ABSTRACT SYNTAX TREE-BASED CODE SMELL DETECTION AND REFACTORING

A THESIS

Presented to the Department of Computer Engineering and Computer Science

California State University, Long Beach

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

Committee Members:

Frank Murgolo, Ph.D. (Chair)
Birgit Penzenstadler, Ph.D.
Shui Lam, Ph.D.

College Designee:

Antonella Sciortino, Ph.D.

By Aditi aka Palak Patodiya aka Patoliya

B.Tech., 2015, Charotar University of Science and Technology, India

August 2018

ABSTRACT

ABSTRACT SYNTAX TREE-BASED CODE SMELL DETECTION AND REFACTORING

By

Aditi aka Palak Patodiya aka Patoliya

August 2018

Software frameworks are the crux of the numerous businesses in giant technological industries. These software systems are growing in size, becoming more perplexing day by day. Besides, they are subject to frequent alterations that implement new features or resolve bugs. Due to time constraints, engineers do not have enough resources to plan and implement perfect solutions, which results in the occurrence of code smells in the software system.

With the increase of the size of software systems, it is strenuous to detect the code smells in the software and refactor the code manually. This thesis presents a framework that detects seven code smells (Switch case, Long parameter list, Middle man, Long method, Temporary fields, Message chains, and Data class) and provides refactoring suggestions for the code that contains these smells. These smells are detected with the help of metrics and software visualizations generated by an abstract syntax tree.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

SDLC	Software Development Life Cycle
AST	Abstract Syntax Tree
DBR	Defect-Based Reading
PBR	Perspective-Based Reading
UBR	Use-Based Reading
SBR	State-Based Reading
UI	User Interface

CHAPTER 1

INTRODUCTION

Motivation

The methodology of software development has evolved over many years. Software products encounter different changes during their software development life-cycle (SDLC). The fundamental idea of the SDLC has been to seek the improvement of software frameworks in an extremely premeditated, organized, and systematic way, requiring each phase of the life cycle – from the beginning of the plan to the delivery of the final product – to be carried out strictly and consecutively within the context of the structure that is being connected [1]. The major phases of the SDLC are design, development, and maintenance. A software system becomes noticeably harder to maintain as it develops over time. Its design becomes increasingly complicated and difficult to understand. These changes incite a steady degradation of the software product. Due to the time constraints, developers do not have adequate time to plan and create proficient solutions. The software development process is then performed in a surge, prompting undisciplined execution decisions, which causes the introduction of technical debt. Technical debt is the extra maintenance work that is caused by the use of code that is easy to implement in the limited time instead of writing the code that would provide effective solutions for the prevalent issues [2].

Software maintenance is expensive. The aggregate expense for maintenance of a software product is 40–70% of the total cost of the lifecycle of the product [3]. Subsequently, decreasing the exertion spent on maintenance can be viewed as a method for reducing the general expenses of a software product. Due to these reasons, code smell detection and refactoring has been an area of interest for many researchers. According to Fowler et al., “Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code

yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs" [4]. Code smells are not bugs and do not prevent the software code from functioning properly, but they indicate the weaknesses of the software design, which may hinder the process of its development and increase the risk of bugs and failures in the future [5]. According to Fowler et al., "a code smell is a surface indication that usually corresponds to a deeper problem in the system" [4]. Code smells are specific patterns in object-oriented software systems that cause troubles in the maintenance of such systems [5]. According to Fowler et al., there are 22 code smells. These code smells can be divided into seven different categories: Bloaters, Object-Oriented Abusers, Change Preventers, Dispensables, Encapsulators, Couplers, and Others [6].

Refactoring is the answer to the issue demonstrated by the code smell. Code refactoring is the process of rebuilding the existing code without changing its external conduct. Refactoring incorporates an enhanced code clarity that, in turn, helps in the easy maintenance of the code. Refactoring is carried out in three phases: locating the bad smells, selecting the appropriate refactoring for the bad smell, and, finally, implementing the selected refactoring. The accurate identification of code smells is the fundamental requirement for refactoring. The refactoring techniques are dependent on the code smells found in the code. Presently, there are some implementations that distinguish code smells, but all the tools allow the detection of only a few code smells [7]. The tool proposed in this thesis helps the detection of a larger set of code smells and suggests ways to refactor the code by using design patterns.

Problem Statement

Recurrent refactoring is a technique that is used for preserving and improving the design quality of a software product. Errorless refactoring can be achieved by employing proficient code

smell detection. Such code smell detection techniques need to collaborate well with refactoring techniques to provide ample ideas in order to enable the client to comprehend the design flaws. Moreover, the procedure of code smell detection and its removal using refactoring is complex. Without the understanding of the outline of the specific software product, the chances of breaking the code and degradation of the design are greater. Incorrect refactoring of the code can change its behavior. Numerous refactoring tools have been produced, and there are many code smell detection tools that have been proposed by researchers in the past [8]. In most of these cases, the tools work independently. A few attempts have been made in the areas of combining the tools for code smell detection and refactoring.

The primary purpose of this thesis is to focus on developing a tool that detects multiple code smells and provides refactoring solutions for them using design patterns. The tool automatically detects seven different bad smells and provides refactoring solutions to the code. The target language for this code is Java, but the underlying algorithms can be used for the detection of code smells in other object-oriented languages as well. In order for the tool to detect code smells, the path of the source folder of the project is given as an input to the tool. All the Java files in the source folder are parsed with the help of an Abstract Syntax Tree (AST), and the bad smells are detected using various algorithm implementations. Apart from this, there are other contributions of this thesis:

1. Analysis of the correlation between code smells and refactoring techniques.
2. Algorithms to detect specific code smells.
3. Major reasons for code smell introduction in the code.
4. Computer-aided refactoring of the bad smells.

Thesis Outline

The points described below outline the remainder of the report:

- Chapter 2 discusses the previous work, and related research in the field of code smells and refactoring. It also discusses the various code smell detection techniques that have been used in the past in depth.
- Chapter 3 consists of the analysis of different code smells and discusses their classification into various categories. It also discusses an empirical study that analyzes when and why code smells are introduced in the code.
- In Chapter 4, an analysis is carried out on the various refactoring techniques; their complications and the conditions that maintain the underlying semantics are studied. It clarifies the significance of semantic checks and demonstrates cases of the manner in which refactoring could influence the conduct of the code.
- Chapter 5 discusses the architecture of AST along with a demonstration of ASTVisitor and how it is used in this thesis.
- Chapter 6 presents the framework of the code smell detection tool that detects seven different types of code smells in the Java source code and provides refactoring suggestions for the code that contains the smells. This chapter also discusses the results obtained by verifying the tool against different test cases.
- Chapter 7 draws the conclusion of this thesis and suggests the work that can be accomplished in future.

CHAPTER 2

RELATED WORK

This section discusses the previous work in the field of code smell detection. It also reviews the different refactoring techniques used for code smell removal and improving the code quality. In this chapter, an extensive review is conducted for the analysis of the different methods used for detecting code smells and anti-patterns as well as their influence on the non-functional attributes of source code management and maintenance. The presence of a code smell is an indication that there are possibilities of an error occurring in the code. The code smell is not an error in itself but may result in an error if not handled carefully. An anti-pattern is a pre-defined template for a solution to a commonly occurring problem, which is ineffective and increases the risk even further [9].

According to Fowler, "a code smell is a surface indication that usually corresponds to a deeper problem in the system" [2]. In the book, *Refactoring: Improving the Design of Existing Code,* Fowler et al. introduced 22 different code smells, which are discussed in depth in the later sections [4]. Code smell can also be defined in terms of design principles and quality as "smells are certain structures in the code that indicate the violation of fundamental design principles and negatively impact design quality" [10]. Code smells are not bugs in the system and do not affect the functional behavior of the code; they are design deficiencies in the code, which slow down the maintenance process and introduce the risk of errors occurring in the future. In 2002, Van Emden and Moonen provided the initial formal elucidation of code smell detection techniques [11]. They approached the problem by dividing the code smells into two groups: primitive smell aspects and derived smell aspects. They constructed the source model of the system by inspecting the primitive smell aspects and derived smell aspects that was later used for code

smell detection. In 2006, Mantyla and Lassenius investigated the approaches used by developers to distinguish the code smells and compared them to the detection processes carried out by automatic detection tools [12]. A recent study revealed that code smells accumulate in the source-code over time. They are first introduced upon the initial writing of the code snippet that contains the smell. These code smells are not removed due to the requirement of additional refactoring work [13].

The presence of code smells makes the code vulnerable to error. Such a type of code is complex to manage. In the past, there was a significant amount of research conducted on the efforts for code smell detections. The techniques used for detection can be classified into various classes, such as manual detection techniques, visualization-based detection technique, semi-automatic techniques, automatic techniques, empirical-based techniques, and metric-based techniques [14].

The initial approach used by researchers and developers for code smell detection was manual detection. A manual detection technique required human interaction, and while it was time-consuming, it formed the foundation for different techniques such as the semi-automatic and automatic techniques. James Noble presented the relationship between the design patterns by classifying the design patterns into two groups: primary relationships and secondary relationships [15]. Relationships such as “pattern uses another pattern,” “pattern refines another pattern” and “pattern conflicts another pattern” are primary relationships. Relationships such as “pattern similar to another pattern” and “pattern combining with another pattern” are secondary relationships. Other secondary relationships between the design patterns defined by him are “variant,” “requires,” “tiling,” and “sequence of elaboration.” Travassos et al. detected bad smells in object-oriented design by providing a set of reading techniques [16]. Software reading

techniques endeavor to improve the viability of the analysis by providing a set of rules that can be utilized in the analysis of a given piece of code. The reading techniques used by Travassos et al. for bad smell detection are Defect-Based Reading (DBR), Perspective-Based Reading (PBR), and Use-Based Reading (UBR). DBR detected smells in the requirements that were presented using a state machine notation called State-Based Reading (SBR). Similarly, PBR detected smells in the requirements, but the requirements were represented using natural language. UBR detected the anomalies in user interfaces.

In the past, many researchers used visualization-based techniques to detect the smells in the source code. These techniques use methods that are visualization-oriented, such as graphs and UML diagrams, to detect the design defects. Langlier et al. provided a visualization-based approach for the quality assurance of complex systems. They described four features in their visualization framework: class representation, program representation, navigation, and data filtering [17]. They used a geometrical 3D box for class representation because it can be used very efficiently owing to the fact that multiple entities can be represented by it. The qualities measured by them in the class representation using the geometrical 3D box are cohesion, coupling, inheritance, and size complexity. Tekin et al. suggested an approach for mining an object-oriented design structure [8]. They used sub-graphs to detect similar design structures in the source code. In a visualization-based approach, using geometrical structures leads to a clear understanding of the frequently used designs for software development as well as the model-specific patterns that help engineers to improve the product quality. Moreover, the identification of similar design patterns and anti-patterns leads to the formulation of easier refactoring solutions. Tekin et al. experimented with this framework on many open-source projects and successfully found identical design patterns and anti-patterns in the code.

Recently, developers and researchers have turned towards semi-automatic and automatic detection techniques for bad smell detection. These detection techniques use automatized models with minimal human interactions. Lakshmanan and Manikandan proposed an approach for automatic code smell detection and refactoring [18]. They built the resolution sequence of code smells based on a priority that is calculated by applying topological sorting to the code smells. To remove these code smells, they performed multi-step refactoring, and its steps included model evaluation, extract design, preferred design, and refactoring. The results of this experiment using the prioritized sequence of code smells and multi-step refactoring proved to work better than other semi-automatic approaches. Another automatic refactoring work was proposed by Fokaefs et al. [19]. They presented a tool developed as an Eclipse plug-in that recognizes Extract Class refactoring by using the clustering algorithm, which identifies closely coupled classes within the system. Apart from this, their tool allocated rankings to the refactoring techniques based on the amount of refinement a particular technique brings to the system. The ranking procedure for the refactoring techniques was carried out by using the Entity Placement metric, which measures the overall design quality of the system. Orchard and Rice developed a tool named CamFort, which is a refactoring tool for source code written in Fortran. This open-source automatic refactoring tool focuses on domain-related features and design models related to memory and data management. CamFort provided three types of refactoring solutions: equivalence statement elimination, common block elimination, and derived data type introduction [20].

Apart from manual and automatic code smell detection, there is metric-based code smell detection in which different types of metrics are used to measure the different characteristics of the refactoring techniques and code smells. Also, many researchers use the metric-based system

to measure the quality of code. Dijkman et al. built a metric-based system that can be utilized to correct the anti-patterns and, additionally, the degree to which the processes overlap [21]. They assessed it by applying it to two vast process architectures. The assessment demonstrated that the procedure could be utilized to pinpoint the exact area of three refactoring techniques with high accuracy. Elish and Alshayeb categorized the refactoring techniques based on metrics such as internal quality measurements and code testing exertion. This demonstrated an accurate representation of the refactoring techniques to be used in order to optimize the software architecture [22]. Huston suggested an approach for determining the compatibility between the design patterns and the design metrics [23]. He analyzed the situations where the design pattern and metrics are in contradiction. He analyzed three design patterns against the anti-patterns: Mediator, Bridge, and Visitor. The results of his experiments demonstrate that pattern-metric incongruity can be reduced if the reduced high metric score is used to determine the pattern usage.

A software system can be made more maintainable by applying various refactoring techniques to the code. Various code modifications are conducted by applying multiple refactoring techniques to different segments of code. Normally, code smells are recognized in programming, due to which engineers consider numerous refactoring techniques to improve the quality of the software. This leads to a refactoring chain that decreases the time complexity and cost of system maintenance. The code smells can be withdrawn, and the time required for making changes in the code can be reduced by applying suitable refactoring techniques. Harman and Tratt presented an approach for search-based refactoring using Pareto optimality [24]. Executing the search-based refactoring framework iteratively led to the generation of a Pareto front, the estimations of which leads to the Pareto ideal chain of refactoring techniques. The

Pareto front expands the different metrics that are employed in order to decide the refactoring techniques to be used. They also demonstrate the manner in which the generation of a Pareto front reduces the requirement for complex blends of metrics.

CHAPTER 3

CODE SMELLS

The term “code smell” was first coined by Fowler et al. [4]. A code smell is a signal in the source code that demonstrates potential issues, and it is the design flow that indicates the necessity for refactoring. Code smells are the indications to the engineer that some sections of the code might be prone to errors in the future. These code smells need to be fixed to increase understandability and maintainability of the code. To carry out the detection and refactoring of code smells effectively, this chapter discusses briefly when and why code smells are introduced. Additionally, code smells are classified into different categories for better understandability.

When and Why Are Code Smells Introduced?

Code smells are the manifestations of faulty and improper design and implementation practices. A code smell is one of the critical factors that results in technical debt and also influences the maintainability of the software [25]. There are some reasons for the occurrence of code smells in the code, including critical maintenance actions and the requirements of delivering the product in a situation where a shorter time-to-market is preferred over code quality. Therefore, software evolution can be considered a prime reason for the introduction of code smells in the code.

According to the study conducted by Tufano et al., most of the code smells are introduced just before the release of the product [26]. The amount of smells introduced in the initial stages is low. This indicates the fact that the pressure to deliver the product on time can be one of the causes of code smells. Their study also suggests that the number of smells introduced during the enhancement of a feature in a software product are more compared to the number of smells introduced during the initial development of the code. Enhancement of a feature can be

defined as any change applied to the existing code in an attempt to improve the product. In addition, the cost of refactoring the code is greater when the possibility of having side-effects outweighs the benefits. Further, the tools available in the market for code smell detection and refactoring are inefficient to use for the developers of software products. As a result, there is a high increment in the number of code smells within the code.

Classification of Code Smells

Based on Properties

In 2000, Fowler et al. introduced 22 different types of code smells [4]. According to Mantyla et al., the flat list of code smells makes it complex to understand and determine the relationship between the different code smells [6]. Therefore, they divided the code smells into seven categories depending upon the properties and the nature of code smells. Table 1 shows the classification of code smells based on properties.

Bloaters. Bloaters are the set of code smells that increases the code size to such an extent that it becomes difficult to cope with the code. Bloaters include five code smells:

- *Long method*: A method, procedure, or function that contains too many lines of code is considered to be a long method. When a method is extensively long, it uses a large number of variables and performs multiple operations. It is plausible that the method does more than what its name suggests. When a method is small, understandability and maintainability of the code increases.
- *Large class*: Similar to the long method code smell, the large class code smell occurs when the class is large and contains too many methods, instances, and variables. It is challenging to understand such classes, and in case of errors in these classes, the finding and resolution of these errors is complex.

TABLE 1. Classification of Code Smells Based on Properties

Categories	Code smells
Bloaters	Long Method
	Large Class
	Primitive Obsession
	Long Parameter List
	Data Clumps
Change Preventers	Divergent Change
	Shotgun Surgery
Couplers	Feature Envy
	Inappropriate Intimacy
	Message Chain
	Middle Man
Dispensables	Lazy Class
	Data Class
	Duplicate Code
	Speculative Generality
Object-Oriented Abusers	Switch Statements
	Temporary Field
	Refused Bequest
	Alternative Classes with Different Interfaces
	Parallel Inheritance Hierarchies
Others	Incomplete Library Class
	Comments

- *Primitive obsession*: A primitive obsession bad smell occurs when the primitive data types are used excessively for the rendering of the domain ideas. Primitive data types are the building blocks of a language. When the data structure becomes complex, using primitive data types is much easier compared to creating a new class. Excessive usage of primitive data types leads to the primitive obsession bad smell. As a result of the primitive obsession code smell, the code becomes inflexible and harder to control. Additionally, the ease of the object-oriented design is lost, and it tends to behave like a procedural code.
- *Long parameter list*: When more than three or four parameters are passed as an argument to a method, the long parameter list bad smell occurs. When there is a need for additional data in the method, an object can be used instead of passing that data as an argument. A long list of parameters is hard to understand, and it becomes harder to maintain as it increases in size.
- *Data clumps*: When there are similar data types at multiple parts of the code, the data clumps bad smell occurs. When a code is treated by removing the data clumps bad smell, the code reduces in size as well as renders a better understanding and organization.

Change preventers. Code smells classified in this category make software modification difficult. In presence of these code smells, when a single block of code is changed, many other parts of the code also require modifications.

- *Divergent change*: When a single class is changed in multiple ways for different reasons, the divergent change bad smell occurs. When a single class is modified for different reasons, it can be assumed that the class has too many responsibilities. In many instances, the divergent change is the result of poor programming practices.

- *Shotgun surgery*: Shotgun surgery is similar to the divergent change code smell. When the divergent change code smell is present, multiple changes are made to a single class, while when the shotgun surgery code smell is present, a single change is made to multiple classes in cases where there is excessive coupling between the classes and a single responsibility is shared among multiple classes.

Couplers. As the name suggests, this group of code smells occur as a result of excessive coupling between classes.

- *Feature envy*: The feature envy code smell occurs when a method uses more objects of another class as compared to its own class. This situation may occur when the variables and methods are moved to a data class.
- *Inappropriate intimacy*: The inappropriate intimacy code smell occurs when two classes are highly coupled, and the methods of these classes use each other's private variables.
- *Message chain*: When a class has a high coupling with other classes in the form of a chain, the message chain bad smell occurs. A message chain occurs when an object of one class calls an object of another class that, in turn, calls an object of another class and so on. Due to the high coupling, the main class requires changes when the intermediate classes are changed.
- *Middle man*: The middle man bad smell occurs when a method or a class passes most of its requests to another class or method. A middle man is a class or a method that is not effectively contributing to the system. Removing the middle man smell results in a cleaner and more understandable code.

Dispensables. Code smells classified in this category contributes to a trivial and inessential code whose absence in the code will make it more efficient and understandable.

- *Lazy class*: A class is considered to be a lazy class when it does not perform enough functions. Such a situation arises after the class has been refactored and downsized or if the class has been developed to support future needs that never came to be.
- *Data class*: A class that contains only data fields and methods like getters and setters for accessing those data fields is a data class. A data class is generally a repository for data to be utilized by another class. These classes do not have any auxiliary usefulness due to which they cannot act on their own, and it leads to an increase in the coupling between the classes.
- *Duplicate code*: The duplicate code bad smell occurs when there are multiple code fragments that perform the same function in the source code. Code duplication occurs if different programmers are unaware of each other's work.
- *Speculative generality*: The speculative generality code smell occurs when there is unused code written to support the future needs that were never implemented. It makes the code harder to understand and maintain.

Object-oriented abusers. All the smells classified as object-oriented abusers occur when the concepts associated with the object-oriented paradigm are applied incorrectly.

- *Switch statements*: A good case of object-oriented programming indicates the lesser usage of switch cases. Generally, there are many switch cases across the code when a single case is added to satisfy all conditions. When a new condition is added, all the switch cases across the code need modification.
- *Temporary field*: The temporary field bad smell occurs when specific variables get their value only in particular circumstances. This happens when these variables are used only in a certain method. Thus, variables receive their values only when a method is called.

- *Refused bequest*: When a subclass uses only a few methods of its parent class, the refused bequest smell occurs. The main objective behind inheritance is code reuse. When a subclass does not use enough code from its parent class and has its own methods, it leads to less reuse of code, which contradicts the inheritance paradigm.
- *Alternative classes with different interfaces*: When multiple methods perform similar functions but have different names, the alternative classes with different interface smell occurs. It increases code duplication and causes the code to become incompressible.

Others.

- *Incomplete library class*: Built-in libraries in a programming language are useful as they reduce the effort required by the developer to implement various functionalities. When the applicability provided by the built-in library classes stop meeting the needs of a developer, the incomplete library class code smell occurs due to these libraries being only readable. As a result, it is not possible to modify them based on the individual's requirements.
- *Comments*: Including comments in the code is a sign of good documentation, but the excessive use of comments results in the comments code smell. When there are a large number of comments, the code becomes arduous to understand.

Based on Coverage

Depending upon the coverage of the code smells, they are classified into two categories [27]:

1. Code smell between classes.
2. Code smell within a class.

In a large-scale software application with multiple classes, there are object references and method calls between the classes. The code smells, such as the message chain and the middle man, generated in such situations are the result of multiple classes. Such code smells are categorized into the first category. On the other hand, code smells within the class affect only the class that contains it. Table 2 shows the classification of code smells based on their coverage.

TABLE 2. Classification of Code Smells Based on Coverage

Code smells between classes	Code smells within classes
Alternative Classes with Different Interfaces	Comments
Primitive Obsession	Long Method
Feature Envy	Long Parameter List
Lazy Class	Duplicated Code
Message Chains	Conditional Complexity
Middle Man	Combinatorial Explosion
Divergent Change	Large Class
Shotgun Surgery	Type Embedded in Name
Parallel Inheritance Hierarchies	Uncommunicative Name
Incomplete Library Classes	Inconsistent Name
Data Class	Dead Code
Data Clumps	Speculative Generality
Refused Bequest	Oddball Solution
Inappropriate Intimacy	Temporary Field
Indecent Exposure	

CHAPTER 4

REFACTORING TECHNIQUES

Due to the variety of reasons, there can be many smells in the code, and engineers are expected to clean the code. Code refactoring is the systematic procedure of taking code that falls short of an ideal design and progressively improving it to a better design. This is achieved by changing the internal structure of the code while keeping the external behavior of the code intact. It reduces the chances of the occurrence of bugs in the code and increases the maintainability and understandability of code [4]. In order to ensure the consistent behavior of the code, automated tests should be carried out after every step of any refactoring technique. Fowler et al. cataloged 72 refactoring techniques and classified them into seven categories as shown in Table 3 [4].

Out of these refactoring techniques, we will discuss 12 refactoring techniques in this chapter. These refactoring techniques can be utilized to remove the bad smells discussed in this thesis.

Rename Method

When the name of the method does not signify the intention completely, the rename method refactoring can be applied. A poor naming convention can lead to confusion and increase the time that is required for the understanding and maintenance of the code. Inefficient method names can occur when the programming is carried out in a rush or the method name stops providing enough insight as the functionalities of the method evolve.

Refactoring Procedure

- If the method is defined in a superclass or subclass, repeat all the steps.
- Declare a new method with a meaningful name and copy the code from the old method to the new method and replace the body of the old method with a call to the new method.

TABLE 3. Classification of Refactoring Techniques

Categories	Refactoring techniques
Composing Methods	Extract Method
	Replace Method with Method Object
	Inline Method
	Replace Temp with Query
	Inline Temp
	Split Temporary Variables
	Introduce Explaining Variables
	Substitute Algorithms
	Remove Assignments to Parameters
Big Refactoring	Convert Procedural Design to Object
	Tease Apart Inheritance
	Extract Hierarchy
	The Nature of the Game
	Separate Domain from Presentation
Organizing Data	Replace Data Value with Object
	Change Reference to Value
	Duplicate Observed Data
	Encapsulate Collection
	Encapsulate Field
	Replace Array with Object
	Replace Data Value with Object
	Replace Magic Number with Symbolic Constants
	Replace Record with Data Class

TABLE 3. Continued

Categories	Refactoring techniques
	Replace Subclass with Fields
	Replace Data Value with Object
	Replace Magic Number with Symbolic Constants
	Replace Record with Data Class
	Replace Subclass with Fields
	Replace Type Code with Class
	Replace Type Code with State/Strategy
	Replace Type Code with Subclasses
	Self-Encapsulate Field
Moving Features between Objects	Extract Class
	Introduce Local Extension
	Hide Delegate
	Move Field
	Inline class
	Move Method
	Introduce Foreign Method
	Remove Middle Man
Making Method Calls Simpler	Add Parameter
	Rename Method
	Encapsulate Downcast
	Replace Constructor with Factory Method
	Hide Method
	Replace Error Code with Exception
	Introduce Parameter Object

TABLE 3. Continued

Categories	Refactoring Techniques
	Replace Exception with Test
	Parameterize Method
	Replace Parameter with Explicit Methods
	Preserve Whole Object
	Replace Parameter with Method
	Remove Parameter
	Separate Query from Modifier
	Remove String Method
Dealing with Generalization	Collapse Hierarchy
	Pull Up Field
	Extract Interface
	Pull up Method
	Extract Subclass
	Push Down Field
	Extract Superclass
	Push Down Method
	Form Template Method
	Replace Delegation with Inheritance
	Pull Up Constructor Body
	Replace Inheritance with Delegation
Simplifying Conditional Expressions	Consolidate Conditional Expression 18
	Consolidate Duplicate Conditional Fragments
	Decompose Conditional
	Introduce Assertion

TABLE 3. Continued

Categories	Refactoring Techniques
	Introduce Null Object
	Remove Control Flag
	Replace Conditional with Polymorphism
	Replace Nested Conditional

- Run tests.
- Obtain all the references to the old method and replace them with the references to the new method.
- Delete the old method.
- Compile and test.

Example

Figure 1 shows the bad code with the incomplete method name. Figure 2 shows the code refactored with the rename method refactoring technique.

```
public Boolean fnd_emp() {
   if(location.equals("California")) {
                  return true;
   }
   return false;
}
```

FIGURE 1. Bad code – rename method.

Move Method

The move method refactoring can be applied to reduce the coupling between the classes. If a method uses an object of another class more than the object of the class in which it is

defined, the move method should be applied. When method calls are made more frequently from another class, moving method to that class decreases the dependency between the classes.

```
public Boolean isEmployeFromCalifornia() {
    if(location.equals("California")){
        return true;
    }
    return false;
}
```

FIGURE 2. Refactored code – rename method.

Refactoring Procedure

- Check all the attributes used by the method. If there are a few attributes used by the method or the attributes are used only by that method, then these attributes should be moved along with the method.
- Declare a method in the new class and provide a self-explanatory name.
- Replicate all the code from the original method to the new method.
- Replace the code of the source method with the call to the new method.
- Run tests.
- Change all the old method references to the new method references.
- Remove the old method.
- Compile and test.

Example

Figure 3 and Figure 4 are the examples of bad code and refactored code respectively. The code shown in Figure 3 contains a method that invokes other classes more than its own class. It is refactored using the move method refactoring technique as shown in Figure 4.

```
public class Employee {
  private EmployeeBalance _balance;
  private int _daysOutbound;

  double outboundCharge() {
    if(_balance.isPrime()) {
      double result = 15;
      if(_daysOutbound > 5) {
        result += (_daysOutbound - 7) * 0.5;
        return result;
      }
    }
    else
      return _daysOutbound*1.5;
  }

  double bankCost() {
    double result = 10;
    if(_daysOutbound > 0)
      result += outboundCharge();
    return result;
  }
```

FIGURE 3. Bad code – move method.

```
class EmployeeBalance {

  double outboundCharge(Employee employee) {
    if(isPrime()) {
      double result = 15;
      if(employee.getDaysOutbound() > 5) {
        result += (employee.getDaysOutbound() - 7) * 0.5;
        return result;
      }
    }
    else
      return employee.getDaysOutbound()*1.5;
  }
```

FIGURE 4. Refactored code – move method.

Add Parameter

When a method does not have enough data and requires more data from the caller, the add parameter refactoring technique can be applied. This refactoring technique should be used carefully because its excessive usage can result in the long parameter list bad smell.

Refactoring Procedure

- If the method is defined in superclass or subclass, repeat all the steps.
- Declare a new method with a meaningful name and copy all the steps from the old method and add the required parameter. Replace the body of the old method with a call to the new method.
- Run tests.
- Obtain all the references to the old method and replace them with the references to the new method.
- Delete the old method.
- Compile and test.

Example

Figure 5 illustrates the usage of the add parameter refactoring technique.

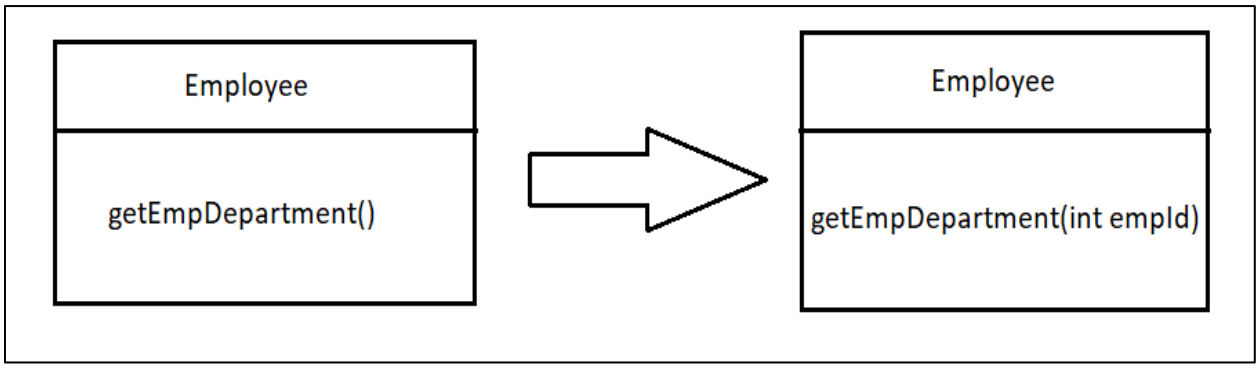

FIGURE 5. Add parameter smell and refactoring.

Parameterize Method

In many cases, multiple methods perform identical tasks with different values. Such code constitutes duplicity and inflexibility. It can be cured by using the parameterize method refactoring technique. A new method is created with additional parameters to address the different values, and the logic of the old methods is added to it.

Refactoring Procedure

- Create a new method with additional parameters to replace the old methods.
- Replace the code of one of the old methods with the call to the new method.
- Run tests.
- Repeat the step for all the old methods.
- Compile and test.

Example

Figure 6 shows the example of hard code that is refactored using the parameterize method refactoring technique as shown in Figure 7.

```
public double  fivePercentRaise() {
  salary = salary * 1.05;
  return salary;
}

public double tenPercentRaise() {
   salary = salary * 1.10;
   return salary
}
```

FIGURE 6. Bad code – parameterize method.

```
public double raise(float percentage) {
  salary = salary * (1 + percentage);
  return salary;
}
```

FIGURE 7. Refactored code – parameterize method.

Extract Super Class

Another example of duplicate code is two or more classes that contain similar functionalities. Code duplication is one of the major problems in programming. It can be solved by creating a super class that contains the common functionalities of these classes.

Refactoring Procedure

- Create a new super class and make the old classes sub-classes of this super class.
- Run tests.
- Use the pull up methods, the pull up constructor-body and the pull up fields refactoring techniques to move the common functionalities of old classes to the super class.
- Compile and test.

Encapsulate Field

One of the four pillars of object-oriented programming is encapsulation. Encapsulation hides the data from the outside environment. When the inner data of the class can be accessed directly by other classes, risks of occurrence of discrepancies in the values and behavior of the data increases. This decreases the reliability of the code. The encapsulate field refactoring technique converts the data of the class to private and adds a getter and setter to access the data.

Refactoring Procedure

- Create getters and setters for the fields in the class.
- Replace all the references to these fields with the call to getters and setters.

- Run tests.
- Change the access modifiers of the fields to private.
- Compile and test.

Example

Figure 8 shows the code that contains a field accessible to the outer classes directly. Figure 9 illustrates the usage of the encapsulate field refactoring technique in order to prevent outside entities from modifying the field directly.

```
public class Employee {

   public String emp_name;

}
```

FIGURE 8. Bad code – encapsulate field.

```
public class Employee {

   private String emp_name;

   public void setEmpName(String name) {
      this.emp_name = name;
   }

   public String getEmpName() {
      return this.emp_name;
   }

}
```

FIGURE 9. Refactored code - encapsulate field.

Extract Method

When a block of code performs a single function and can be placed together, then it can be converted to a method. This increases the modularity and reusability of the code and decrease code duplication. When there are too many lines of code in the method, then the extract method refactoring can be applied to the chunk of code that can be grouped.

Refactoring Procedure

- Create a new method and give it a self-explanatory name. Replicate the extracted code to the new method.
- Replace the extracted code with the call to the new method.
- Run tests.
- Examine the extracted code. If it contains any temporary variable that is used inside the method only, then declare those fields in the new method.
- Run tests.
- If the variables are used outside the extracted code, then pass those fields in the return statement.
- Compile and test.

Example

Figure 10 gives the example of a code that performs multiple tasks. Figure 11 shows the code that is refactored by applying the extract method refactoring technique.

Remove Middle Man

If a class contains many methods that do not do anything except assign the task to other objects, then such methods should be removed, and the methods that perform the tasks should be called directly.

```
public int empCompensation(int empId, String empName, int experience) {
    int bonus;
    System.out.println("***************** Employee Details *****************");
    System.out.println("Employee Name-" + empName);
    System.out.println("Employee Id-" + empId);

    if(experience>10) {
        bonus = 5000;
    }
    else {
        bonus = 10000;
    }
    return bonus;
}
```

FIGURE 10. Bad code – extract method.

```
public int empCompensation(int empId) {
   int bonus;
   printEmpDetails();

   if(experience > 10) {
      bonus = 5000;
   }
   else {
      bonus = 1000;
   }
   return bonus;
}

public void printEmpDetails() {
   System.out.println("*****************Employee Details *************");
   System.out.println("Employe Name - " + empName);
   System.out.println("Employe Id - " + empId);
}
```

FIGURE 11. Refactored code – extract method.

Refactoring Procedure

- Replace all the calls to the intermediate method with the calls to the method that performs all the tasks.

- Run tests.
- Delete the intermediate method.
- Compile and test.

Example

Figure 12 shows the example of a code in which a method calls another method to perform the function. It is refactored by using the remove middle man refactoring technique as shown in Figure 13.

```
public class Person {

    Departmnt _department;

    public Person getManager() {
        return _department.getManager();
    }
}

class Departmnt {
    private Person _manager;
    public Departmnt (Person manager) {
        _manager = manager;
    }

}
```

FIGURE 12. Bad code – remove middle man.

```
class Person {
    public Department getDepartment() {
        return _department;
    }
}
```

FIGURE 13. Refactored code – remove middle man.

Replace Temp with Query

Temporary variables are created to store the results of the expression or some other values that are to be used later in the code. Such temporary variables increase lines of code.

Those expressions can be extracted into a method, and all the references can be replaced by the temporary variables using the method call. This method can also be reused by other classes and methods.

Refactoring Procedure

- Examine the code and verify that the temporary variable is assigned the value only once.
- Create a new method and move the right-hand side of the temporary variable to the body of the method.
- Replace all the references of the temporary variable with the call to the new method.
- Compile and test.

Example

Figure 14 illustrates the example of a bad code that uses temporary field *basePrice.* It can be refactored by using the replace temp with query refactoring technique as shown in Figure 15.

```
double basePrice = _quantity * _itemPrice;
if (basePrice > 1000)
    return basePrice * 0.95;
else
    return basePrice * 0.98;
```

FIGURE 14. Bad code – replace temp with query.

Introduce Parameter Object

If there are multiple parameters in the code that can be grouped, replace those parameters with an object. As a result, all the similar data is clumped together. It reduces code duplication and increases code readability.

```
if (basePrice() > 1000)
  return basePrice() * 0.95;
else
  return basePrice() * 0.98;


  double basePrice() {
      return _quantity * _itemPrice;
  }
```

FIGURE 15. Refactored code – replace temp with query.

Refactoring Procedure

- Create a new class to store the group of parameters.
- Add all the parameters to the class.
- Create an object of the new class and replace all the references to the parameters in the data clump with the object.
- Run tests.
- Remove the old parameters.
- Compile and test.

Extract Class

When a class is too long, it is difficult to know the intention of the class. Initially, the classes are created with clear motivation, and they contain very specific and less amount of data. With the increase in requirements, more methods and parameters are added to them, which results in a very long class. Such classes are a bad sign of programming. To solve this issue, determine blocks of code that can be extracted to a new class. Move the code to a new class and then replace that code in old class with the object reference.

Refactoring Procedure

- Create a new class.
- Create a link from old class to the new class.
- Determine the blocks of code that can be extracted from the old class.
- With the help of the move method and the move parameters refactoring techniques, move the pre-decided parameters and methods to the new class.
- Compile and test after each step.

Decompose Conditional

Complex conditional logic in the code is hard to understand. Writing a code in order to fulfill different conditions increases the lines of code and may result in a long method or a long class bad smell. A code can be made more readable by extracting the conditions and the logic inside the conditions into a new method. Replacing the conditional logic in the code with the method call renders the code cleaner.

Refactoring Procedure

- Copy the conditional logic of code into the new method with the help of the extract method refactoring technique.
- Repeat the above step for then and else block.
- Replace the conditional logic of the code with the call to the new methods.
- Compile and test.

Example

Using complex conditional logic is bad for code readability. Figure 16 shows the example of a bad code containing conditional logic. As shown in Figure 17, the decompose conditional refactoring technique can be used to remove complex conditional logic.

```
if (date.before (SUMMER_START) || date.after(SUMMER_END))
    charge = quantity * _winterRate + _winterServiceCharge;
else
    charge = quantity * _summerRate;
```

FIGURE 16. Bad code – decompose conditional.

```
if (notSummer(date))
    charge = winterCharge(quantity);
else
    charge = summerCharge (quantity);
```

FIGURE 17. Refactored code – decompose conditional.

CHAPTER 5

ABSTRACT SYNTAX TREE

Abstract syntax tree (AST) is the abstract representation of the source code in the form of a tree. The tree representation of the source code is more suitable to examine and alter the code programmatically, as compared to other representations of the source code. Every node of the tree represents a single statement in the source code. The structure of the tree does not represent all the details of the source code, as ASTs are abstract. For example, the start and end parenthesis are not represented in the tree structure, and a statement such as an if-else condition is represented in the abstract syntax tree with three nodes. As a result, AST is different from the other parse trees that are generally built by the parser during compilation of code.

Some properties of the abstract syntax tree make it useful for the analysis of source code and its modification:

- An AST can be altered and upgraded using data. The properties and the information of each component of an AST can be modified easily. Due to various dependencies, such alteration is difficult with the source code directly.
- In contrast with the source code, an AST does not contain any unnecessary braces, semicolons, and other symbols.
- An AST contains additional data about the program because of the continuous phase of compiler analysis. For instance, it stores the location of every component in the source code, enabling the compiler to print helpful error and warning messages.
- AST is less complex and contains fewer elements as compared to the parse tree.

The tool discussed in this thesis uses AST to analyze the source code for bad smell detection. In the initial phase of code smell detection, all the data of the source code, such as the

list of classes, methods, and variables, are collected and used in the later phase to detect the bad smells.

AST Design

The design of the AST is closely related to the design of a compiler. There are a few major requirements to design AST [28]:

- The type of variables should be correctly represented, and the position of every variable declaration should be preserved in the AST.
- The order of the statements to be executed should be well-preserved and defined correctly in the AST.
- Left and right components of binary or ternary operations must be saved and recognized correctly.
- Variables and their values must be saved for assignment statements.
- Unparsing of the AST should result in the original code in terms of appearance and execution. Unparsing is a term in compiler theory. Parsing is the process of converting the source code into an AST and unparsing is the process of converting the AST to the source code.

Figure 18 shows the general flow of abstract syntax tree. The source code is provided as an input in the form of a java file. A character array containing the code can also be passed as an input. In the next step, the source code is parsed by ASTParser. ASTParser takes java source code as an input and parses it into an AST. In the last step, the AST is modified, and the changes are written back with the help of IDocument, which is a java model element that can be used as a wrapper for the source code.

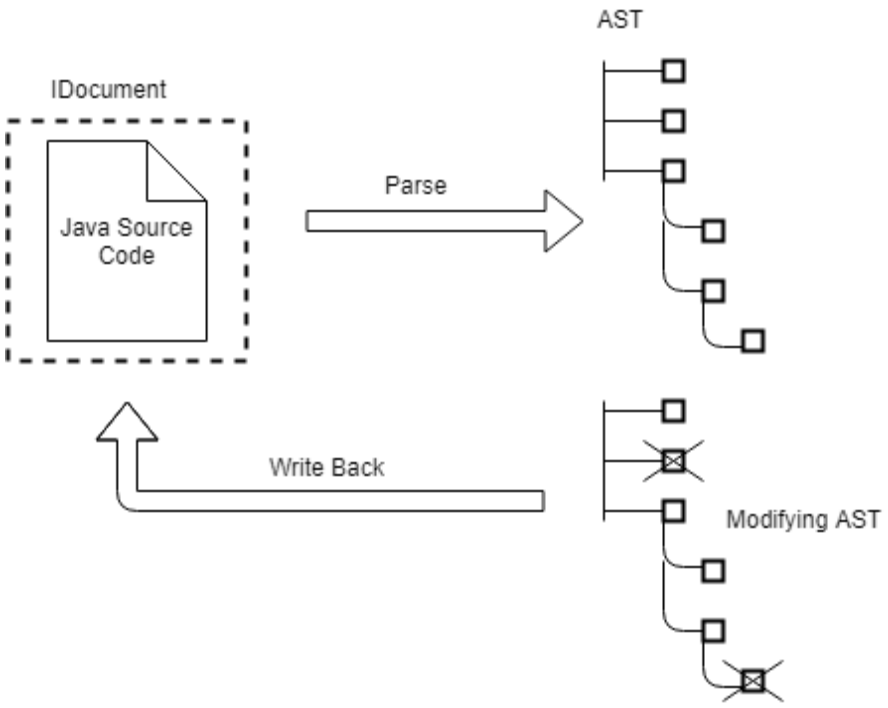

FIGURE 18. Flow of abstract syntax tree.

Figure 19 shows a simple java code of a method that takes two parameters as an argument and returns an integer value, which is the addition of the two parameters. Figure 20 depicts the AST representation of the code shown in Figure 19. Figure 21 contains the Eclipse AST view of the same code. This hierarchy can be easily viewed in Eclipse IDE. Generally, the AST is created by parsing java source code instead of generating it from square one. ASTParser processes java files and creates an AST. As shown in Figure 22, the parse() method parses the source code stored in a file with filePath mentioned in fileName variable.

```
public class Test {
        public int add(int a, int b){
                int c = a+b;
                return c;
        }
}
```

FIGURE 19. Code snippet example.

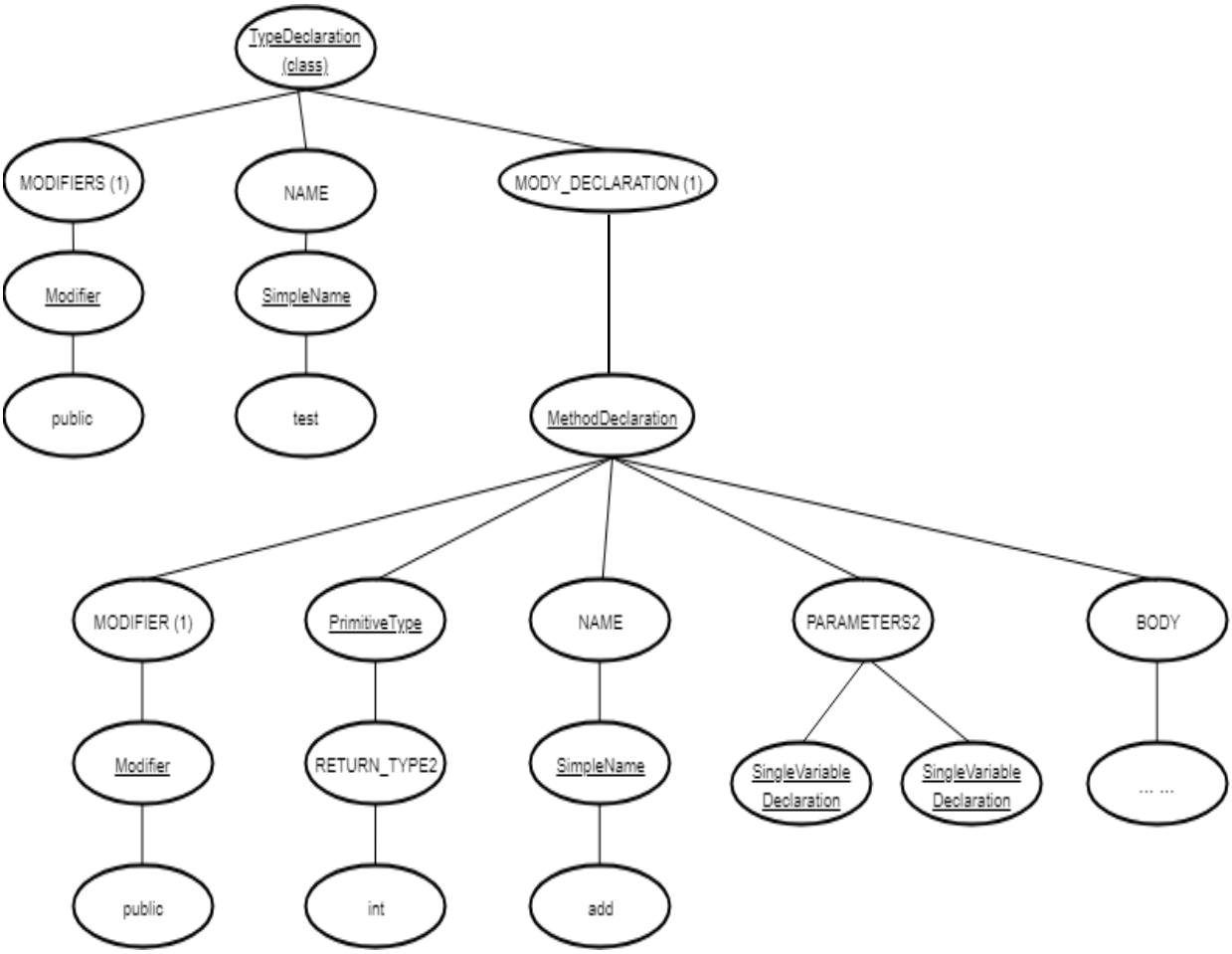

FIGURE 20. Abstract syntax tree.

In Figure 22, ASTParser.newParser(AST.JSL3) instructs the parser to process the code that follows Java Language Specification, third edition (JSL3). All the new language specifications launched in Java 5 are included in JSL3. The next step is to tell the parser that it should expect the input to be in the form of a character array. It is done by parser.setSource(get Chars(fileName), which converts the file stored at the location fileName into a character array and sets the source of the parser. In the last step, the AST is generated, and it is returned in the form of a compilation unit. A compilation unit consists of a single java class definition.

In this thesis, the source code, which contains bad smells, is passed as input, and an AST is generated. This tree is then analyzed for bad smell detection.

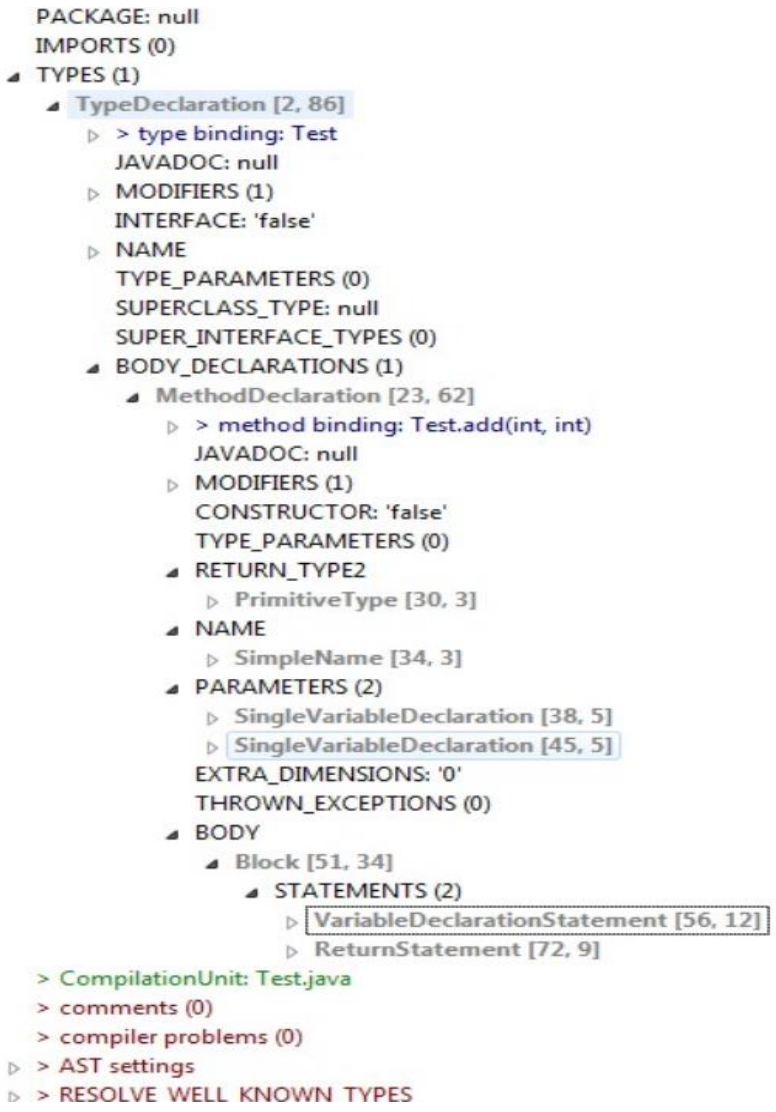

FIGURE 21. Eclipse ASTView.

```
private CompilationUnit parse(String fileName) {
    ASTParser parser = ASTParser.newParser(AST.JLS3);
    parser.setSource(getChars(fileName));
    CompilationUnit cu = (CompilationUnit) parser.createAST(null);
    return cu;
 }
```

FIGURE 22. Parse() – method to convert source code into AST.

CHAPTER 6

CODE SMELL DETECTION

The process of code smell detection is extremely intricate. A detailed and vigilant analysis of the system is needed for the effective detection of bad smells, and it is easy to detect smells in a small system. As the system grows, the risks of breaking the code and making the design worse increase if an engineer is not careful; therefore, the necessity for the automatic detection of code smells increases. Code smells are identifiable in source code when one or more smell features are satisfied [4]. When all the features of a specific smell are found using the static analysis of the code, then that code smell can be identified. However, with the help of data collected from AST, detection of every code smell in the code is impractical [28].

In the tool developed in this thesis, Java source code is provided as an input. The tool analyzes the source to detect the smells. There are seven smells detected in this approach: Middle man, Data class, Temporary field, Long method, Switch case, Message chains, and Long parameter list. The detection and refactoring of all these smells is discussed in detail in the later section of this chapter.

Architecture

A general workflow of the approach used in this thesis for the detection and refactoring of the smell is shown in Figure 23. Java source code is provided as an input, which is parsed by ASTParser to generate the AST. The process of code smell detection is carried out in three phases. In the first phase, the ASTParser analyzes the source code and converts it into an AST. In the second phase, the analyzer traverses the AST and gathers the data. This data is compared with the existing aspects of bad smells to check whether that smell has occurred. In the last phase, the bad smell is detected as well as visualized in the user interface

(UI) and refactoring suggestions are provided for the detected smells. The primary UI of the tool is shown in Figure 24.

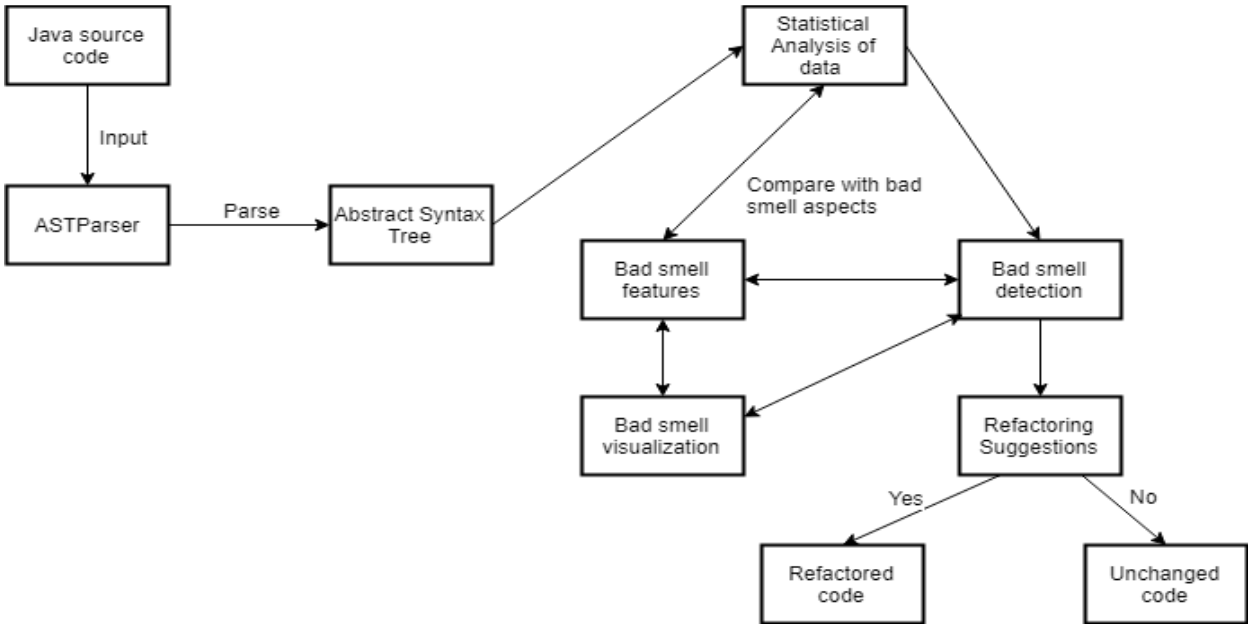

FIGURE 23. Architecture.

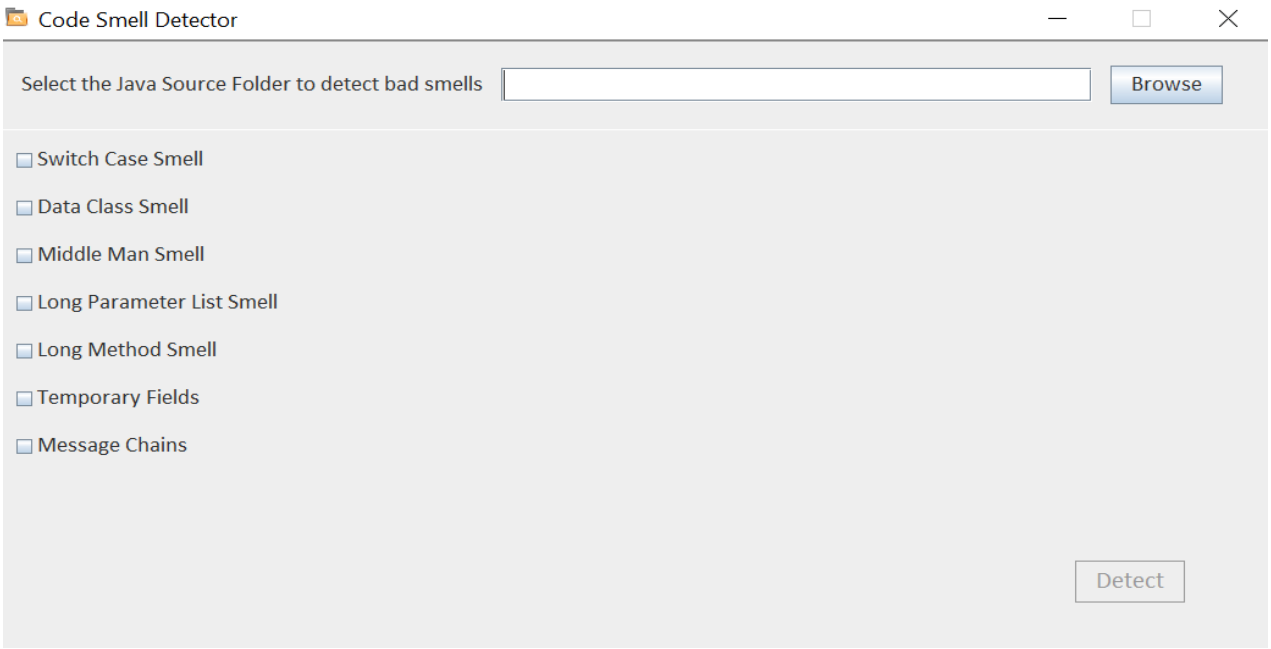

FIGURE 24. Main user interface.

Middle Man

Problem

The middle man bad smell occurs when a method or a class passes most of its requests to another class or method. A middle man is a class or a method that is not effectively contributing to the system.

Objective

Identify the methods that contain the middle man smell and suggest refactoring techniques for the same.

Solution

- Traverse all the method declaration nodes in the AST generated for the source code. Method declaration nodes in the AST contain the method definition.
- Visit block node to get the method body.
- Verify whether the statement in the method body is a return statement by traversing through the return statement node.
- If it is a return statement node, check whether the body of the return statement contains a method call by traversing through the method invocation node.
- Check if the node name of the method invocation node is present in the initially compiled list of methods.
- If the control returns true, then the middle man smell is present in the code.

Refactoring

Apply the remove middle man refactoring technique to remove the middle man bad smell. This refactoring technique is discussed in detail in chapter 4. Figure 25 shows the UI for the middle man code smell.

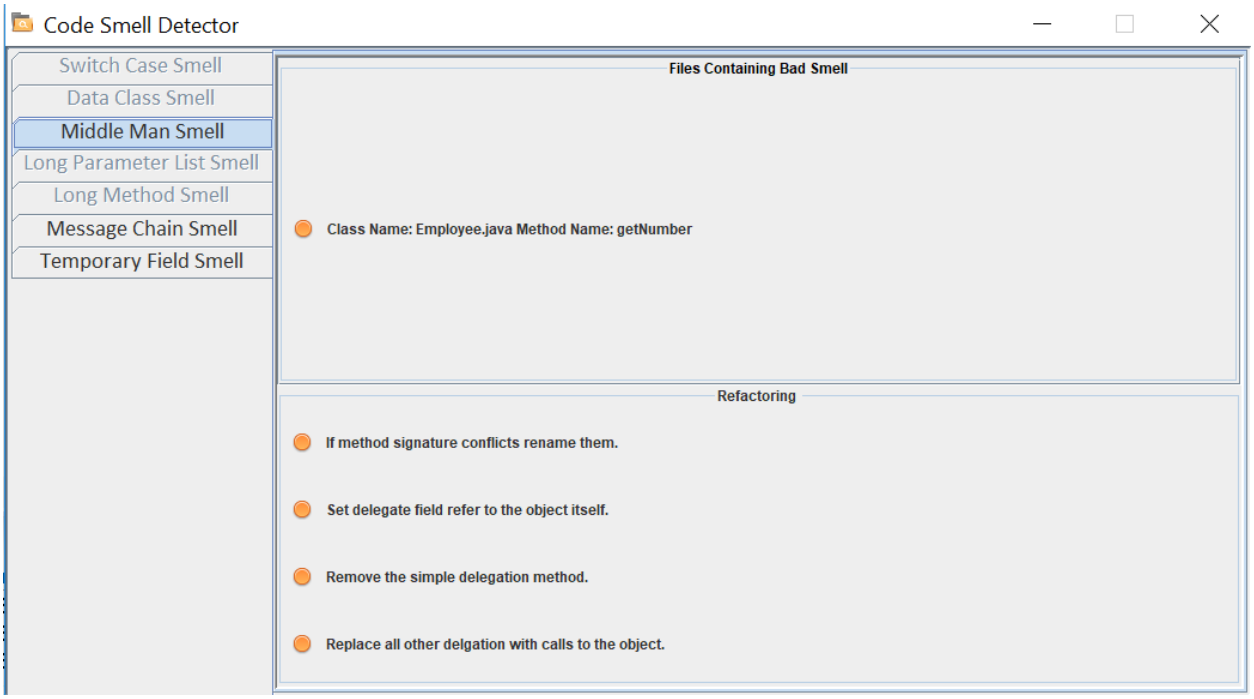

FIGURE 25. Middle man smell.

Data Class

A class that contains only data fields and methods like getters and setters for accessing those data fields is a data class. Data class is generally a repository for data that is utilized by another class. These classes do not have any auxiliary usefulness, and they cannot function on their own. It causes an increase in the coupling between the classes as the data class is referenced every time when its fields are used by another class.

Objective

Check if the class contains the data class bad smell; if it contains the smell, then suggest refactoring solutions.

Solution

- Traverse through the method declaration node in the AST.
- Block node in the tree contains the method body. Visit the block node. Calculate the

number of lines of the method.

- Traverse through the return statement node to verify its presence. If it is present, subtract one from the previously calculated number of lines.
- To count the number of variables declared, visit the variable declaration node. Subtract the number of variable declarations from the number of lines.
- Traverse through the expression node. Visit the assignment node. Calculate the total number of visited assignment nodes and subtract it from the number of lines.
- If the number of lines is greater than zero, then the data class smell exists.

Refactoring

Use the move method refactoring technique to move the getter and setter methods to the class, which calls these methods. If the whole method cannot be moved to another class, use the extract method to extract the getter and setter part of the method and then move the extracted method to another class that calls it.

Figure 26 shows the UI for the data class bad smell.

Temporary Field

The temporary field bad smell occurs when certain variables receive their value only under particular circumstances. This occurs when these variables are used only in a method, and the values are received by these variables only when a method is called. In all the other cases, these variables remain empty.

Objective

Detect if the class contains the temporary field bad smell. Suggest refactoring solutions for it.

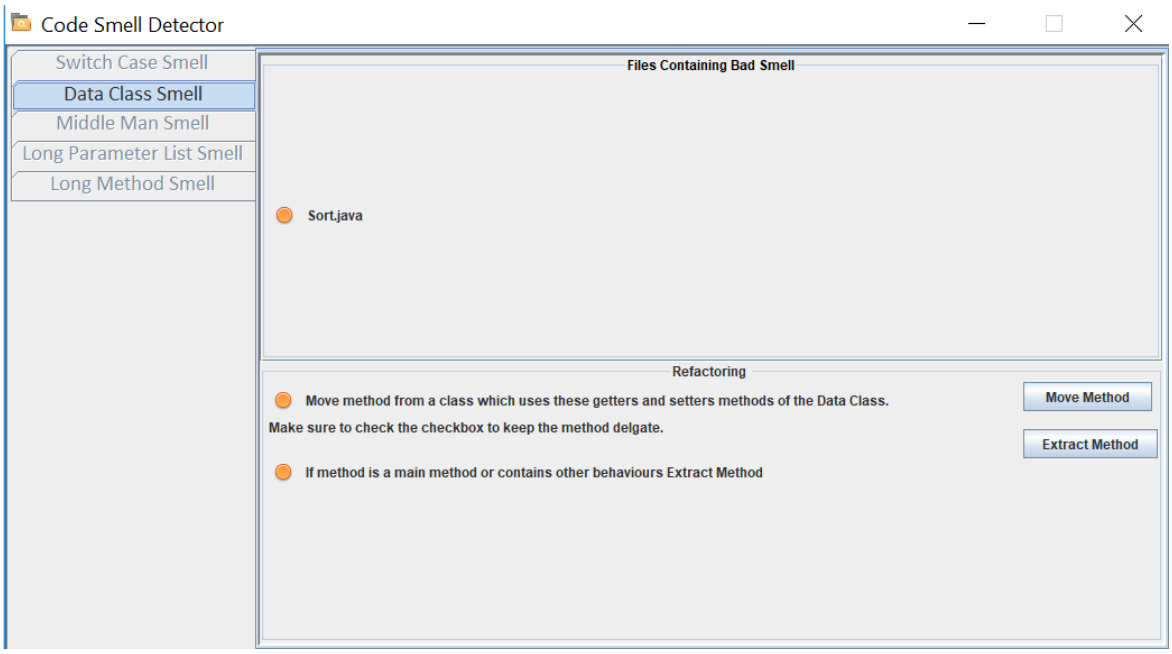

FIGURE 26. Data class smell.

Solution

- Traverse through the method declaration node in the AST.
- Access the body of the method by visiting the block node.
- Visit the assignment node by traversing through the expression node.
- Check if the right hand of the assignment contains a method call by visiting the method invocation node. Increase the counter by one if there is a method call on the right-hand side of the expression.
- Perform above step for every assignment node visited.
- If the counter is greater than two, then the temporary field bad smell is present.

Refactoring

Use the extract method refactoring to extract the part of the code that contains the temporary fields smell. If the temporary fields are used in conditions, then use the introduce null object refactoring to remove the temporary fields and directly call the method.

Long Method

A method, procedure, or function that contains too many lines of code is considered to be a long method. When a method is too long, it uses a large number of variables and performs multiple operations, and it is plausible that the method performs more functions than those suggested by its name suggests. Additionally, when a method is small in size, then understandability and maintainability of the code increase.

Objective

Check if the method is too long and contains the long method bad smell. Suggest refactoring techniques to remove the smell.

Solution

- Traverse through the method declaration node in the AST.
- In order to access the body of the method, visit the block node. Count the number of lines in the method.
- If the method contains while loop or for loop, repeat the above step.
- If the number of lines is more than fourteen, then the long method bad smell is present.

Refactoring

Use the extract method refactoring technique to decrease the length of the method. If the method uses local parameters, use the replace temp with query refactoring technique to move all the parameters along with the method.

Figure 27 shows the UI for the long method bad smell.

Switch Case

A good case of object-oriented programming indicates less usage of switch cases. Generally, there are many switch cases across the code when a single case is added to satisfy all

conditions. When a new condition is added, all the switch cases across the code need modification.

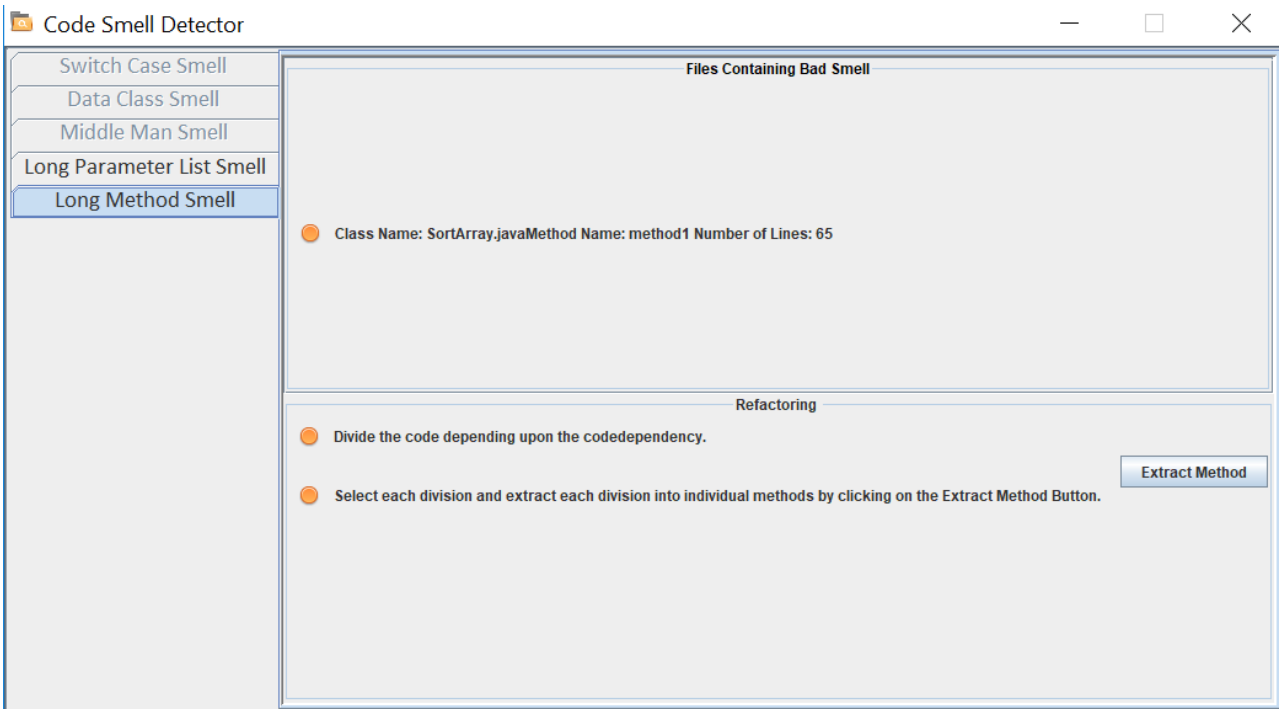

FIGURE 27. Long method smell.

Objective

Check if there are any switch statements in the method, then suggest refactoring techniques if a switch case smell is detected. Figure 28 represents the UI for the switch case bad smell.

Solution

- Traverse through the switch statement node in the method in the AST. Calculate the number of times the switch statement node is visited.
- For every switch statement node visited, traverse through the simple name node to access the name of the switch statement. Store the name.

- If switch statement node is visited more than three times, then the switch case smell is present in the code.

Refactoring

In order to remove dependencies from the switch case block, use the extract method refactoring technique and then the move method refactoring to move the code to the right class.

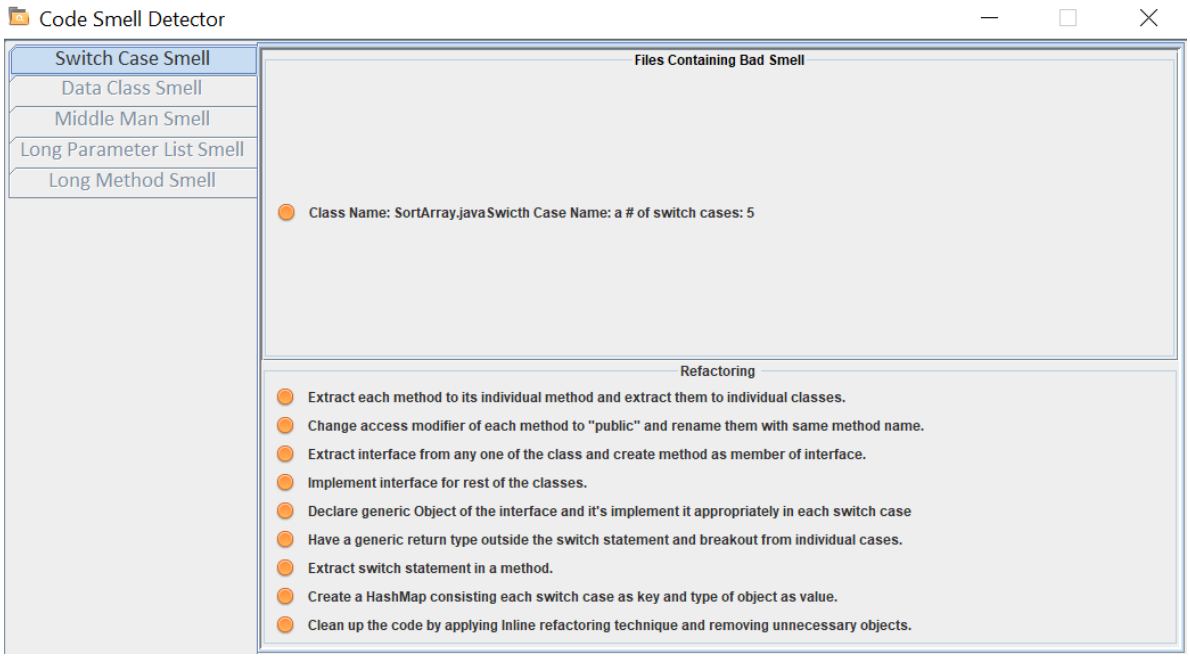

FIGURE 28. Switch case smell.

Message Chains

When a class has a high dependency on other classes and when these classes are dependent on other classes, a dependency chain occurs between the classes. Thus, when one class is modified, the dependent classes also need modification. If the dependent classes are not changed appropriately, errors occur in the code. When a class has a high coupling with other classes in the form of a chain, the message chain bad smell occurs. A message chain occurs when an object of one class calls an object of another class that, in turn, calls an object of another class

and so on. Due to the high coupling between the classes, where there are any changes made to the intermediate classes, the main class also requires changes to be made in it in order to maintain consistency throughout the code.

Objective

Analyze the code to check whether there is a message chain bad smell in the code. If the smell is present, provide refactoring suggestions.

Solution

- Traverse all the method declaration nodes in AST that are generated for the source code. Method declaration nodes in AST contain the method definition.
- Visit the block node to obtain the method body.
- Verify if the statement in the method body is a return statement by traversing through the return statement node.
- If it is a return statement node, check whether the body of the return statement contains a method call by traversing through the method invocation node.
- Check if the node name of the method invocation node is present in the initially compiled list of methods.
- If the control returns true, increase the counter by one.
- If the value of the counter is greater than 3, the message chain code smell is present.

Refactoring

Use the hide delegates refactoring technique to delete the delegating part of the code and directly call the end method. The other option is to use the extract method refactoring technique to extract the end method in the chain and put the extracted code in the main method to remove the chain.

Long Parameter List

When more than three or four parameters are passed as an argument to a method, the long parameter list bad smell occurs. When there is a need for additional data in the method, an object can be used instead of passing that data as an argument. A long list of parameters is hard to understand, and it becomes harder to maintain it as it increases in size.

Objective

Detect the long parameter list smell in a method and suggest refactoring techniques.

Solution

- Traverse through the method declaration block of the AST.
- Traverse through the single variable declaration node from the method declaration node. Single variable declaration node points to the parameters in the method definition.
- Count the number of times a single variable declaration node is visited.
- If the number of parameters is greater than four, the method has the long parameter list bad smell.

Refactoring

If the parameters can be substituted by a method call, use the replace parameter with the method refactoring technique.

Figure 29 shows the UI for the long parameter list bad smell.

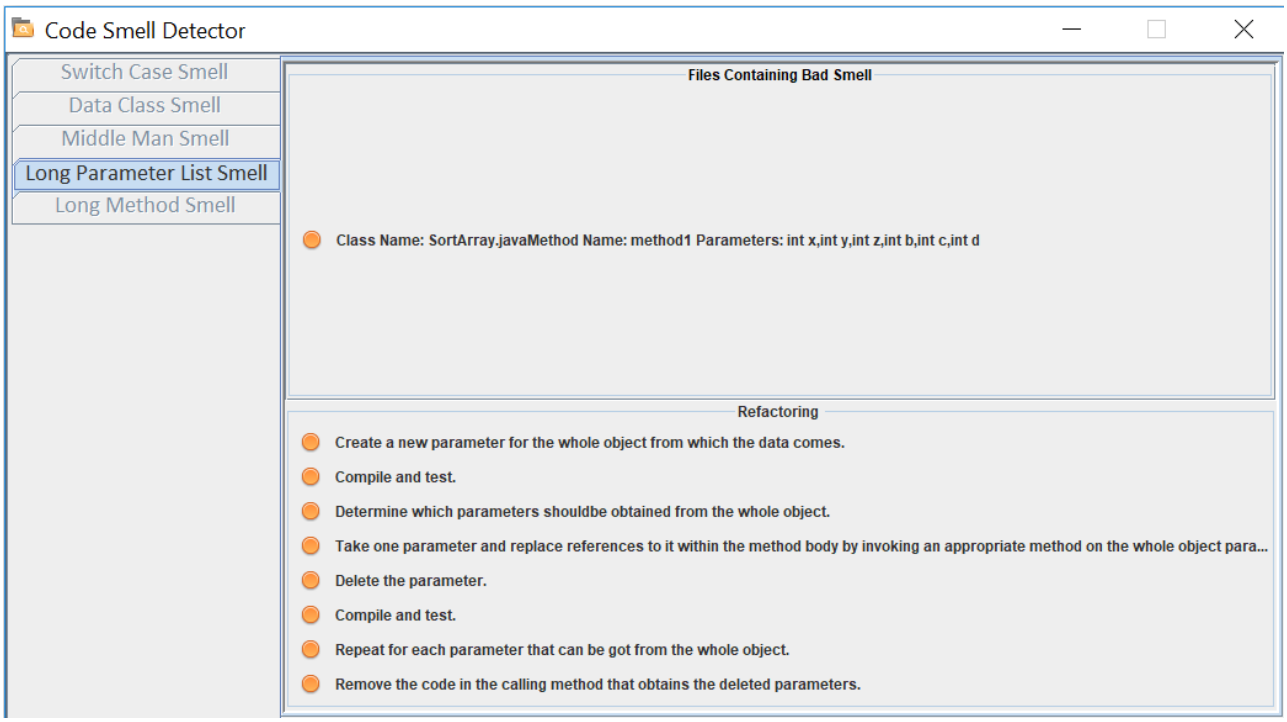

FIGURE 29. Long parameter list bad smell.

CHAPTER 7

CONCLUSION AND FUTURE WORK

Conclusion

Past research works have presented numerous methodologies to refactor the code in order to make it more readable. However, only a small amount of research has specified the ideas of consolidating code smells detection with refactoring, and none have provided a ready-to-use implementation. Most engineers depend only on the different metrics to recognize code smells.

In this thesis, it has been demonstrated that the analysis of the data collected from AST, along with metrics, helps to detect the code smells with greater accuracy. A framework is proposed and built that identifies seven different bad smells and suggests refactoring solutions for the same. As a conclusion, this thesis has contributed three results:

- A methodology to gather data by utilizing an AST for statistical analysis.
- Algorithms to recognize each code smell discussed in this thesis.
- Connections between the code smells and the refactoring techniques.

The methodology used in this thesis for code smell detection and refactoring is more consistent as compared to the other methods discussed in related works, as the AST of the source code provides a clearer view of the framework, and it is easier to change the code without changing its behavior while refactoring the code.

Future Work

All the criteria included in the objective of this thesis were met. However, there are few issues that are still open for future research:

- There are seven bad smells detected in this thesis, and analysis can be carried out for the remaining bad smells that are not discussed in this thesis.

- Currently, the refactoring solutions provided in this approach are semi-manual. It can be possible to make refactoring fully automatized for all the identified smells.
- Further, the graphical UI can be made more interactive and user-friendly.
- This thesis represents a framework that detects bad smells in Java. In future, the algorithms and techniques used in this thesis can be extended to develop a framework that detects bad smells in other object-oriented languages such as C++, Ruby, Python, etc.

REFERENCES

REFERENCES

[1] G. Elliott, "Information Systems Methodologies," *Global Business Information Technology: An Integrated Systems Approach,* Pearson Edison Wesley, 2004, pp. 87-98.

[2] M. Fowler, "Technical Debt," Oct. 2003; https://martinfowler.com/bliki/TechnicalDebt.html.

[3] Y. Asiedu and P. Gu, "Product Lifecycle Cost Analysis: State of the Art Review," *Intl. Journal of Production Research,* vol. 36, no. 4, Nov. 2010, pp. 883-908.

[4] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, "Principles in Refactoring," *Refactoring: Improving the Design of Existing Code,* Addison Wesley Longman, 1999, pp. 13-45.

[5] S. M. Olbrich, D. S. Cruzes, and D. I. K. Sjoberg, "Are all Code Smells Harmful? A Study of God Classes and Brain Classes in the Evolution of Three Open Source Systems," presented at the IEEE Intl. Conf., Sep. 2010, DOI: 10.1109/ICSM.2010.5609564

[6] M. Mantyla, J. Vanhanen, and C. Lassenius, "A Taxonomy and an Initial Empirical Study of Bad Smells in Code," *Proc. Intl. Conf. on Software Maintenance,* 2003, pp. 381-384.

[7] M. L. Maher, J. Gero, and M. Saad, "Synchronous Support and Emergence in Collaborative CAAD," *Proc. Intl. Conf. on Computer-Aided Architectural Design Futures,* 1993, pp. 455-470.

[8] U. Tekin, U. Erdemir, and F. Buzluca, "Mining Object-Oriented Design Models for Detecting Identical Design Structures," *Proc. Intl. Workshop of Software Clones,* 2012, pp. 43-49.

[9] Agile Alliance, "What is an Anti-Pattern? Agile Alliance," 2018; https://www.Agilealliance.org/glossary/antipattern.

[10] G. Suryanarayana, G. Samarthyam, and T. Sharma, "Design Smells," *Refactoring for Software Design Smells: Managing Technical Debt,* Morgan Kaufmann Publishers, 2014, pp. 9-15.

[11] E. Van Emdem and L. Moonen, "Java Quality Assurance by Detecting Code Smells," *Proc. 9th Working Conf. on Reverse Engineering,* 2002, pp. 97-102.

[12] M. Mantyla and C. Lassenius, "Subjective Evaluation of Software Evolvability Using Code Smells: An Empirical Study," *Journal of Empirical Software Engineering,* vol. 11, no. 3, 2006, pp. 395-431.

[13] R. Peters and A. Ziadman, "Evaluating the Lifespan of Code Smells Using Software Repository Mining," presented at the 16th European Conf. on Software Maintenance and Reengineering, 2012, DOI: 10.1109/CSMR.2012.79.

[14] T. Sharma and D. Spinellis, "A Survey on Software Smells," *Journal of Systems and Softwares,* vol. 138, no. 8, Apr. 2018, pp. 158-173.

[15] J. Noble, "Classifying Relationships Between Object-Oriented Design Patterns," *Proc. Australian Software Engineering Conf.,* 1998, pp. 98-109.

[16] G. Travassos, F. Shull, M. Fredericks, and V. Basili, "Detecting Defects in Object-Oriented Designs: Using Reading Techniques to Increase Software Quality," *Proc. 14th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications,* 1999, pp. 47-56.

[17] G. Langelier, H. Sahraoui, and P. Poulin, "Visualization-Based Analysis of Quality for Large Scale Software Systems," *Proc. 20th IEEE/ACM Intl. Conf. on Automated Software Engineering*, 2005, pp. 214-223.

[18] M. Lakshmanan and S. Manikandan, "Multi-Step Automated Refactoring for Code Smells," *Intl. Journal of Research in Engineering and Technology*, vol. 3, no. 3, Mar. 2014, pp. 278-282.

[19] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Identification and Application of Extract Class Refactorings in Object-Oriented Systems," *Journal of Systems and Software,* vol. 85, no. 10, Oct. 2012, pp. 2241-2260.

[20] D. Orchard and A. Rice, "Upgrading Fortran Source Code Using Automatic Refactoring," *Proc. ACM Workshop on Refactoring Tools,* 2013, pp. 29-32.

[21] R. Dijkman, B. Gfeller, J. Kuster, and H. Volzer, "Identifying Refactoring Opportunities in Process Model Repositories," *Journal of Information and Software Technology,* vol. 53, no. 9, Sep. 2011, pp. 937-948.

[22] K. Elish and M. Alshayeb, "Investigating the Effects of Refactoring on Software Testing Efforts," presented at the Software Engineering Conf., Dec. 2009, DOI: 10.1109/APSEC.200 9.14.

[23] B. Huston, "The Effects of Design Pattern Application on Software Metrics Score," *Journal of Systems and Software*, vol. 58, no. 3, Sep. 2001, pp. 261-269.

[24] M. Harman and L. Tratt, "Pareto Optimal Search Based Refactoring at the Design Level," *Proc. 9th Annual Conf. Genetic and Evolutionary Computation*, 2007, pp. 1106-1113.

[25] P. Krutchen, R. Nord, and I. Ozkaya, "Technical Debt: From Metaphor to Theory and Practice," *IEEE Software,* vol. 29, no. 6, Dec. 2012, pp. 18-21.

[26] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Penta, A. Lucia, and D. Poshyvanyk, "When and Why Your Code Starts to Smell Bad," *Proc. 37th Intl. Conf. Software Engineering*, 2015, pp. 403-414.

[27] A. Yamashita and L. Moonen, "Do Developers Care About Code Smells? An Exploratory Survey," presented at the 20th Working Conference on Reverse Engineering, Oct. 2013, DOI: 10.1109/WCRE.2013.6671299.

[28] Eclipse, "Abstract Syntax Tree," Nov. 2006; https://www.eclipse.org/articles/ArticleJavaCodeManipulation_AST/index.html.