not suffice. Therefore, in addition to static code analysis, dynamic analysis is also employed in the paper to monitor and infer information about objects and their relations at runtime. In this approach, the configuration containing the defined metrics and thresholds, is fed into the code smell detector. The JavaScript code of a given web application is then intercepted by setting up a proxy between the server and the browser. The codes are then traversed to generate an Abstract Syntax Tree (AST). During the AST traversal, the analyzer visits all program entities, objects, properties, functions, and code blocks, and stores their structure

and relations. At the same time, patterns from the AST such as names of objects and functions are extracted. Also, JavaScript objects, their types, and properties are inferred dynamically by querying the browser at runtime. Finally, based on all the static and dynamic data collected, code smell is detected using the metrics.

Elvis Ligu et al. have proposed a combination of static and dynamic code analysis based technique to identify refused bequest [1]. At first, all the source codes of the project are collected and parsed to generate the inheritance hierarchy. After that, the super class methods are overridden which are not overridden by the subclasses. Then, intentional errors are introduced in those overridden methods. The significance is that, if these methods are actually employed by subclasss users, there will be failures while performing test cases. Then, unit test cases of the project are executed and all the instances of failures are documented. All these information are used to identify the classes that contain refused bequest code smell and to categorize the smell intensity from least catastrophes to most catastrophes. The authors have developed an Eclipse IDE plugin on top of JDeodorant using the technique described. In this plugin, user can choose a whole project or a package to be tested. It then runs the identification algorithm and shows the user findings ordered by severity.

3.3 Summary

Although all the above papers provide different approaches to identify refused bequest code smell, benefits provided by version control systems are not employed to limit search space for the code smell identification. So, there is a need for further research to accommodate the change history into the detection of refused bequest code smells.

Chapter 4

Methodology

An approach to identify refused bequest code smells using version control history is proposed in this chapter. As discussed in the earlier chapters, identification time of refused bequest code smell identification algorithm depends on how much information or source codes the algorithm has to process. So, if the identification process can be run on only changed parts and affected hierarchies, the amount of information or source codes to identify refused bequest code smell can be reduced significantly. However, current algorithms run the identification process on the total source codes, rather than on the changed and change affected source codes.

The proposed approach will use a combination of static and dynamic analysis based technique along with version control history to identify refused bequest code smell. As discussed in earlier chapters, static analysis provides information of the class structure, underlying hierarchies and how these hierarchies are employed by the subclasses. That is, static analysis provides how these inheritance hierarchies are employed by the subclasses. On the other hand, dynamic analysis provides information on how these hierarchies are used by their clients. That is, dynamic analysis provides how these inheritance hierarchies are used by the clients. Moreover, version control history assists in identifying the changed parts of the source codes.

4.1 Overview of the Proposed Approach

The overview of the proposed approach is shown in Figure 4.1.

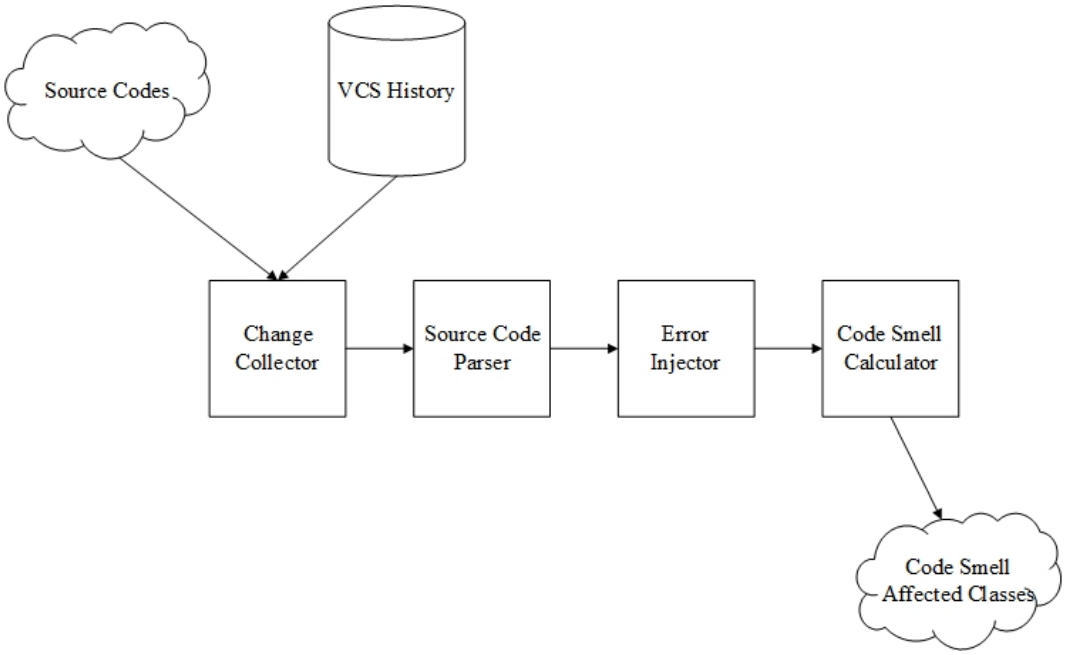

FIGURE 4.1: Components of Refused Bequest Code Smell Identification using Version Control System History

The proposed approach broadly contains four component as given below.

- Change Collector
- Source Code Parser
- Error Injector
- Code Smell Calculator

The proposed approach takes source codes and version control system history as input, and produces refused bequest code smell affected classes as output. Change Collector identifies changed parts of given project from the version control history and provides the changed classes to Source Code Parser. Source Code Parser then parses the identified source codes and identifies the inheritance hierarchy as well

as code metrics such as inherited method, overridden method, super classs method invocation etc. Then Error Injector implements the inherited classes that are not overridden by the subclass and introduces intentional error in those implemented methods. After injecting intentional errors, Error Injector runs the test classes of the project to identify in which subclasses the error actually invokes. At last, from the code metrics identified by source code parser and error injector, Code Smell Calculator identifies refused bequest code smell affected classes.

Table 4.1 shows the components of the proposed approach along with their responsibility.

TABLE 4.1: Components of Proposed Approach and Their Responsibility

Sr. No.	Component Name	Responsibility
1	Change Collector	Identifies the parts that have been changed from a previous version
2	Source Code Parser	Parses source codes to identify metrics such as inherited method, overridden method, super classs method invocation etc.
3	Error Injector	Implements the inherited classes that are not overridden by the subclass and introduces intentional error in those implemented methods
4	Smell Calculator	Identifies refused bequest code smell affected classes

4.2 Description of the Proposed Approach

The total work flow of the proposed approach is shown in Figure 4.2.

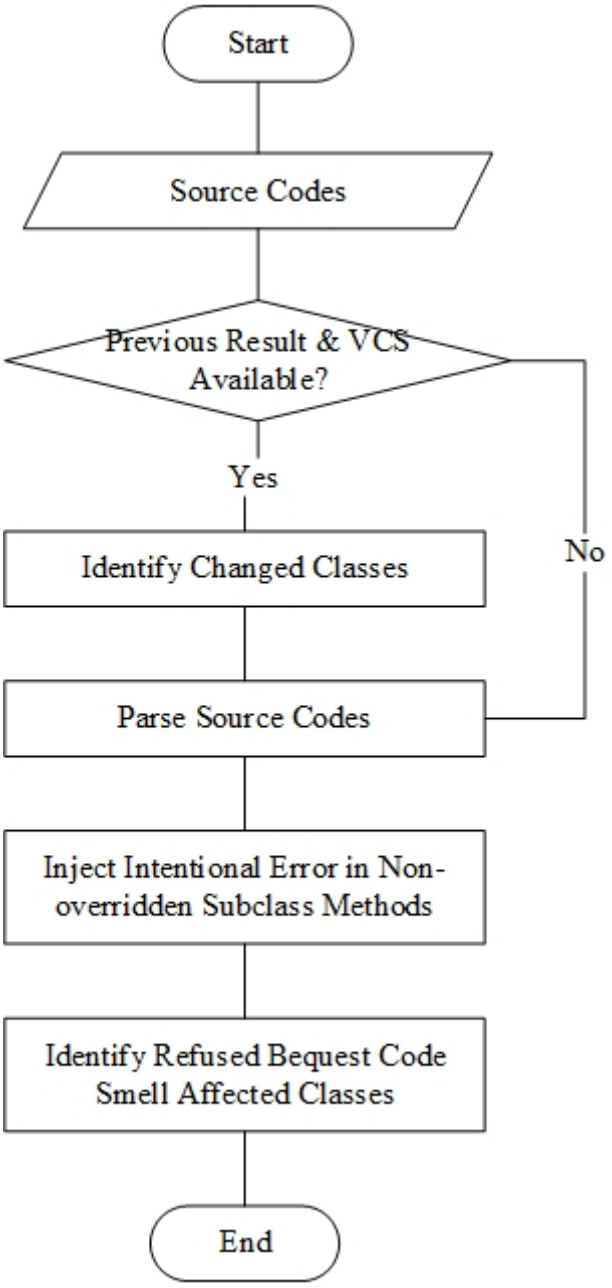

FIGURE 4.2: Flow Chart of Refused Bequest Code Smell Identification using Version Control System History

The approach starts with identifying whether Version Control System is used and refused bequest code smell algorithm was previously run on this project. If

both these conditions are true, the approach delegates the procedure to Change Collector. Otherwise, the approach delegates the procedure to Source Code Parser.

4.2.1 Change Collector

Change Collector executes the below command to identify changed classes from Version Control System (VCS) history.

```
git diff --name-only HEAD~{version}
```

The command provides the names of the classes that have been changed since the given {version}.

After getting the class names from Version Control System history, these classes are retrieved from source codes and used as input to Algorithm 1.

Algorithm 1: Change Collector

Result: List of Classes

Data: List of Classes

for *class* ∈ *classes* **do**

  if *class is Interface or class is Abstract or class is not subclass* **then**

    classes ← classes \ class;

  end

  else if *class is super class* **then**

    classes ← classes ∪ subclasses of class;

  end

end

The algorithm examines each of the changed classes and checks whether a class is an interface or abstract or is not a subclass. If a class is an interface or abstract or is not subclass, the class is excluded from the list of classes. On the other hand, if a class is a super class, all the subclasses are searched and included in the list

of classes. The identified list of classes is then provided to Source Code Parser for further operations.

4.2.2 Source Code Parser

Source Code Parser takes these included classes as input. It then generates class structure of those classes.

The procedure to generate class structure is depicted in Figure 4.3.

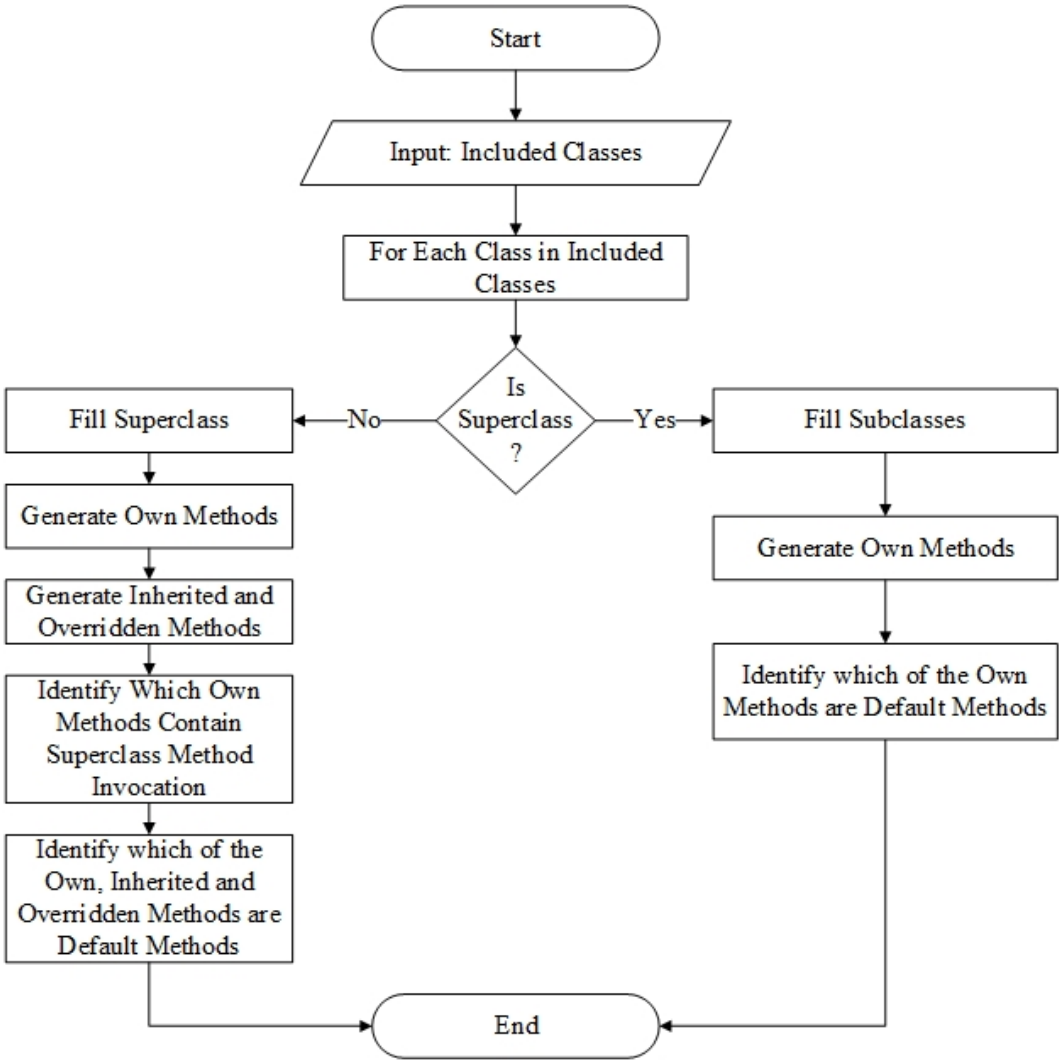

FIGURE 4.3: Flow Chart of Class Structure Generation

To understand the details of class structure generation procedure, following definitions are required.

- **Own Methods** : The methods that are created in the class under consideration.
- **Inherited Methods** : The methods that are inherited from the super class.
- **Overridden Methods** : The inherited methods that are specialized by the class under consideration.
- **Default Methods** : In some programming languages such as Java, each of the class is implicitly inherited from Object super class. So, class may contain some methods such as toString(), notify() etc. that are inherited from Object. These methods are known as Default Methods.

The generated class structure has the following format.

```
String className;
List<MethodDeclaration> ownMethods;
List<MethodDeclaration> inheritedMethods;
List<MethodDeclaration> overriddenMethods;
List<String> superClasses;
List<String> subClasses;
boolean isSubClass;
boolean isAbstract;
int containsSuper = 0;
int clientInvokes = 0;
```

The generated class structures are used as input for Error Injector component.

4.2.3 Error Injector

The work flow of Error Injector is depicted in Figure 4.4.

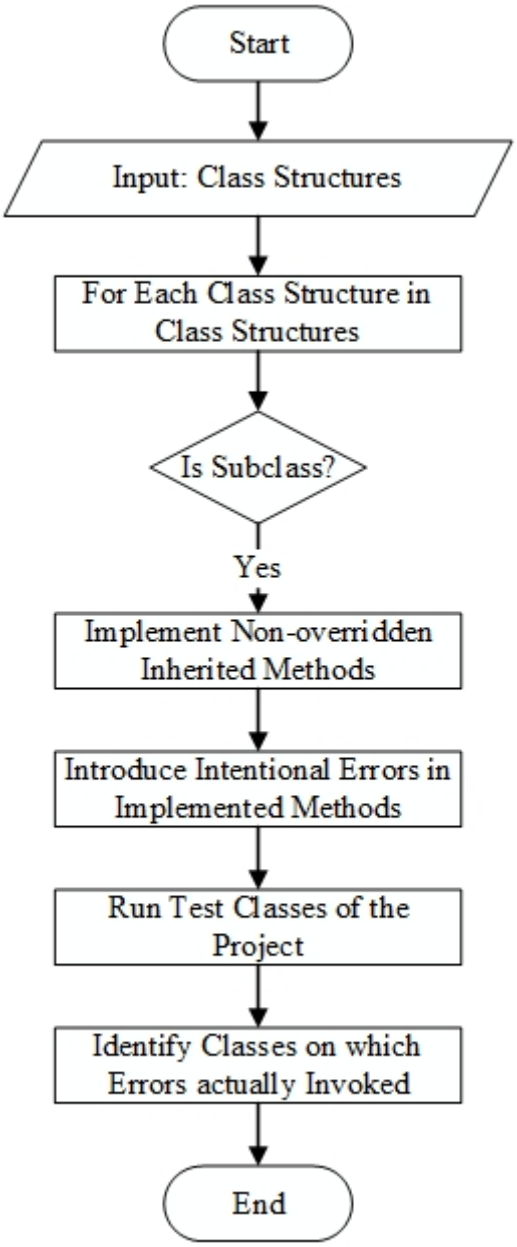

FIGURE 4.4: Flow Chart of Error Injector Component

Error Injector takes list of class structures as input. It then implements the non-overridden methods of the subclasses. When implementing the methods, Error Injector introduces an intentional error in the method. The method along with the intentional error is shown below.

```
access_modifier return_type method_name(parameters) {
    try {
        int x = 1/0;
    } catch(Exception e) {
          System.out.println(e.getStackTraces[0].
          getFileName());
    }
    return return_value;
}
```

After injecting intentional errors, Error Injector runs all the test classes of the given project. The implication is that, if these error injected methods are employed by client classes, these errors will be invoked. Thus, the number of error injected methods in a class that are actually invoked will be saved as *clientInvokes* in corresponding class structure. At the end of the procedure, Error Injector undoes all the changes that it commits while implementing inherited non-overridden methods.

4.2.4 Code Smell Calculator

Code Smell Calculator identifies which of the subclasses contain refused bequest code smell. It takes generated class structures as input and identifies the existence of refused bequest code smell from the characteristics of the class structures. It also ranks the severity of the smell. For ranking the smell severity, methodology discussed in [1] is used. The procedure to identify refused bequest code smell along with the smell severity is given below.

- **Severity Level I** : Subclasses override one or more super class methods. However, no error is invoked in the implemented inherited methods. This

scenario represents that there is no failure as no non-overridden methods are occupied by the clients. This shows a stronger sign that the subclass is refusing these methods. However, the subclass also overrides some methods at the same time. It indicates that subclass has some specific implementation for its own functionality, which justifies the necessity of inheritance.

- **Severity Level II** : Subclasses do not override any super class method. However, some errors are invoked in the implemented inherited methods. As subclass does not override any method, then it must add some new methods. And in these methods, some super class methods are invoked using super keyword. Also, failures indicate that some super class methods are employed by the clients. It implies that there are certain degrees of reuse of the methods.

- **Severity Level III** : Subclasses do not override any super class method and no own method of subclass invokes super class method. However, some errors are invoked in the implemented inherited methods. It is almost visible that refused bequest code smell is present. Because, there is no existence of generalization in the hierarchy. However, some failures indicate that there is some level of reuse in the hierarchy.

- **Severity Level IV** : Subclasses do not override any super class method and there is no intentional error invoked. So, there is no presence of generalization in the hierarchy. Also, no failure indicates that clients of the subclass do not use any of the methods provided by the super class. So, it is evident that refused bequest code smell is present in the subclass.

4.3 Summary

The proposed approach to identify refused bequest code smell using Version Control System history is described in this chapter. The core components of the approach are then discussed elaborately. The use of Version Control System history in the approach has assisted in limiting the source codes for identifying refused bequest code smells to the changed and change affected parts. The next chapter will cover result of the proposed approach and comparison with other existing approaches.

Chapter 5

Implementation and Result Analysis

This chapter aims to experimentally evaluate the performance of the proposed approach by applying it on sample projects. The approach is implemented in Java programming language. The environment setup for the experiments is discussed at first. Then a brief description of the sample projects used in the evaluation is provided. After the discussion of metrics used in the evaluation process, a thorough comparative analysis is provided for proving the effectiveness of the approach. The chapter concludes with providing the justification for the superiority of the technique introduced in this work.

5.1 Environmental Setup

This section discusses the equipments that were used to implement the proposed approach as well as to run the experimental procedures for evaluating the approach. This approach is implemented in Java programming language. As discussed in earlier chapters, refused bequest code smell originates from designing poor inheritance hierarchy where subclasses do not specialize or use the parent

methods or attributes. So, refused bequest code smell identification would be more appropriate in an object oriented programming language such as Java where the concept of inheritance is widely used.

In order to implement the approach, following tools and libraries are used -

- Eclipse Mars (4.5.0) [12]
- JavaParser (2.3.1) [13]
- JUnit (4.1.2) [14]
- Jackson JSON (2.6.3) [15]
- Git Bash (2.5.0) [16] and
- Apache Commons Codec (2.4) [17]

The experiments are performed on following PC configuration-

- 2.4GHz Intel Core i5 Quad Core
- 6GB RAM
- Windows 7 64bit
- Java SE 1.8

Before running the implementation of proposed approach on a project, some values such as project path need to be provided. Also the given project should have some predefined structure as given in Figure 5.1 for program readability.

```
Project Home
|-----src  \\source codes
|-----test \\test codes
|-----bin  \\compiled codes
```

FIGURE 5.1: Directory Structure of Project

If it does not have the project structure, the source code directory path, test code directory path and output directory path must be supplied individually.

5.2 Sample Projects

For the evaluation of the proposed approach, two open source projects were selected that contain version history. Table 5.1 shows the projects with their corresponding attributes - Line of Code, Number of Classes and Number of Versions available. All these projects were implemented considering the actual requirements, therefore were taken as sample projects for analysis.

TABLE 5.1: Experimented Projects

Project Name	Line of Code	Number of Classes	Versions
Animal World	245	8	2
Institute Hierarchy	312	12	3

"Animal World" is a simple Java project that demonstrates the use of a object oriented concept namely- inheritance in real life. There are 8 classes and 2 inheritance hierarchies in the project. There are also 2 versions and 2 test cases. The project was developed as an academic project for a course in Institute of Information Technology.

"Institute Hierarchy" is another Java project which is developed as part of a school management system. In the project, there are three types of hierarchy namely - Teacher, Student and Course. Teachers are registered for taking courses and students are enrolled in the courses. This project has 12 classes, 3 versions and 2 test cases.

5.3 Metrics Used to Analyze the Results

The performance of the proposed approach is measured by the efficiency and the accuracy of the identification process. As the project considers version control history in the identification algorithm, it should decrease the execution time. On the other hand, it should be able to identify refused bequest code smells as accurately as the existing approach.

Efficiency of the identification process is considered in terms of execution time. So, efficiency can be calculated as the ratio of difference between the execution time of existing approach and proposed approach, and the execution time of the existing approach. So, it can be written as following.

$$Efficiency = \frac{ExecutionTime_{Existing} - ExecutionTime_{Proposed}}{ExecutionTime_{Existing}} \tag{5.1}$$

The accuracy of the proposed approach is measured to be the number of refused bequest code smells identified correctly. The accuracy is calculated as ratio between the identified code smells and actual code smells. So, the accuracy can be measured using the following equation.

$$Accuracy = \frac{CodeSmells_{Identified}}{CodeSmells_{Actual}} * 100 \tag{5.2}$$

5.4 Comparative Analysis

The comparative analysis of proposed approach shows a significant contribution of proposed approach over the existing state-the-art methodologies. The existing approaches do not take into account the version history of a project to identify refused bequest code smells. However, the proposed approach employs the version history of a project for the identification process.

5.4.1 Efficiency

In order to determine the execution time, the proposed approach was run on the selected projects as shown in Table 5.1. The proposed approach was run 10 times on each of the projects and the average time is computed. Table 5.2 shows the number of code smells identified in Animal World project for its two versions and corresponding execution time for the identification process.

TABLE 5.2: Identified Smells by Proposed Approach & Average Execution Time on the PC Configuration (Project: Animal World)

Version Number	Identified Smells	Average Execution Time (seconds)
1	2	1.311
2	1	.93

For comparing the efficiency of proposed approach, one of the existing approaches [1] was executed on the same project set.

TABLE 5.3: Comparison of Execution Time between Proposed Approach and Existing Approach [1] (Project: Animal World)

Version	Identified Smells		Average Execution Time	
	Proposed Approach	Existing Approach	Proposed Approach	Existing Approach
1	2	2	1.311	1.311
2	1	1	.93	1.2

Table 5.3 shows that the average execution time on Version 1 of the Animal World project is same for both existing and proposed approach. However, the average execution time on Version 2 significantly decreases for the proposed approach with respect to existing approach.

Similarly, the proposed approach and the existing approach are run on the Institute Hierarchy project. Table 5.4 shows that the average execution time on Version 1 of the Institute Hierarchy project is same for both approaches. However, in Version 2 and 3, the execution time decreases for the proposed approach.

TABLE 5.4: Comparison of Execution Time between Proposed Approach and Existing Approach [1] (Project: Institute Hierarchy)

Version	Identified Smells		Average Execution Time	
	Proposed Approach	Existing Approach	Proposed Approach	Existing Approach
1	3	3	1.407	1.407
2	5	5	1.4	1.65
3	7	7	1.711	2.01

5.4.2 Accuracy

The accuracy of the proposed approach is measured to be the number of refused bequest code smells identified correctly. If the proposed approach increases the efficiency of the identification algorithm without considering the accuracy of the algorithm, the approach will not be desirable at all.

Table 5.5 shows that the accuracy of proposed approach and existing approach for the two versions of Animal World project. The actual smells are identified by manual inspecting the projects. In Version 1, there are actually 2 refused bequest code smells in the given project. The both approaches are able to identify those smells. There is only one refused bequest code smell in Version 2 of the given project. Similarly, both of the approaches are able to identify that smell.

TABLE 5.5: Comparison of Accuracy between the Proposed Approach and Existing Approach [1] (Project: Animal World)

Version	Identified Smells		Actual Smells	Accuracy	
	Existing Approach	Proposed Approach		Existing Approach	Proposed Approach
1	2	2	2	100%	100%
2	1	1	1	100%	100%

Similarly, Table 5.6 shows that the proposed approach is able to identify all the refused bequest code smells for all the 3 Versions of the Institute Hierarchy project. Thus, it clearly shows that the proposed approach is able to identify code smells as correctly as the existing approach.

TABLE 5.6: Comparison of Accuracy between the Proposed Approach and Existing Approach [1] (Project: Institute Hierarchy)

Version	Identified Smells		Actual Smells	Accuracy	
	Existing Approach	Proposed Approach		Existing Approach	Proposed Approach
1	3	3	3	100%	100%
2	5	5	5	100%	100%
3	7	7	7	100%	100%

5.5 Discussion of Result

This section justifies the reasons behind the obtained results from the proposed approach. The proposed approach employs version history of a given project while identifying the refused bequest code smell using a combination of static and dynamic code analysis. The implication is that the proposed approach has taken into consideration the incremental growth of real life software systems to use previously calculated results in the identification process.

As shown in Table 5.3 and Table 5.4, the average execution times for Version 2 of Animal World project, Version 2 and Version 3 of Institute Hierarchy project are significantly decreased by the proposed approach. However, the average execution time in latter versions does not vary significantly for the existing approach. This is because the existing approach runs the identification algorithm on the whole project, rather than to the parts of the project that are changed from last version. However, the proposed approach first identifies the parts that are changed from previous version and runs identification algorithm on those parts only. It then merges the existing results with the calculated results to identify refused bequest code smell affected classes. As the proposed approach is able to limit the search space of identification algorithm to changed parts only, it produces significantly good result for the versions where previous result is available.

Although the proposed approach runs the identification process on the changed parts, it is still able to identify all the refused bequest code smells accurately as

shown in Table 5.5 and 5.6. Because it takes into account all the information about the source codes by merging previous information with the new information. Thus it is able to identify all the smells correctly in less time than the existing approach.

5.6 Summary

This chapter intends to demonstrate the implementation and result analysis of the proposed approach. The proposed approach is implemented in Java programming language. The implementation results are analyzed and compared with one of the existing approach [1]. The analysis shows that the proposed approach is more time efficient than the existing approach. The proposed approach is found to be 16.2% more time efficient. Moreover, it is able to identify refused bequest code smells correctly as the exiting approach. Thus the proposed approach is more time efficient and still equally accurate in identifying code smells.

Chapter 6

Conclusion

The existing approaches for identifying refused bequest code smell do not incorporate version control history into the identification process. A new approach is proposed in the thesis for this purpose. The proposed approach uses version control history to limit the search space for refused bequest code smell identification to changed and affected parts of the source codes. In this chapter, a brief discussion on the research as well as several factors that may be proven as threat to validate the outcome of the research is provided. Future direction for research in the proposed approach is also discussed at the end of the chapter.

6.1 Discussion of the Research

This thesis introduces a new approach to identify refused bequest code smells using version control history. The existing refused bequest code smell identification approaches either use static code analysis or a combination of static and dynamic code analysis. However, these approaches do not take into account the version control history of the software systems. The implication of using version control history is that if the identification process is run on only the changed parts from a previous version, the search space or source codes to identify refused bequest

code smell will be smaller than the whole source codes. Thus, the introduction of version control history in refused bequest code smell identification can reduce the identification time. This hypothesis is established throughout the thesis.

The proposed approach broadly contains four components namely - Change Collector, Source Code Parser, Error Injector and Smell Calculator. The Change Collector identifies the parts that have been changed from the last time when the proposed approach was run on the projects. The Source Code Parser parses the source codes to identify metrics such as inherited method, overridden method, super classs method invocation etc. Then Error Injector implements the non-overridden inherited methods, introduces intentional errors in those methods and run project's test cases to identify metric namely client invokes. At last, Smell Calculator calculates refused bequest code smell for each of the subclasses from the above mentioned metrics.

The experiments conducted on the sample projects show that the proposed approach is 16.2% more time efficient than the existing approach. Moreover, it is able to identify all the refused bequest code smells correctly for the given projects.

6.2 Threats to Validity

The proposed approach tries to consider all possible cases. However, there are some cases that may be considered in the approach.

- The proposed approach is implemented in Java programming language. So, the performance of the approach may be changed for other programming languages and platforms.
- The assumption in the approach is that there will be extensive test cases for the functionalities of the classes. If there are not enough test cases to reflect the functionalities, the dynamic analysis will not be fruitful in determining

the client usage of subclass's methods. However, ensuring the coverage of test cases is a different research topic.

- The proposed approach is experimented on a limited set of projects. How the proposed approach would work for other projects is one of the future works.

- The accuracy of the proposed approach is determined by manually inspecting the source codes. However, there is no standard process of identifying the actual code smells from the source codes.

6.3 Future Work

The incorporation of version control history in code smell identification provides a number of directions for future research. In this thesis, an approach incorporating version control history is proposed for refused bequest code smell identification. The core idea of the approach can be incorporated to reduce the execution time of other code smells such as parallel inheritance, shotgun surgery, feature envy etc. identification algorithms.

The implementation of proposed approach evaluated the performance of the approach on Java projects only. The future challenge lies in achieving the desired performance on different platforms rather than Java only.

The proposed approach considers class as granularity level or computation unit for the identification process. The future research on the proposed approach may try to discover the effect on efficiency by changing the granularity to method level.

Bibliography

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

[2] P. F. Mihancea, "Towards a client driven characterization of class hierarchies," in *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*, pp. 285–294, IEEE, 2006.

[3] 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.

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

[5] M. Fowler, K. Beck, J. Brant, and W. Opdyke, "Refactoring: Improving the design of existing code,"

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

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

[8] A. van Deursen, L. Moonen, A. van den Bergh, and G. Kok, *Refactoring test code*. CWI, 2001.

[9] H. V. Nguyen, H. A. Nguyen, T. T. Nguyen, A. T. Nguyen, and T. N. Nguyen, "Detection of embedded code smells in dynamic web applications," in *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*, pp. 282–285, IEEE, 2012.

[10] M. Cherubini, G. Venolia, R. DeLine, and A. J. Ko, "Let's go to the whiteboard: how and why software developers use drawings," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, pp. 557–566, ACM, 2007.

[11] 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.

[12] E. Foundation, "Eclipse mars," dec 2015.

[13] F. Tomassetti, "Java parser and abstract syntax tree," oct 2015.

[14] T. D. Stefan Birkner, "Junit 4," oct 2015.

[15] T. Saloranta, "Jackson json," oct 2015.

[16] G. Team, "Git bash," sep 2015.

[17] A. Foundation, "Apache commons codec," oct 2015.