San Jose State University
SJSU ScholarWorks

Master's Projects | Master's Theses and Graduate Research

Spring 2011

Java Smell Detector

Nitin Mathur
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_projects

 Part of the Other Computer Sciences Commons

Recommended Citation

Mathur, Nitin, "Java Smell Detector" (2011). *Master's Projects*. 173.
DOI: https://doi.org/10.31979/etd.rhbq-w72d
https://scholarworks.sjsu.edu/etd_projects/173

This Master's Project is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Projects by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

JAVA SMELL DETECTOR

A Writing Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Masters of Science

By

Nitin Mathur

Spring 2011

© 2011

Nitin Mathur

All Rights Reserved

SAN JOSÉ STATE UNIVERSITY

The Undersigned Project Committee Approves the Project Titled

Java Smell Detector (JSD)

by

Nitin Mathur

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

Dr. Robert Chun	Department of Computer Science	Date
Dr. Mark Stamp	Department of Computer Science	Date
Mr. Naveen Roperia	Cornerstone OnDemand, Inc.	Date

APPROVED FOR THE UNIVERSITY

Associate Dean	Office of Graduate Studies and Research	Date

Abstract

"Code Smell" or "Bad Smell", at the very least, is an indicator of badly written code and is often indicative of deeper problems in software design. In layman terms, it signals flaws in the core foundation or architecture of the software that can cause any number of more serious problems – from usability and runtime performance to supportability and enhancement. These problems can mostly be prevented by the systematic refactoring of the code. Refactoring is the process (and according to some, an 'art') of making incremental changes to existing source code to improve its nonfunctional attributes, without modifying its external functional behavior. Code smells are symptoms of deep-rooted problems in design, which, in most common cases, inhibit the understandability of the system for present and future programmers, hence rendering the program un-maintainable. The later these problems are identified, the costlier they are to correct as it is much harder to refactor a system in production and regression. Issues caused by refactoring can spiral out of control in advanced stages of the software development life cycle. So far, identification of these code smells has been thought of as an intuitive art rather than an exact science, as there are very few empirical measures or methodologies for doing so.

In this project, I will examine each of the 22 code smells identified in prior research. I will implement Java Smell Detector (JSD), which will follow a scientific approach to detect five of these 22 code smells. JSD will give suggestions to refactor the code for all five of these smells. Further, the tool will provide an interactive process to refactor two of these cases; while for the rest, it will suggest an ideal refactoring technique that would need to be applied manually. I will be using Java code written by students of San Jose State University (SJSU) as test data for JSD and will compare its output against the code smells identified by the graduate students.

Table of Contents

List of Figures

List of Tables

List of Graphs

1 Introduction

Software undergoes various transformations throughout its Software Design Life Cycle (SDLC). These transformations lead to the deterioration of its quality by the introduction of undesired design flaws in the code. Developers generally do not pay much attention to the overall architecture and system design in the beginning of the SDLC. During later stages of quality analysis (QA) and maintenance, most developers look for quick and easy fixes for defects rather than an overall system design. This can be attributed to inexperience, pressure of deadlines, and management perception of budgets. Effort spent in improving system design can be perceived as an upfront cost with little short-term return when compared to a quick bug fix, even though its long-term payback in terms of reduced maintenance costs can be worthwhile. These problems lead to the introduction of various design flaws which are called code smells. According to Beck, Brant, Fowler, Opdyke, & Roberts (2000), there are 22 code smells in object-oriented source codes. These code smells can be categorized in seven groups (Lassenius, Mantyla, & Vanhanen, 2003).

The term "Code Smell" appears to have been coined by Kent, Beck on WardsWiki in the late 1990s (Roperia, 2009). Let us consider an example to understand the meaning of code smell. An unusual long method in an object-oriented programming language like Java might indicate a "Long Method" smell. It indicates lack of understanding of or disregard for a proper object design and the use of simple procedural programming in an object-oriented language. This is a very common sign of code smell in object-oriented language. In rare cases this could be normal, but most likely such long-winded methods perform too much functionality thus making it difficult to understand and maintain. To solve this problem, such methods should first be split into smaller methods or functions to promote reusability. As a second step, if an object-oriented language is being used, the object model should be reviewed as well. However, that

does not mean that a long method is always a code smell problem – it is most likely an indicator of a design issue in the software than of a problem.

Refactoring is a solution to the problem which code smell indicates. It is the implementation technique used to apply a better design to an existing, fully or partially functional, software program. It does present a larger upfront cost compared to quick fixes and patchwork; however, its long-term payoff can be significant. When contrasted with a complete software rewrite, refactoring is a cost effective option. Large software usually goes through a long cycle of development and testing. A complete rewrite means another long cycle of development and testing. Refactoring existing code offers a way of improving the software design and removing code smells without actually rewriting the entire code from scratch.

If a method is long, splitting it into smaller methods is actually a refactoring technique that improves the design of the source code, making it easier to understand, enhance, and maintain. To better understand and emphasize the importance of refactoring, we can take "Duplicate Code" smell as an example. Duplicate code is not just a design issue; it also leads to incorrect calculations and redundant code and data. Therefore, to ensure the software works as per specification, it is necessary to refactor duplicate lines of code into smaller, more manageable methods and invoke these methods from the client code. Thus, the refactoring technique improves the structure of the code dramatically while retaining its functionality.

Correct detection of code smells is the main prerequisite to create a refactoring plan. The correct implementation of the plan will improve the quality of code. The refactoring process depends on the smells found in the system, and it directly affects software maintenance cost. There are in all 72 possible refactoring techniques which are classified into seven sections (Lassenius, Mantyla, & Vanhanen, 2003). Depending on the code smell, one or more of these

refactoring technique(s) can be applied. For example, if a class has been identified containing the "Data Class" smell, we can first apply the "Extract Method" technique and then the "Move Method" from the client class to data class if the client class method has multiple functionalities going on; or we can apply just "Move Method" if the client class has an independent method to move.

Human intuition is believed to be the best way for detection of a code smell and there is no exact science behind it (Brant, Beck, Fowler, Opdyke, & Roberts, 2000). Various scientists have tried and come up with a metrics-based approach to measure and detect code smells with varying degrees of success.[1] The advantage of using the metric approach is that it is easily verifiable with human eyes. Currently there are tools which detect code smells, depending on the metrics, and help the programmers to identify the design problems, but there are no statistical analyses and reasons for how and why they came up with these metrics. Moreover, none of the tools have been developed with students as the target audience and the refactoring approach is non-interactive. The tool (Java Smell Detector) proposed as part of this research helps visualize the problem and derive a solution – i.e. proposed refactoring technique(s) – specifically for the type of design defect detected in the system.

[1] N. Tsantalis, A. Chatzigeorgiou, N. Roperia, M. Lungu, G. Ersze, R. Marinescu, P. F. Mihancea and Gendarme Google Group are few of the scientists who gave some statistical details for the detection.

2 Related Research Work

Existing literature provides a number of resources and support in the field of refactoring. This section describes research done in the field of code smells and few of the currently existing software are which helps to detect code smells in object-oriented systems.

2.1 Current Literature

As mentioned earlier, Fowler introduces 22 code smells in his book "Refactoring: Improving the design of existing code" (Fowler et al., 2000). These smells were later classified into seven sub-categories depending upon their similarities in their characteristics by Mantyla, Vanhanen, and Lassenius (2003). Additionally, Munro focuses primarily on the characteristics of code smells (Munro, 2005). In his article, Munro (2005) describes "the agile software development of eXtreme Programming (XP) devised by Beck as an incremental approach to software design." At each build, the new requirements are fulfilled by integrating the solutions with the existing system, and refactoring is implemented in XP to incorporate the new functionality. In another article, Cusumano and Shelby (1995) portray how Microsoft uses 20% of their development resources to re-develop the code base of old products. They also compare Netscape and Microsoft to show how Netscape's inability to refactor hindered the growth of their software, while Microsoft's redesign efforts paid off with Internet Explorer 3.0.

Software Development Life Cycle (SDLC) consists of different stages, one of which is a design stage. Marinescu (2005) defines the detection strategies that relate to the flaws in the design stage. He describes that the detection strategy has the following four stages:

1. Analysis of the problem: An identified problem taken from literature is analyzed to quantify the informal description.
2. Selection of metrics: Selection of metrics that best matches the problem's characteristics is made using the quantitative description from the previous step.
3. Detection of the candidates: The detection strategy is chosen using the identified metrics.
4. Examination of candidates: The detection strategy is identified, and results indicate whether refinements are required (Marinescu, 2005).

These four stages of detection can be applied to any stage of SDLC with minor changes.

In the press article "Microsoft Secrets", Cusumano and Shelby (1995) followed the software metrics approach and detected code smells such as "Lazy Class" and "Temporary Field." He used software metrics including LOC (Lines of Code), NOM (Number of Methods), CBO (Coupling Between Objects), and WMC (Weighted Methods per Class) to detect code smells.

As described by Lassenius, Mantyla, and Vanhanen (2003), "Code smell detection process is based on software quality." Study on the use of these indicators has suggested that code smells are subjectively perceived on the basis of code quality. Further improvement is being performed on their work to enhance detections of code smells at the method level. This research was primarily aimed at evaluating, validating, and improving the understanding of the subjective indicator.

2.2 Tool Support

There are various tools and IDEs available for the code smells detection and refactoring process. Some of them are jDeodorant for Java (Chatzigeorgiou, Fokaefs, & Tsantalis, 2007; Chaikalis, Chatzigeorgiou , & Tsantalis, 2008), JSmell (Roperia, 2009) for C#, csharprefactory for C# and Eclipse (Crespo, Lopez, & Marticorena, 2005) for Java. Some of the commonly used tools and the approach they adopt in order to detect code smell will be discussed in this section.

"Feature Envy" and "Type Checking" are two kinds of code smells identified by JDeodorant (Fokaefs, Tsantalis, & Chatzigeorgiou, 2007). JDeodorant uses the ASTParser API of Eclipse to detect the code smell from the source code. "Feature Envy" smell is detected based on the notion of the distance between entities (methods) and system classes. This code smell is identified if the distance of a method from a system class is less than the distance of this method from the class that it belongs to (Chatzigeorgiou, Fokaefs, & Tsantalis, 2007; Chaikalis, Chatzigeorgiou , & Tsantalis, 2008). The distance violates the principle of high cohesion, which requires a method to be less cohesive to any other class except the class to which it belongs. In "Type Checking" the underlying aim is to implement polymorphism by using refactoring. Identification of code smell involves two cases (Chaikalis, Chatzigeorgiou, & Tsantalis, 2008):

- First case - There could be a field which represents state (if-else-if loop or switch case). Depending on its value, the corresponding conditional branch is executed.
- Second case - There is a conditional statement that employs Run Time Type Identification (RTTI) in order to cast a reference from base type to the actual

derived type and invoke methods of the specific subclass (Chaikalis, Chatzigeorgiou, & Tsantalis, 2008).

Metrics (Crespo, Lopez, & Marticorena, 2005), an Eclipse plug-in, detects "Parallel Inheritance Hierarchy" code smell. Some of the metrics included in this approach are DIT (Depth Inheritance Hierarchy) and NOC (Number of Children). The results are evaluated based on data mining techniques.

Prodeos (Ersze, Lungu, Marinescu, & Mihancea, 2008) detects design flaws in C++ and Java programs. The basic strategy used by this tool for analyzing the code is based on software metric detection. The metrics of the software is analyzed through the statistical data captured while parsing through the program. Applying a detection strategy creates a report containing all the design entities suspected to be affected by the quantified flaws

JSmell (Roperia, 2009) is a smell detector developed in C# language for Java. It detects seven of the code smells: "Data Class," "Message Chain," "Primitive Obsession," "Speculative Generality," "Parallel Inheritance Hierarchy," "Duplicate Code," and "Comments." It uses the ANTLR (ANother Tool for Language Recognition) parser to parse the code file and gathers the statistical results to classify the Smells. JSmell has two phases to identify a smell. During the first phase, it parses all the Java source code files and gathers required data like method declaration, variable declaration, and class names. In the second phase, it uses this statistical data and parses all the code again to identify the smells present in each of them.

Code smells that are constrained to the domain specific language are detected using DÉCOR (Moha, 2007) which uses a design-defect detection algorithm. The smell detection

rules are specified using the structural and lexical properties and their relationship. Detection of the code smell is a three-step process.

- First step - Parse the source code using JFlex and JavaCup.
- Second step – Reification, which is basically a specification of defects based on the meta-model of the target system. This process is followed in order to capture the high level defects, and a repository is maintained.
- Third step - The detection algorithm is generated and implemented as visitors on the meta-model. "The algorithm generation process uses the services of Software Architectural Defects (SAD) framework and is based on the templates which are excerpts of Java source with well-defined tags to be replaced by concrete code." (Moha, 2007)

All the above mentioned tools have a scientific approach to detect code smells. They provide the foundation for using statistical analysis approach. However, the smells detected by all of the above mentioned tools are more susceptible in industrial systems and not in the systems of students who are learning object-oriented language. Moreover, JSmell is the only one which provides suggestions for the refactoring technique and none of them provide an interactive tool for the user to refactor the code. JDeodorant as built in Eclipse can use the inbuilt refactoring techniques, but it does not provide any refactoring suggestions.

3 Code Smells Overview

The detection of code smells in code assists the software architects and developers in identifying the need of software redesign. Software redesign can be implemented by refactoring – as rewriting large software program is almost never cost effective or viable.

3.1 Taxonomy of Code Smells

In Java and related objected-oriented languages code smells can be classified in two ways:

3.1.1 Depending upon the Class:

One way of categorizing code smells is based on whether they are found within the class or outside the class. The smells which are recognized within the class are "Long method," "Long Parameter List," "Comments," and "Duplicate Code." The ones which are recognized outside the class are "Data Class," "Data Clumps," and "Primitive Obsession."

3.1.2 Sub-Categorizing depending upon similarity:

Another way of categorizing the code smells is based upon similarity amongst themselves. The 22 code smells can be sub-categorized into seven different categories based on the listing provided in (Lassenius, Mantyla, & Vanhanen, 2003). These smells are closely related to each other based on the relationship among them. If a smell does not fit in any of the categories, it is put into "Others" category.

Category	Code Smells
Bloaters	• Long Method • Large Class • Long Parameter List • Primitive Obsession • Data Clumps
Object-Oriented Abusers	• Switch Statements • Temporary Field • Refused Bequest • Alternate Class with Different Interface • Parallel Inheritance Hierarchies
Change Preventers	• Divergent Changes • Shotgun Surgery
Dispensable	• Lazy Class • Data Class • Duplicate Code • Speculative Generality
Encapsulators	• Message Chain • Middle Man
Couplers	• Feature Envy • Inappropriate Intimacy
Others	• Incomplete Class Library • Comments

Table 1: Categorization of Code Smells depending on similarity

3.2 Code Smells

It is easier to understand and identify individual code smells if divided into seven sub-categories rather than as a list. In this chapter, we will look into each of the code smells in individual sub-categories to have a better understanding of them.

3.2.1 Bloaters

"Bloaters" represent the set of code which has grown so large that it cannot be handled effectively. This group includes the following code smells:

- ***Long Method:*** A method which contains large number of lines and performs more than one action is considered as "Long Method." It is comparatively difficult to understand a large method in comparison to a

number of small methods. As a general rule, any method which has more than 20 lines of code (LOC) is considered bad and any code which has less than 10 line of code is considered good (Whitehead, 2009). After running some sample test codes we found out that most of the methods which have less than 15 lines of codes are good.

- ***Large Class:*** A large class is a class which tries to do too much, i.e. when a class has too many responsibilities and has a large number of instances, variables and methods in the system. These classes are classified as the "Large Class" smell.
- ***Long Parameter List:*** When the number of parameters passed to a method is more than what is actually required for the functionality of the method, it indicates the presence of "Long Parameter List" smell. Most likely a method which has more than three parameters as the passed argument list, it is considered as a "Long Parameter List" smells (Rutheford, 2010).
- ***Primitive Obsession:*** The "Primitive Obsession" itself is not exactly a code smell but it is more of an indication of a code smell. It is not advisable to use a lot of primitive data type variables just as a substitute for a class in software. If you find that your data structure is sufficiently complex, a class should be written to represent it rather than manipulating the data with the help of many primitive data types.
- ***Data Clumps:*** "Data Clumps" is the existence of similar data types in a number of places. In simpler terms, if you always see the same data

appearing together at multiple places, it probably belongs together and should be combined together to form a class.

3.2.2 Object-Oriented Abusers

Smells of the "Object-Oriented Abuser" kind involve cases when the system does not take advantage of the full capabilities of object-oriented design. A common origin of this problem is programmers having prior experience in procedural programming and lack of training or understanding of object-oriented programming. In the worst case, it introduces classic documented anti-patterns in the software design.

- ***Switch Statements:*** This occurs when a system uses a lot of switch statements which are scattered throughout the code. It may cause duplication in the system too. If any switch statement has more than two switch cases it is considered to be a "Switch Statements" smell (Baddoo, Hall, Wernick, & Zhang, 2008).
- ***Temporary Field:*** A "Temporary Field" smell is said to exist when a variable is in the class scope instead of being in the method scope. It is considered as a code smell as it violates the principle of information hiding.
- ***Refused Bequest:*** When a sub-class inherits unnecessary data and methods from their parent classes, it falls under the category of "Refused Bequest" smell.
- ***Alternate Class with Different Interface:*** If an instance of a method appears in the system with a different signature, it is said to exhibit the "Alternate Class with Different Interface" smell.

- ***Parallel Inheritance Hierarchies:*** "Parallel inheritance Hierarchies" smell occurs when creation of a subclass forces us to create a subclass for another class.

3.2.3 Change Preventers

As the name suggests, this group of code smells consists of smells which makes the process of software modification difficult.

- ***Divergent Changes:*** In a scenario in which a single class is modified for a number of changes made in the system i.e. for most of the functionality a single class is changed and change of one of the method within this class inhibits the change of another method in the class indicates the presence of "Divergent Changes" smell.
- ***Shotgun Surgery:*** This smell is similar to the "Divergent Changes" smell. The only difference is that if a change is made to one of the operations in class, many other classes are changed to bring this change into effect. This indicates the presence of "Shotgun Surgery" smell.

3.2.4 Dispensable

This category includes code smells which contain unnecessary code, such as duplicity.

- ***Lazy Class:*** A class which resides in the system for future use but with no responsibility at present represents a "Lazy Class" smell.
- ***Data Class:*** A class which contains variables and their getter and setter methods is called a data class. These methods are used by other classes to exhibit any of their own behavior. We should avoid classes that passively

store data and methods to operate on that data. Hence it is also categorized as a "Data Class" smell.

- ***Duplicate Code:*** "Duplicate Code" smell occurs when the same code structure is seen at more than one place. It occurs if you see the same expression in two different methods of the same class, or in two sibling subclasses. If "Duplicate Code" smell occurs it is much better to find a way to unify them.
- ***Speculative Generality:*** "Speculative Generality" is suggested by Brian Foote (Brant, Beck, Fowler, Opdyke, & Roberts, 2000). It occurs if the code is developed to handle all sorts of hooks and special cases. This leads to harder understandability and maintenance. This can be identified; if methods or a class are used only by test cases it can be identified as "Speculative Generality" smell.

3.2.5 Encapsulators

This group contains the code smells which deal with the data communication mechanism or encapsulation.

- ***Message Chain:*** "Message Chain" smell occurs when an object makes call to method of another class, which in turn makes call to method of some third class and so on. It is not advisable to have intermediaries as they create undesired dependencies. To consider it statistically if you have a chain of more than two methods which are user-defined, that expression is classified as "Message Chain" smell (Baddoo, Hall, Wernick, & Zhang, 2008).

- ***Middle Man:*** "Middle Man" smell is the method (delegate method) which puts forward the request to the client from another method.

3.2.6 Couplers

Code smells which occur because of coupling issues in the code are included in this category.

- ***Feature Envy:*** The indication of one method of a class seeming more interested in other class rather than the one which contains it is called "Feature Envy" smell.
- ***Inappropriate Intimacy:*** When two classes are tightly coupled with each other and are extensively accessing the private variables of the each other, it exhibits "Inappropriate Intimacy" smell.

3.2.7 Others

Smells which do not fit in any of the above six sub-categories are included into this category.

- ***Incomplete Class Library:*** When a library class exhibits a larger or lesser amount of functionality than what is required it is said to exhibit "Incomplete Class Library" smell.
- ***Comments:*** Comments are considered as good. If they are excessively used in the code they are classified into "Comments" smell.

To improve the design of an existing system, we need to apply refactoring methods to the detected smells. For that we need to know where and when to apply which refactoring technique. To determine where refactoring technique is needed, we need to identify code

smells. The refactoring should be applied when a new function is added, a fix to a bug is made in the code or when there is a code review (Fowler et al., 2000).

4 Refactoring Techniques

"Refactoring is changing the structure of a program without changing its functionality" (Gallardo, 2003). As mentioned earlier, there are, in all, 72 refactoring techniques which are categorized into seven sub-categories.

Category	Refactoring Technique
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
Moving Features between Objects	• Extract Class • Introduce Local Extension • Hide Delegate • Move Field • Inline class • Move Method • Introduce Foreign Method • Remove Middle Man
Organizing Data	• Change Bidirectional Association to Unidirectional • 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 • Replace Subclass with Fields • Replace Type Code with Class • Replace Type Code with State/Strategy • Replace Type Code with Subclasses • Self Encapsulate Field
Simplifying Conditional Expressions	• Consolidate Conditional Expression

	• Consolidate Duplicate Conditional Fragments • Decompose Conditional • Introduce Assertion • Introduce Null Object • Remove Control Flag • Replace Conditional with Polymorphism • Replace Nested Conditional with Guard Clauses
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 • 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
Big Refactoring	• Convert Procedural Design to Object • Tease Apart Inheritance • Extract Hierarchy • The Nature of the Game • Separate Domain from Presentation

Table 2: Refactoring Techniques and Sub-categories

This chapter will describe 18 of these 72 refactoring techniques in detail, which can be applied to five of the code smells that Java Smell Detector (JSD) detects depending upon scenario.

4.1 Extract Method

Introduction

"Extract Method is one of the most common refactoring (Fowler et al., 2000)." It can be implemented when the logic of a method is complex and not easy to understand, and if the method body is too long or needs a lot of comments to understand its purpose. In these scenarios these fragments can be turned into individual methods.

During creation of these new individual methods, method names should be chosen properly, as they work well only when they are named properly. Small methods increase the chances of being reused by other methods if they are finely grained, thus reducing code duplication. It also provides the functionality of reading the method names as the comments, thus providing better understanding of the code.

Mechanics

- Create a new method and name it as per its functionality (and not by how does it achieve it)
- Take the intended lines of code from the source method and copy them to the target method.
- Go through the extracted code to capture the references to variables that are local in scope of the source method. Treat these as local variables and parameters to the method.

- Look for any temporary variables that are used only within the extracted code. If any, retain them as temporary variables in the target method.
- Check if any modifications have been made to the local-scope variables that need to be returned. If the number of such variables is limited to one, return it. If it two or more, split the method again or make them final.
- Make a call to the target method in place of the extracted code.
- Compile and test. (Fowler et al., 2000)

Example

Bad Method Code

```
void printOwing() {
    // variable declaration
    String _name = _customer.getName();
    String _address = _customer.getAddress();
    int _id = _customer.getId();
    Enumeration e = _orders.elements();
    double outstanding = 0.0;

    // print header
    System.out.println ("**************************");
    System.out.println ("***** Customer Owes ******");
    System.out.println ("**************************");
    // calculate outstanding
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }
    //print details
    System.out.println ("name:" + _name);
    System.out.println ("identification number:" + _id);
    System.out.println ("address:" + _address);
    System.out.println ("amount:" + outstanding);
}
```

Figure 1: Extract Method Bad Code

Refactored Code

```
void printOwing(double previousAmount) {
        printHeader();
        double outstanding = getOutstanding();
        printDetails(outstanding);
}

void printHeader() {
   System.out.println ("*****************************");
   System.out.println ("****** Customer Owes ******");
   System.out.println ("*****************************");
}

double getOutstanding() {
        Enumeration e = _orders.elements();
        double result = 0.0;
        while (e.hasMoreElements()) {
           Order each = (Order) e.nextElement();
           result = each.getAmount();
        }
        return result;
}

void printDetails (double outstanding) {
   System.out.println ("name:" + _customer.getName());
   System.out.println ("identification number:" + _customer.getId());
   System.out.println ("address:" + _customer.getAddress());
   System.out.println ("amount:" + outstanding);
}
```

Figure 2: Extract Method Refactored Code

In the above example, the code is extracted in three separate methods. Before refactoring, the method *printOwing* performed functionalities like printing header, calculating the outstanding amount and printing the details. The method has more than 15 lines of code; as well as multiple functionalities that could be easily identified from the comments. As seen above the method *printOwing* is divided into *printHeader*,

getOutstanding and *printDetails* methods. The "Enumerator e" is declared in the *getOutstanding* method and the return variable outstanding is named as result of the better understanding of the code. The variable outstanding is inline with the method calling in the main method *printOwing*. Also the other variable, *name, address, id* are moved to *printDetails* method and are inline with the print statements.

4.2 Inline Method

Introduction

Too much indirection between the methods in a code is a sign to apply "Inline Method" refactoring technique. "Inline Method" is mostly applied on the methods whose body is as clear as the name itself as it shows the presence of needless indirection. It can also be applied when a group of methods seems to be badly refactored. During this scenario the inline methods can extract into one big method and then re-extracted using the "Extract Method" refactoring technique.

Mechanics

- Check that the method is not overridden in the child classes.
- Find and replace each call to the method with the definition of the method.
- Compile and test.
- Delete the method definition (Fowler et al., 2000).

Example

Bad Method Code

```
// Method 1
int getRating() {
      return (moreThanFiveLateDeliveries()) ? 2 : 1;
}
 // Method 2
boolean moreThanFiveLateDeliveries() {
      return _numberOfLateDeliveries > 5;
}
```

Figure 3: Inline Method Bad Code

Refactored Code

```
int getRating() {
      return (_numberOfLateDeliveries > 5) ? 2 : 1;
}
```

Figure 4: Inline Method Refactored Code

In the above example, the *getRating* method is calling the method *moreThanFiveLateDeliveries* which checks whether the number of late deliveries is greater than five or not. This method body can be easily substituted in the *getRating* method which will make it easier to understand and remove the needles indirection.

4.3 Replace Method with Method Object

Introduction

In certain scenarios of long method, "Extract Method" refactoring technique cannot be applied due to large presence of local variables. This problem even cannot be solved using "Replace temp with Query" (discussed in chapter 4.4) refactoring technique. In this scenario "Replace Method with Method Object" should be applied. First step is to create a new class which has the same name as the method name. Create a new method named

compute in the new object class. The *compute* method has the same functionality as the source object method. The new object class should also create a final field of the source object and a field for each temporary variable and the parameters in the method. These all fields will be set during the call of the constructor.

Mechanics

- Create a new class with the name of the method.
- Create a final field in the new class of the source object. Also, create a field for each temporary variable and parameter in the method.
- Create a constructor of the new class that takes the source object and each parameter.
- Create a method named *compute* in the new class.
- Copy the body of the original method into *compute*. Use the source object field for any invocations of methods on the original object.
- Compile.
- Replace the old method call with the new one and call *compute* (Fowler et al., 2000)

Example

Bad Code

```
Class Account { ...
    int gamma (int inputVal, int quantity, int yearToDate) {
        int importantValue1 = (inputVal * quantity) + delta();
        int importantValue2 = (inputVal * yearToDate) + 100;
        if ((yearToDate - importantValue1) > 100)
            importantValue2 -= 20;
        int importantValue3 = importantValue2 * 7;
        // and so on...
        return importantValue3 - 2 * importantValue1;
    }
```

Figure 5: Replace Method with Method Object Bad Code

Refactored Code

```
class Gamma { ...
   private final Account _account;
   private int inputVal;
   private int quantity;
   private int yearToDate;
   private int importantValue1;
   private int importantValue2;
   private int importantValue3;
   Gamma (Account source, int inputValArg, int quantityArg, int yearToDateArg) {
      _account = source;
      inputVal = inputValArg;
      quantity = quantityArg;
      yearToDate = yearToDateArg;
   }
   int compute () {
      importantValue1 = (inputVal * quantity) + _account.delta();
      importantValue2 = (inputVal * yearToDate) + 100;
      importantThing();
      int importantValue3 = importantValue2 * 7;
      // and so on...
      return importantValue3 - 2 * importantValue1;
   }
   void importantThing() {
      if ((yearToDate - importantValue1) > 100)
         importantValue2 -= 20;
   }

class Account { ...
   int gamma (int inputVal, int quantity, int yearToDate) {
      return new Gamma(this, inputVal, quantity, yearToDate).compute();
   }
}
```

Figure 6: Replace Method with Method Object Refactored Code

In the above example, the *gamma* method of *Account* class in the bad code section had too many local and passed variables; thus making it not possible to apply the "Extract

Method." In the refactored code section, a separate class named *Gamma* containing the *compute* method is created which is later divided using the "Extract Method." The constructor of the *Gamma* class initializes the local variables of the class also passing the object of the source object. The old method of the *Account* class is changed with one which creates the instance of the new class and calls the *compute* method.

4.4 Replace Temp with Query

Introduction

The main cause of the long methods is usually the temporary variable declaration which assigns value only once. These temporary variables are usually not of much use as their scope is only inside the method. Thus the right hand side of these variables can be extracted into individual methods and the variable can be declared as final making it easier to apply "Extract Method." This refactoring technique is called as "Replace Temp with Query."

Mechanics

- Look for an already assigned temporary variable.
- Declare *temp* as final.
- Create a new method and use the right-hand side of the assignment as body of method.
- Compile and test (Fowler et al., 2000).

Example

Bad code

```
double getPrice() {
   int basePrice = _quantity * _itemPrice;
   double discountFactor;
   if (basePrice > 1000) discountFactor = 0.95;
   else discountFactor = 0.98;
   return basePrice * discountFactor;
}
```

Figure 7: Replace Temp with Query Bad Code

Refactored Code

```
double getPrice() {
   return basePrice() * discountFactor();
}
private double discountFactor() {
   if (basePrice() > 1000) return 0.95;
   else return 0.98;
}
private int basePrice() {
   return _quantity * _itemPrice;
}
```

Figure 8: Replace Temp with Query Refactored Code

In the above example the *basePrice* and the *discountedFactor* both are the temporary variables in *getPrice* method. These variable declarations and assignments are replaced by creating individual methods which are used as and when required in the method.

4.5 Hide Delegate

Introduction

One of the important features of the object-oriented programming is Encapsulation - meaning information hiding. It is described as the protective barrier to protect the variables

in a class to be directly accessed by the other objects. This functionality makes it easier to modify the code and reduce the number of objects to be informed about the change. This type of delegation is mainly observed in case of message chain. To remove this dependency the delegate method can be created thus reducing the number of objects to be informed.

To understand it much better, consider a scenario of client and server. If a client calls a method defined on one of the fields of the server object, the client needs to know about this delegate object. If the delegate objects changes, the client may have to change as well. This dependency can be removed by placing a delegating method on the server to hide the delegation. This leads to limit the changes to the server and does not propagate to the client.

Mechanics

- Create a simple delegating method for each method on the delegate on the server.
- Change the client to call the server. Verify that it is working after each change.
- If there are no clients that access the delegate, remove the server's accessor for the delegate.
- Compile and test (Fowler et al., 2000).

Example

Bad Code

```
class Person {
   Department _department;

   public Department getDepartment() {
      return _department;
   }
   public void setDepartment(Department arg) {
      _department = arg;
   }
 }

 class Department {
   private Person _manager;

   public Department (Person manager) {
      _manager = manager;
   }

   public Person getManager() {
      return _manager;
   }
}
 class Client {
   private Person supervisor;

   public Preson getSupervisor (Person manager) {
        supervisor = abc.getDepartment().getManager();
        return supervisor;
   }
 }
```

Figure 9: Hide Delegate Bad Code

Refactored Code

```
class Person {
   Department _department;

   public void setDepartment(Department arg) {
      _department = arg;
   }
   public String getManager() {
        return _department.getManager();
   }
 }

 class Department {
   private Person _manager;

   public Department (Person manager) {
      _manager = manager;
   }

   public Person getManager() {
      return _manager;
   }
}
 class Client {
   private Person supervisor;

   public Preson getSupervisor (Person abc) {
        supervisor = abc.getManager();
        return supervisor;
   }
 }
```

Figure 10: Hide Delegate Refactored Code

In the above example, the client class has a method *getSupervisor* which gets the supervisor of an employee. To retrieve the supervisor, it first gets the department of the employee using *Person* class and then gets the manager of that department using the *Department* class. Thus the *Client* class needs to know about both the *Department*, as well as

the *Person* class showing the delegation. This delegation can be removed by creating another method in the *Person* class *getManager* hiding the delegation from the client. Later the *getDepartment* method can be also removed from the person class if it's not being used anywhere else.

4.6 Move Method

Introduction

"Move Method" can be applied if the class has too much behavior or if the classes are highly coupled. The best way to identify when to use move method is to look into each method and find if it refers another object more than the object it lives in. This helps to identify the method to be moved and also the destination class of that method.

Mechanics

- Examine all features used by any method defined in the source class. Consider if their removal is required.
- Declare a method in the target class and copy the source to the target method.
- Ensure that the new method works.
- Determine how to refer the correct target object from the source.
- Change the source method into a delegating method.
- Compile and test.
- Take a decision if removal of the source method is required.
- In case source method is removed, replace all the references with the target method references.
- Compile and test (Fowler et al., 2000)

Example

Bad Code

```
class Account...
  private AccountType _type;
  private int _daysOverdrawn;

  double overdraftCharge() {
    if (_type.isPremium()) {
      double result = 10;
      if (_daysOverdrawn > 7) result += (_daysOverdrawn - 7) * 0.85;
      return result;
    }
    else return _daysOverdrawn * 1.75;
  }

  double bankCharge() {
    double result = 4.5;
    if (_daysOverdrawn > 0) result += overdraftCharge();
    return result;
  }
```

Figure 11: Move Method Bad Code

Refactored Code

```
class AccountType...
  double overdraftCharge(Account account) {
    if (isPremium()) {
      double result = 10;
      if (account.getDaysOverdrawn() > 7)
        result += (account.getDaysOverdrawn() - 7) * 0.85;
      return result;
    }
    else return account.getDaysOverdrawn() * 1.75;
  }

class Account...
  double bankCharge() {
    double result = 4.5;
    if (_daysOverdrawn > 0) result += _type.overdraftCharge(_daysOverdrawn);
    return result;
  }
```

Figure 12: Move Method Refactored Code

In the above example, *Account* class (Bad code) has two methods *bankCharges* and *overdraftCharge*. The *overdraftCharge* method was using the object of *AccountType* class more than its own class showing that it belongs to *AccountType* class more than *Account* Class. Also the *overdraftCharge* method has no functionality in the *Account* Class, so the delegate method of the *Account* class can be removed and a direct call to *overdraftCharge* method of *AccountType* class should be called.

4.7 Remove Middle Man

Introduction

The cost of applying "Hide Delegate" refactoring technique is the formation of the "Middle Man" smell. As discussed earlier "Hide Delegate" helps to maintain Encapsulation.

If the delegate method is added a lot to the server, it becomes a delegate class. Thus it is important to maintain a balance between "Hide Delegate" and the "Remove Middle Man."

Mechanics

- Create an object for the delegate.
- Instead of calling the delegate method, call the method directly from the client.
- Compile and test (Fowler et al., 2000).

Example

Bad Code

```
class Person...
  Department _department;
  public Person getManager() {
    return _department.getManager();
  }
class Department...
  private Person _manager;
  public Department (Person manager) {
    _manager = manager;
  }
class Client {
  private Person supervisor;
  public Preson getSupervisor (Person abc) {
      supervisor = abc.getManager();
      return supervisor;
  }
}
```

Figure 13: Remove Middle Man Bad Code

Refactored Code

```
class Person...
   Department _department;
   public Department getDepartment() {
      return _department;
   }
 class Department...
   private Person _manager;
   public Department (Person manager) {
      _manager = manager;
   }
 class Client {
   private Person supervisor;
   public Preson getSupervisor (Person abc) {
        supervisor = abc.getDepartment().getManager();
        return supervisor;
   }
 }
```

Figure 14: Remove Middle Man Refactored Code

In the above example, *getManager* method in *Person* class (Bad Code) is a 'Middle Man method' as it invokes the *getManager* method of the *Department* class. This middle man is removed by creating a *getDeaprtment* in a *Person* class which return the department type for that person and making necessary changes in the *Client* class. In this case the *getManager* method of the *Person* class is removed, but it can be retained if Encapsulation is required for some of the objects.

4.8 Encapsulate Collection

Introduction

Collection is a group of data which is manipulated as a single object. Java has many collections like Arrays, Iterator, Set, List which consists of their corresponding getter and setter methods. However, the getters should not return the collection object itself, as the

server class will not come to know about the operations on each element of collection. Also it lets the client know what type of Data structure is being used at the server side. For the setter methods the client should be just allowed to remove or add elements to the collection rather than having functionality to clear the whole collection. This in turn reduces the coupling between the client and the server.

Mechanics

- Create *add* and *remove* methods for the collection.
- Initialize the collection with null.
- Find the existing setters and modify them to use *add* and *remove* methods (depending on the operation performed) or have the clients call *add* and *remove* methods.
- Find the users of the getter that modify the collection. Change them to use the *add* and *remove* methods.
- Compile and test (Fowler et al., 2000).

Example

Most of the programmers use arrays which are lot susceptible to modifications and they provide the getters and setters method for the whole array instead of providing for individual elements. So, here we will talk specifically about the array and we will also convert it to the higher collection List.

Bad Code

```
class Skills { …
   String[] getSkills() {
      return _skills;
   }
   void setSkills (String[] arg) {
      _skills = arg;
   }
   String[] _skills; … }
```

Figure 15: Encapsulate Collection Bad Code

Refactored Code

```
class Skill…
   String[] getSkills() {
      return (String[]) _skills.toArray(new String[0]);
   }
   void setSkill(int index, String newSkill) {
      _skills.set(index,newSkill);
   }
   void setSkills (String[] arg) {
      _skills = new String[arg.length];
      for (int i=0; i < arg.length; i++)
         setSkill(i,arg[i]);
   }
   List _skills = new ArrayList();
```

Figure 16: Encapsulate Collection Refactored Code

In the above example, *Skill* class (Bad Code) has the get and set method which let its objects manipulate the whole array at any point of time instead and not the individual elements. Thus in the refactored code, the get and the set methods are changed to access and set a specific element at specific index. If an object wants to set a bunch of elements at the same time, it can use the *setSkills* method. The *setSkills* method helps to hide the data structure at the *Skill* class from its caller class.

4.9 Encapsulate Field

Introduction

Every declared variable in an object-oriented language should have a corresponding getter and setter method. If a class does not have a getter and setter method, its object can interact and change the field value directly. This leads to let the other objects change and access data without the owning object being notified.

Mechanics

- Create getters and setters for the field.
- Find the clients that refer the field outside the class and replace the calls with calls to getters and setters created in first step.
- Change the declaration of the field to *private*.
- Compile and test (Fowler et al., 2000).

Example

Bad Code

```
class Info { …
  public String _name
}
```

Figure 17: Encapsulate Field Bad Code

Refactored Code

```
class Info {…
  private String _name;
  public String getName() {return _name;}
  public void setName(String arg) {_name = arg;}
}
```

Figure 18: Encapsulate Field Refactored Code

In the above example, the *variable_name* is public and is not having getter and setter method in the *Info* class of Bad code. As it is public the calling object can directly change or access the *variable_name* without letting the *Info* class know about it. In the Refactored code its corresponding getter and setter methods are created thus encapsulate the *variable_name*.

4.10 Replace Type Code with Subclasses

Introduction

"Type Code" is the code implemented in if-then-else block or switch cases as it depends on which condition being satisfied. These if-then-else block or switch cases need to be refactored using "Replace Conditional with Polymorphism." Before applying this refactoring each "Type Code" needs to be with inheritance structure using "Replace Type code with Subclasses." Another reason to apply "Replace Type code with Subclasses" refactoring technique is when certain features are only present in specific type of condition.

The major advantage of "Replace Type code with Subclasses" is seen when a new variant to the "Type Code" needs to be added. If polymorphism is used only a new subclass needs to be added for the variant whereas in case of "Type Code" all the occurrence of them needs to be searched to add a condition individually.

Mechanics

- Encapsulate the “Type Code”.
- Create a subclass for each value of the “Type Code”. Override the getter method of the “Type Code” in the subclass to return the relevant value.
- Verify the functionality.
- Remove the “Type Code” field from the superclass. The accessors for the “Type Code” are declared abstract.
- Compile and test (Fowler et al., 2000).

4.11 Decompose Conditional

Introduction

Conditional logic can in itself become complicated in condition checking as well as implementation for each condition. In practical, the code both in the condition check and the actions can be individually extracted into separate methods as these method names will make it easier to understand the logical flow of this block of code.

Mechanics

- Create a new method whose body constitutes of the condition.
- Create new methods for the extracted “then” part and the “else” part (Fowler et al., 2000).

Example

Bad Code

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

Figure 19: Decompose Conditional Bad Code

Refactored Code

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

private boolean notSummer(Date date) {
    return date.before (SUMMER_START) || date.after(SUMMER_END);
}

private double summerCharge(int quantity) {
    return quantity* _summerRate;
}

private double winterCharge(int quantity) {
    return quantity* _winterRate + _winterServiceCharge;
}
```

Figure 20: Decompose Conditional Refactored Code

In the above example, condition check as well as then and the else condition is extracted in individual methods. This makes the conditional check easier to understand and modify if needed.

4.12 Replace Conditional with Polymorphism

Introduction

"Polymorphism is a mechanism that is often summarized as one interface multiple implementations" (O'Neil, 1999). "Replace conditional with Polymorphism" technique is applied when you wish to avoid an explicit conditional for each "Type Code" when the behavior varies. As a result, the "Type Codes" (switch statements, if-then-else statements) are avoided in object-oriented program. In this scenario if polymorphism is used, only a separate subclass needs to be created whereas in conditional type scenario each conditional block needs to be searched and modified separately.

Note: The "Replace conditional with Polymorphism" technique can only be applied if there is an inheritance structure. If there is no inheritance structure it can be created using "Replace Type code with Subclasses." refactoring technique.

Mechanics

- If there is a conditional statement in the larger method, take it apart using "Extract Method."
- If required, use "Move Method" to place the conditional at the top of the inheritance structure.
- In each of the subclasses create a subclass method which overrides the conditional statement method. Copy the body of the corresponding conditional statement into the subclass method and adjust it to fit.
- Remove the copied part of the conditional statement.
- Compile and test.
- Repeat the same steps for each leg of the conditional statement.

- Change the superclass method definition to abstract (Fowler et al., 2000).

4.13 Hide Method

Introduction

With the expansion of the system, the classes might be updated performing more functionality. It might also happen that the getter and setter method are replaced and no longer accessible to the client. Thus the visibility of these methods should be hidden from the client by making them private. Also if these methods are no longer needed then the methods should be permanently removed from the code.

Mechanics

- Check for opportunities to make a method more private.
- Make each method as private as you can.
- Compile and test (Fowler et al., 2000).

4.14 Introduce Parameter Object

Introduction

Often a group of parameter list passed together in different methods of same class or different class. This parameter list usually produces the "Long Parameter List" smell. These parameters go along each other and can be extracted into a separate class making the code more consistent and easier to understand. Later the behavior for these parameters can also be moved to this class and helps in reducing the duplicate code.

Mechanics

- "Create an immutable class to represent the group of parameters you are replacing.
- Use "Add Parameter" for the new data clump and use a null for this parameter in all the callers.
- For each parameter in the data clump, remove the parameter from the signature. Modify the callers and method body to use the parameter object for that value.
- Compile and Test.
- When you have removed the parameters, look for behavior that you can move into the parameter object with "Move Method" (Fowler et al., 2000)."

4.15 Preserve Whole Object

Introduction

When a method passes to many parameters from a single object "Preserve Whole Object" can be applied. The problem with this arises when a new data value is added to the called object. In such a situation all the calls to this method need to be find out and changed. This can be avoided by passing the whole object letting the caller object to decide what to use. This reduces the "Long Parameter List" smell as well as makes the parameter list more robust.

Mechanics

- Create a new parameter of the data object.
- Determine the parameters that need to be obtained from the object.

- Within the method body, take one parameter and replace its references by invoking an appropriate method on the object parameter.
- Delete the parameter.
- Compile and test.
- Repeat for each parameter of the object.
- Remove the code in the calling method that obtains the deleted parameters.
- Compile and test (Fowler et al., 2000).

Example

Bad Smell

```
class Room...
 boolean withinPlan(HeatingPlan plan) {
    int low = daysTempRange().getLow();
    int high = daysTempRange().getHigh();
    return plan.withinRange(low, high);
 }
```

Figure 21: Preserve Whole Object Bad Code

Refactored Code

```
class Room...
boolean withinPlan(HeatingPlan plan) {
   return plan.withinRange(daysTempRange());
}
```

Figure 22: Preserve Whole Object Refactored Code

In the above example, for the Bad code section, the *withingRange* method is passed two parameters which are retrieved from the same object *daysTempRange*. This increases the size of *withinPlan* method and also increases the size of parameter list of *withingRange*

method. Also if we add any parameter to the *daysTempRange* which needs to be passes to the *withinRange* method it needs to be explicitly taken care of at all the calling occurrence of *withinRange* method. In case of the Refactored Code section, the only change which needs to be made is inside the *withinRange* method as the whole method object of *daysTempRange* is passed and not the explicit values.

4.16 Replace Parameter with Explicit Methods

Introduction

"Replace Parameter with Explicit Methods" is applied for a particular case of set methods. If the parameter passed to the caller method decides which value to set using "Type Code", it is better to extract each condition into a separate method. This makes the code much clearer, helps to avoid conditional behavior and also helps in compile time checking.

Mechanics

- For each value of the parameter, create an explicit method.
- For each part of the conditional, call the appropriate new method.
- Compile and test.
- Get rid of the conditional method once all callers have been changed (Fowler et al., 2000).

Example

Bad Code

```
void setValue (String name, int value) {
  if (name.equals("height"))
     _height = value;
  if (name.equals("width"))
     _width = value;
}
```

Figure 23: Replace Parameter with Explicit Methods Bad Code

Refactored Code

```
  void setHeight(int arg) {
    _height = arg;
}
void setWidth (int arg) {
    _width = arg;
}
```

Figure 24: Replace Parameter with Explicit Methods Refactored Code

For the above example every conditional case of the *setValue* method is extracted into an explicit method as the parameter name decides which condition to call. This makes the code more robust and easy to understand.

4.17 Replace Parameter with Method

Introduction

"Replace Parameter with Method" can be applied if the passed parameter to a method can be retrieved in some other way like by calling a method. This helps to reduce the parameter list and make the code more readable.

Mechanics

- In the body of the method, replace references to the parameter with references to the method.
- Compile and test after each replacement.
- Remove the parameter (Fowler et al., 2000).

Example

Bad Code

```
Public double getPrice() {
  int basePrice = _quantity* _itemPrice;
  discountLevel = getDiscountLevel();
  double finalPrice = discountedPrice (basePrice, discountLevel);
  return finalPrice;
}
```

Figure 25: Replace Parameter with Method Bad Code

Refactored Code

```
public double getPrice() {
   int basePrice = _quantity* _itemPrice;
   double finalPrice = discountedPrice (basePrice);
return finalPrice;
}
```

Figure 26: Replace Parameter with Method Refactored Code

In Bad Code section, the method *discountedPrice* is passed on a parameter *discountLevel* which is retrieved by calling the method *getDiscountLevel* method. Instead of passing the *discountLevel* as a parameter to *discountedPrice*, it can be retrieved in the body by explicitly calling the *getDiscountLevel* method.

4.18 Replace Delegation with Inheritance

Introduction

"Replace Delegation with Inheritance" is applied when the methods of a class is delegating their work to methods of some other class. This shows that the delegating class can be a subclass. This can be only implemented if the delegating class is using all the methods of the superclass.

Mechanics

- "Make the delegating object a subclass of the delegate.
- Set the delegate field to be the object itself.
- Remove the simple delegation methods.
- Compile and test.
- Replace all other delegations with calls to the object itself.
- Remove the delegate field (Fowler et al., 2000)."

Example

Bad Code

```
class Employee {
  Person _person = new Person();

  public String getName() {
     return _person.getName();
  }
  public void setName(String arg) {
      _person.setName(arg);
  }
  public String toString () {
     return "Emp: " + _person.getLastName();
  }
}

class Person {
  String _name;

  public String getName() {
     return _name;
  }
  public void setName(String arg) {
     _name = arg;
  }
  public String getLastName() {
     return _name.substring(_name.lastIndexOf(" ")+1);
  }
}
```

Figure 27: Replace Delegation with Inheritance Bad Code

Refactored Code

```
class Person {
  String _name;

  public String getName() {
     return _name;
  }
  public void setName(String arg) {
     _name = arg;
  }
  public String getLastName() {
     return _name.substring(_name.lastIndexOf(' ')+1);
  }
 }

class Employee extends Person {
  public String toString () {
      return "Emp: " + getLastName();
  }
}
```

Figure 28: Replace Delegation with Inheritance Refactored Code

In the above example, the class *Employee* is delegating all its work to the class *Person*. This shows that the class *Employee* can be subclass of the class *Person* in turn remove all the delegating methods. Also the reference of calling the *getLastName* method is changed to *toString* method.

5 Java Smell Detector

5.1 Introduction

Code smell detection is a very complex procedure. Careful design analysis of the software system is required for detecting the code smell. It is believed that human intuition is the best way to detect code smell (Roperia, 2009); however, detection becomes more difficult as system size increases. This creates the need for automated detection, especially for larger systems. The automatic detection of code smell can be done using statistical data analysis. However, the detection of all of the 22 smells using statistical data is not possible (Bhalla, 2009).

Our tool - Java Smell Detector (JSD), analyzes Java source code to detect code smell automatically. JSD's functionality is limited to the detection of five code smells: "Switch Cases," "Data Class," "Middle Man," "Long Parameter List," and "Long Method". These code smells will be discussed in detail later in this section. In cases where "Data Class" and "Long Method" smells are detected, JSD provides an option to refactor the code. Refactoring techniques are suggested for "Switch Cases," "Middle Man," and "Long Parameter List" smells.

5.2 Proposed Tool - Java Smell Detector

5.2.1 Implementation Platform

JSD is implemented in Java and uses the Abstract Syntax Tree (AST) parser. Abstract syntax tree is the tree structure representation of the source code in any programming language. Each node of the syntax tree represents a part of the abstract syntactic structure of the source code. The IDE used for the development is Eclipse SDK

3.4.0. For refactoring, JSD uses the built in refactoring API of Eclipse, which is a part of Language Toolkit (LTK). The input of the JSD is a Java project folder.

5.2.2 Architecture and Control Flow

A high level system architecture of JSD is shown in Figure 29:

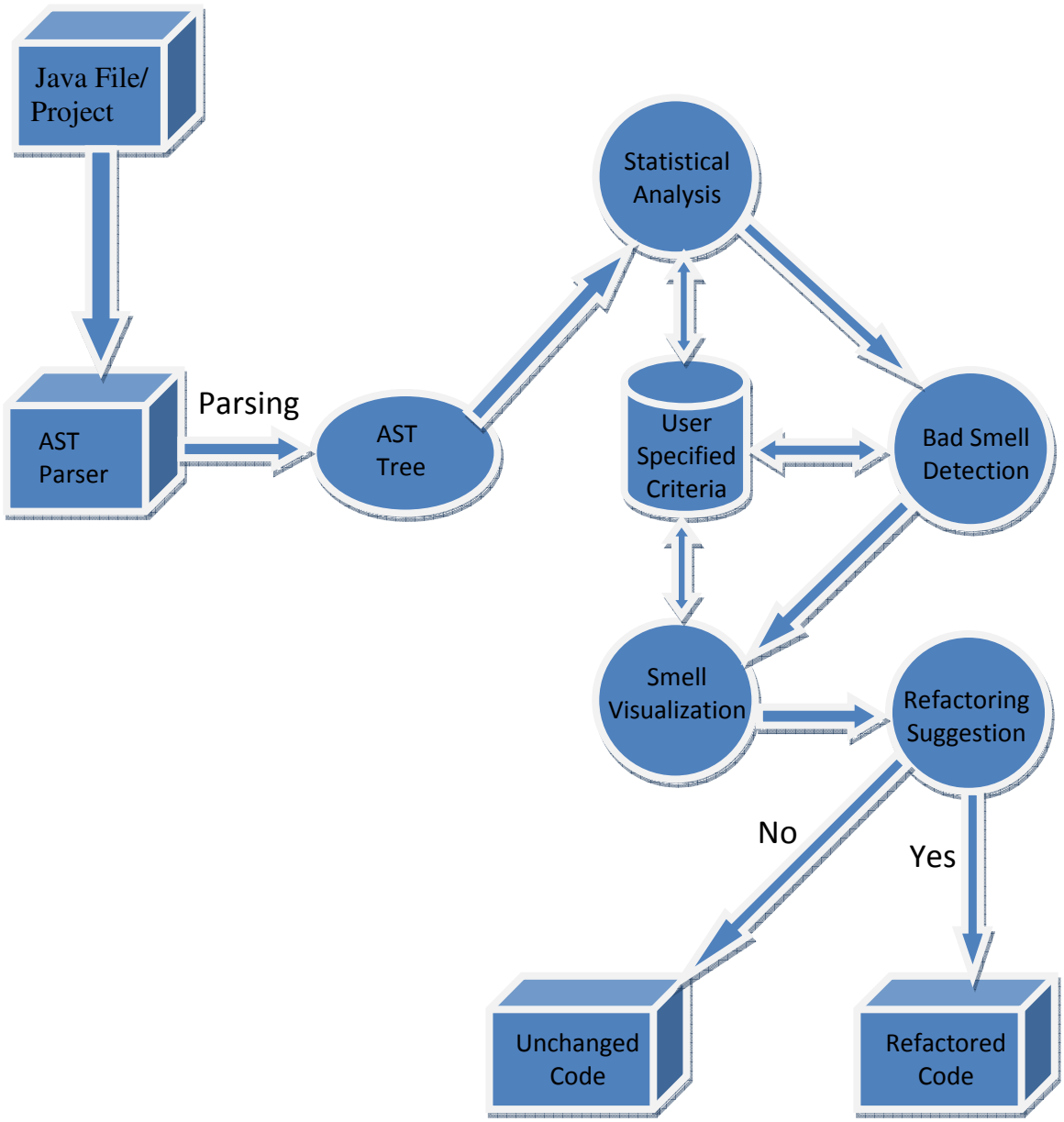


Figure 29: High Level System Architecture.

The classes of the input Java project are parsed through the AST Parser. The detection process is done in two phases: During the initial phase, JSD parses each class to gather statistical data by visiting each AST node and creates an array list of the method and variable names for each class. JSD also creates a list of all the class names used during the detection for "Data Class" smell. During the second phase, the JSD uses the gathered statistical data and the AST to identify the code smells requested by the user. The detected code smells are then presented to the user. JSD also provides the option of applying refactoring technique(s) step by step. The user can choose to accept or discard the refactoring suggestions.

The main interface of the JSD is shown below (Figure 30).

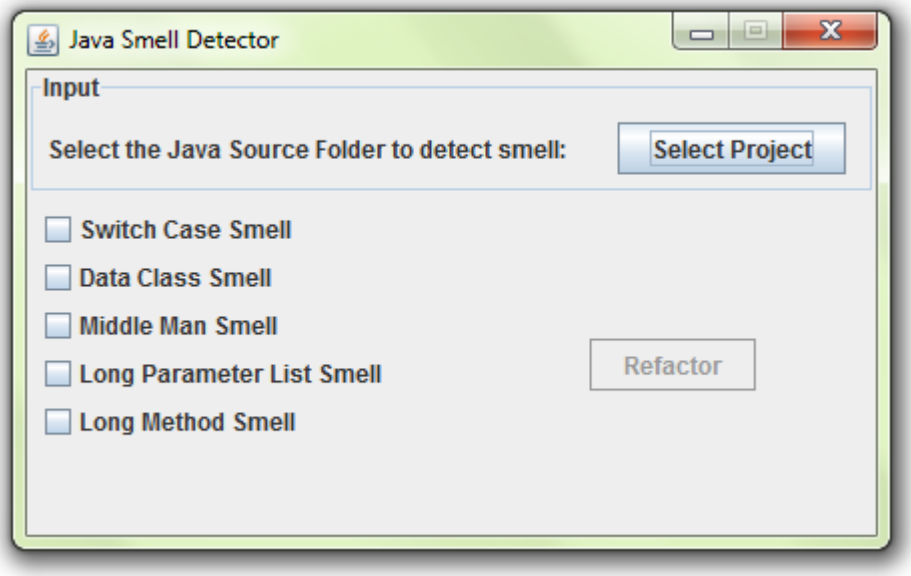


Figure 30: JSD Main GUI

5.2.3 Smell Detected by JSD

Being a relatively small tool, JSD detects five smells as listed below:

- Switch Cases
- Data Class

- Middle Man
- Long Parameter List
- Long Method

Now we will discuss each one of these code smells in detail. We will also review the algorithms used by JSD to detect and suggest refactoring for each code smell.

5.2.3.1 Switch Cases

Problem

When a system uses a significant amount of switch statements scattered throughout the code. This allegedly creates problems of duplicate code.

Description

A switch statement is considered a "Type Code." "Type Code" is a set of programming statements that do not execute all at once; rather only a subset is executed depending upon the control flow. The if-then-else case is also considered a "Type Code." The main problem with "Type Code" is duplication. During the maintenance phase of coding, the same "Type Code" may be scattered at different places in the code. So to add a new clause in the "Type Code" all the occurrences of the "Type Code" need to be searched and changed individually. Polymorphism helps to avoid this problem. Zhang, Baddoo, Wernick, & Hall (2008) suggested that if any "Type Code" consists of more than two cases, it is considered as code smell. For example, in the below code snapshot, the switch case, *movie.getTypeCode,* consists of more

than three cases. Here we can use polymorphism to create an interface and a subclass for each case to refactor the code.

```
int balance = 0;
foreach (Movie movie in movies) {
    switch (movie.GetTypeCode ()) {
        case MovieType.OldMovie: {
            balance += movie.DaysRented * movie.Price / 2;
            break;
        }
        case MovieType.ChildMovie: {
            //its an special bargain !!
            balance += movie.Price;
            break;
        }
        case MovieType.NewMovie: {
            balance += (movie.DaysRented + 1) * movie.Price;
            break:
        }
    }
}
```

Figure 31: Switch Case Smell Example
(Anonymous, n.d.)

Intent

Detect the presence of "Switch Case" smell and suggest the corresponding refactoring technique to remove it.

Detection Technique & Algorithm

Switch Case

1. In AST, visit the "Switch Statement" node for each one of the switch statement blocks.

2. For each “Switch Statement” node, visit “Simple Name” node to retrieve the name of the switch statement and store it in a string. (“Simple Name” node is the node used to express the actual name of any expression of the code in the AST.)
3. Visit all, “Switch Case” nodes and increment the counter to count the number of switch cases.
4. Return the count and the name of the switch statement.
5. If the count is more than three, it is considered to have code smell.

If-then-else

1. Visit, “If Statement” node to visit if-then-else block.
2. Check if, “Else Statement” is there or not using *getElseStatement* method.
    a. If, “Else Statement” exists, visit “If Statement” node and increase the counter. Go back to 2.
    b. If “Else Statement” doesn’t exist, return the counter.
3. Check if the count is more than three, if so it is considered as code smell.

Refactoring Suggestion

To remove the “Switch Case” smell, polymorphism should be used. Polymorphism can be achieved by creating an inheritance structure. Use the “Extract Method” to extract each switch case into individual methods and then, “Move Method” to move each one of the methods into respective classes. Apply “Replace Type Code with Subclasses” to acquire inheritance.

After acquiring the inheritance structure, apply “Replace Conditional with Polymorphism.”

JSD Switch Smell Result Interface

Figure 32 below shows the resulting interface of the “Switch Case” smell for JSD. The “Result” section details the different “Switch Case” smells present in the test data. To assist the user, JSD provides the switch case name with the number of cases and the class name containing it. As each if-then-else statement does not have a unique name, JSD only provides the class name for each if-then-else block of size three or more. The “Refactoring” section provides the steps which a user can follow to refactor the code.

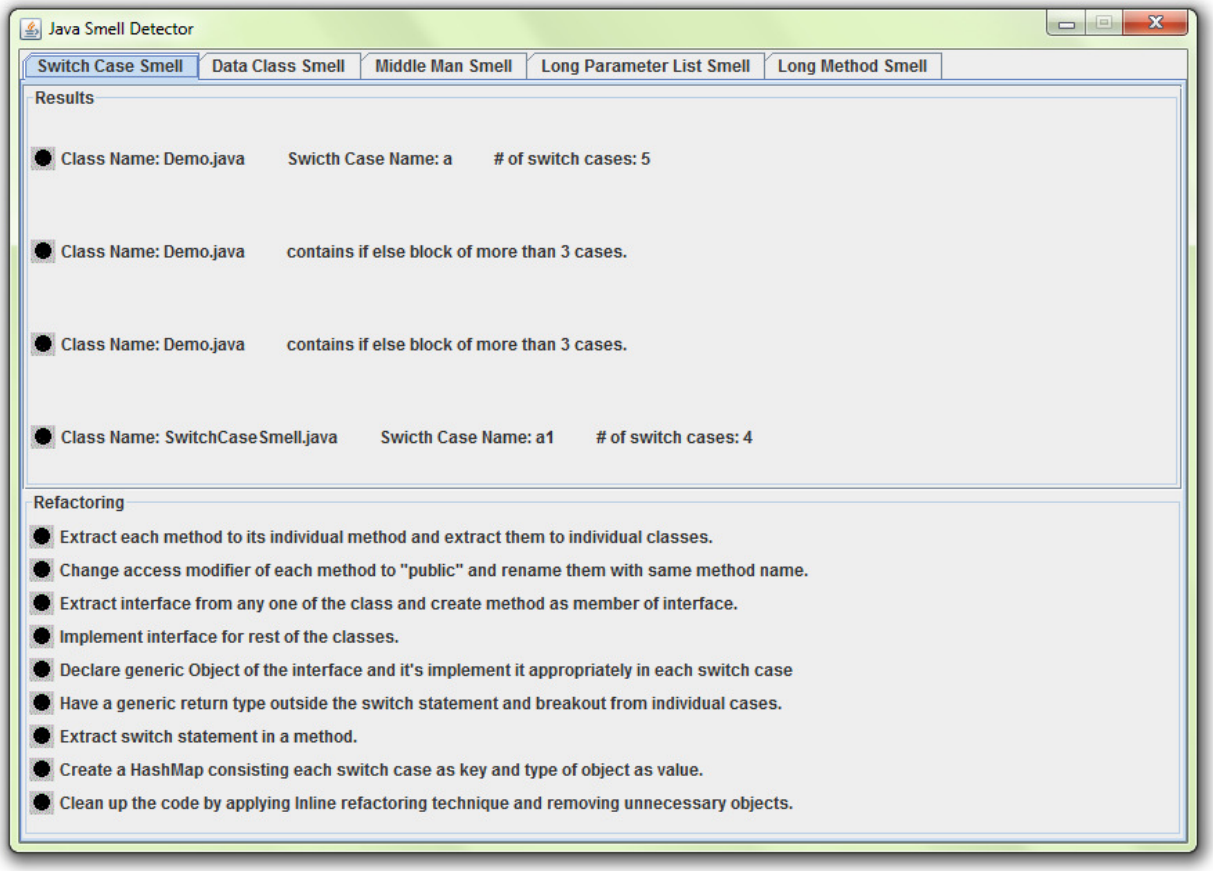


Figure 32: JSD Switch Case Result Interface

5.2.3.2 Data Class

Problem

A class containing only getter and setter methods is considered as the "Data Class" smell.

Description

Getter and setter methods are also known as accessors and mutators, respectively (Roperia, 2009). An accessor is a method that assigns a value to an object and returns it back to the caller object. It does not modify or perform any action on the retrieved object. Mutator is a method that performs

computation on the value and alters the object value. If the mutators of the class do not perform any pre-computation on the values and the class does not contain any other method apart from mutators and accessors, it is categorized as "Data Class" smell. For example, the *Brick* class below is categorized as "Data Class" smell.

```
package model;

public class Brick {

	public Brick(int length, int width, int height)
	{
		this.length = length;
		this.width = width;
		this.height = height;
	}

	public void setLength(int len)
	{
		length = len;
	}

	public void setHeight(int height)
	{
		this.height = height;
	}

	public void setWidth(int width)
	{
		this.width = width;
	}

	public int getHeight()
	{
		return height;
	}

	public int getWidth()
	{
		return width;
	}

	public int getLength()
	{
		return length;
	}

	private int height;
	private int width;
	private int length;

}
```

Figure 33: Data Class Smell Example

Intent

Detect whether a class is "Data Class" smell and provide a refactoring solution.

Detection Technique & Algorithm

1. Visit each "Method Declaration" node of the AST. (Each method definition in AST is represented by a "Method Declaration" node.)
2. The body of the method is represented by "Block" node in the AST. Visit a "Block" node to access the method body. Count the number of statements and store the count in a variable: *numOfStatements*.
3. From the "Block" node, visit the "Return Statement" node (if present) to check for return statements. If the "Return Statement" node exists, subtract one from *numOfStatements*.
4. Visit the "Variable Declaration" node(s) to count the number of local variables declared in the method. Subtract the count from *numOfStatements*.
5. Visit the "Assignment" node through the "Expression" node. (Assignment statements are represented by the "Assignment" node in the AST.) Subtract the total count of visited "Assignment" nodes from *numOfStatements*.
6. Create a hash map with the method name as the key and the counter, *numOfStatements,* as the value.

7. Check the hash map to see if *numOfStatements* is greater than zero. If so exists, remove the corresponding class name from the list of classes created in the initial phase of JSD.
8. The remaining lists of classes have been identified to have "Data Class" smell.

Refactoring Suggestion

In order to remove the "Data Class" smell, look for the functions using these getters and setters. Use "Move Method" to move the identified functions to the data class making sure not to keep it delegated. If the complete function can't be moved to its specified destination, use "Extract Method" to extract part of the function into a separate function and then use "Move Method."

JSD Data Class Result Interface

Figure 34 represents the resulting interface of the JSD for the detection of "Data Class" smell. The "Results" section provides the name of the classes identified to have "Data Class" smell in the test data. The "Refactoring" section provides an interactive approach for the user to refactor for "Data Class" smell. JSD provides a "Move Method" button to move the identified method using these getters and setters. If the user needs to apply "Extract Method" prior to "Move Method," JSD provides an "Extract Method" button as well.

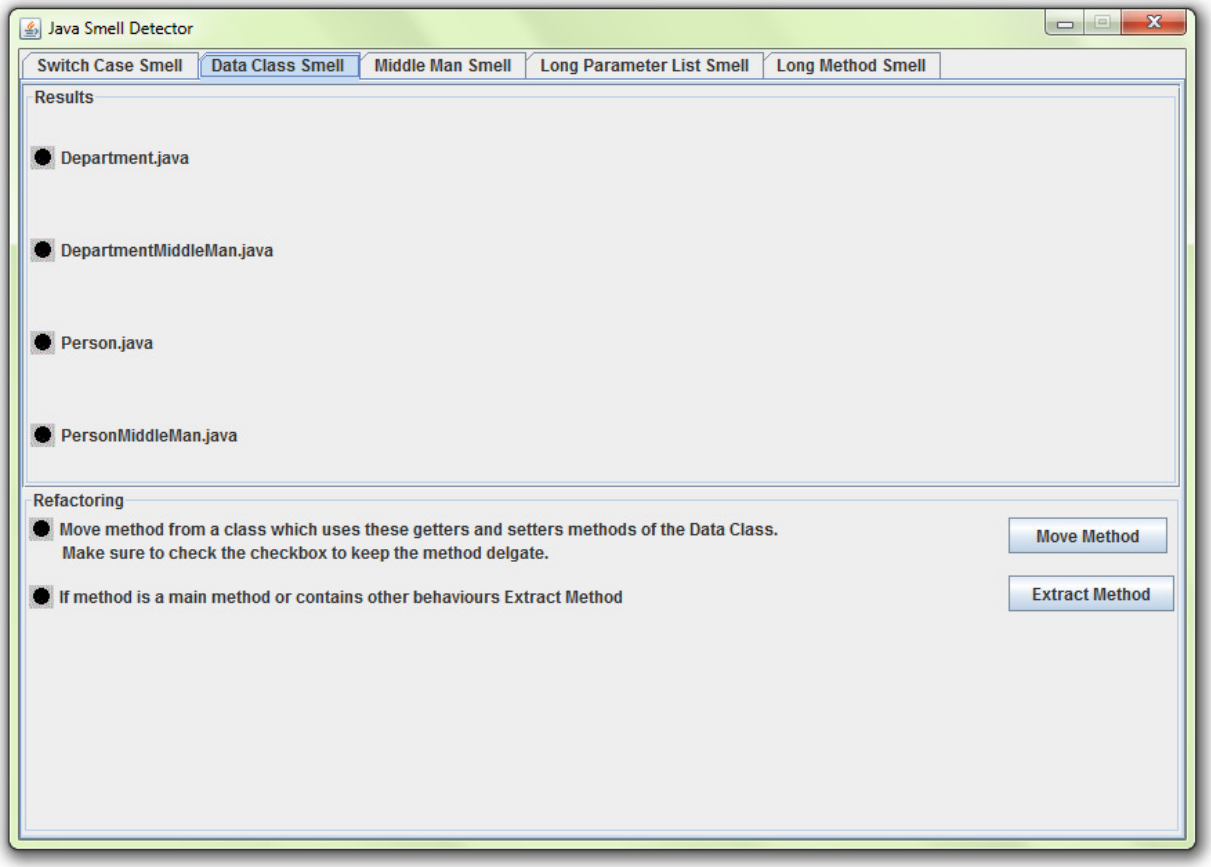


Figure 34: JSD Data Class Result Interface

5.2.3.3 Middle Man

Problem

A method that passes on its work to some other class is considered as "Middle Man" smell.

Description

Encapsulation is one of the vital features of object-oriented language. Encapsulation is when internal execution details are hidden from other objects. Most of the time, encapsulation is achieved at the cost of delegation. For example, if someone wanted to book a room at a hotel, they would call the receptionist at the hotel and enquire about a vacancy. The receptionist would

ask the reservation agent and reply back with availability. The individual requesting the vacancy would not know the means by which the receptionist finds out the availability of rooms. In this process, the message was delegated to the reservation agent by the receptionist. The individual wouldn't know about the reservation agent, hence it satisfies the encapsulation property making the receptionist a middle man. In the example shown in Figure 35, the *Person* class shown below consists of a "Middle Man" smell in the *getManager()* method as it is delegating its work to the *Department* class.

```
class Person...

  Department _department;

  public Person getManager() {

      return _department.getManager();


class Department...

  private Person _manager;

  public Department (Person manager) {

      _manager = manager;

  }
```

Figure 35: Middle Man Smell Example

(Brant, Beck, Fowler, Opdyke, & Roberts, 2000)

Intent

Detect if a method is acting as "Middle Man" or not and suggest corresponding refactoring technique to eliminate it.

Detection Technique & Algorithm

1. Visit each "Method Declaration" node of the AST that represents each of the method definitions in the AST.

2. From the “Method Declaration” node, visit the “Block” node to access the method body and store the number of statements in the method in a counter, *numOfStatements*. Check if *numOfStatements* equals to one:
   If yes, continue to 3 below.
   If no, do nothing.
3. Check if the statement is a return statement by searching the “Return Statement” node.
4. If it is a “Return Statement” node check for the existence of the “Method Invocation” node.
5. If the node exists, retrieve the “Method Invocation” node name and check its existence in the method name list generated during the first pass.
6. In case the name matches, categorize the method as middle man.

Refactoring Suggestion

To remove the “Middle Man” smell, apply the “Remove Middle Man” refactoring technique and interact with the object that directly performs the functionality. Use “Replace Delegation with Inheritance” if a “Type Code” like switch case or if-then-else statement exists and convert the middle man into the respective subclass of the real object.

JSD Switch Smell Result Interface

Figure 36 represents the resulting interface of JSD. In the “Results” section, JSD provides the names of the classes and the method names that are

classified as “Middle Man” smell. JSD also provides the suggestions for removing the smell.

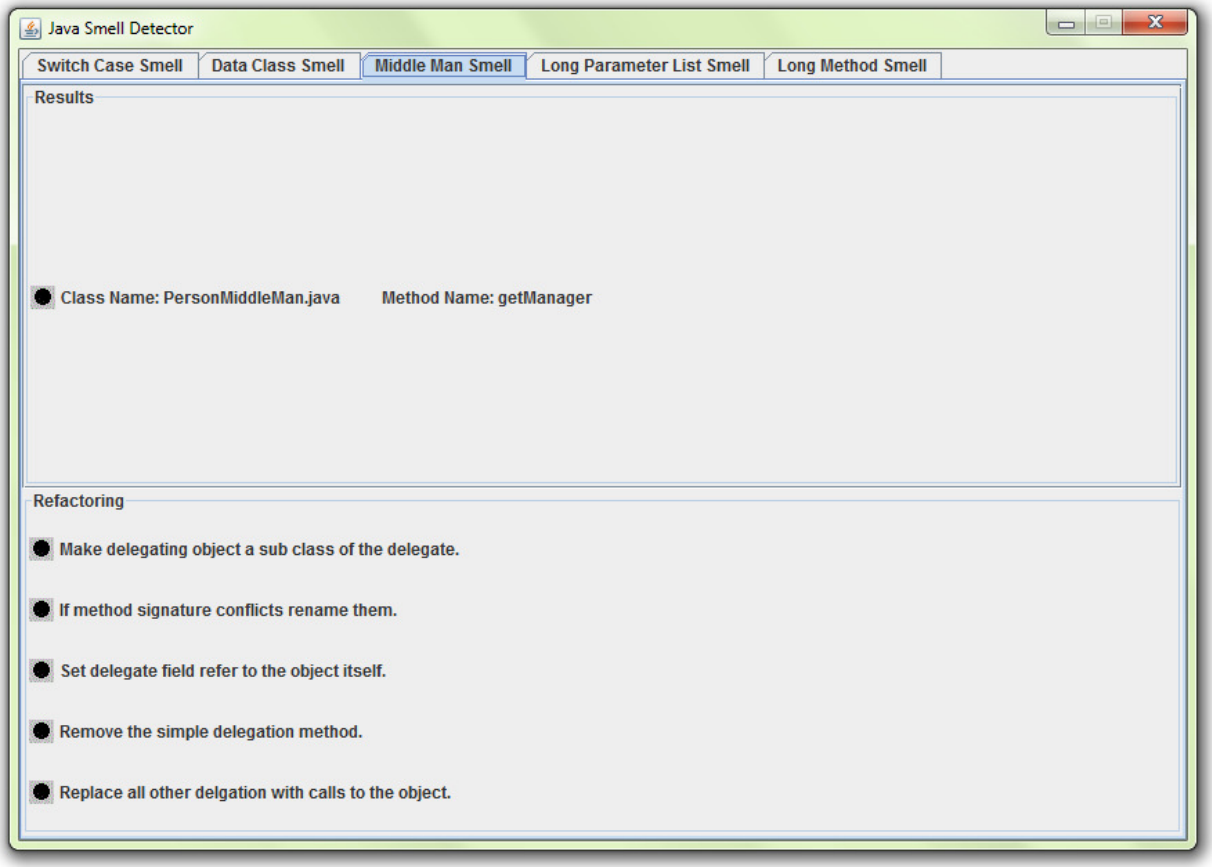


Figure 36: JSD Middle Man Result Interface

5.2.3.4 Long Parameter List

Problem

A method that consists of a long parameter list is categorized as “Long Parameter List” smell.

Description

In procedural programming languages we pass all the required variables as parameters. The other way of accessing the variables in

procedural programming language is to make them global, which is less efficient than passing the variables in a parameter list. In object-oriented languages, instead of passing individual variables, we pass an object and the required variables are retrieved through the object. This makes the code robust, easy to understand, and less susceptible to parameter list changes. If a parameter list contains more than three parameters, it is classified as "Long Parameter List" smell (Rutheford, 2010). The example shown in Figure 37 is an instance of "Long Parameter List" smell.

```
public void add(Name name, int id, Salary salary, Employee boss, Department
dept, String title)
{
   addName(name);
   addID(id);
   addSalary(salary);
   addSupervisor(boss);
   addDept(dept);
   addJobTitle(title);
}
```

Figure 37: Long Parameter Smell Example

(Ronquillo, 2009)

Intent

Categorize a method as a "Long Parameter List" smell and suggest refactoring technique to eliminate it.

Detection Technique & Algorithm

1. Visit each "Method Declaration" node of the AST that represents method definitions in the AST.
2. From the "Method Declaration" node, visit the "Single Variable Declaration" node. (The "Single Variable Declaration" node

represents the individual member of the parameter list.) Increment the parameter counter while visiting each node, and add the parameter name into an array list.

3. If the number of parameters is greater than three, the method has "Long Parameter List" smell.

Refactoring Suggestion

The "Long Parameter List" smell can be removed in the following two ways:

- If the parameters can be replaced by making a call to an existing object, use "Replace Parameter with Method."
- If the parameter list has no common logical object, apply "Introduce Parameter Object."

JSD Switch Smell Result Interface

Figure 38 represents the resulting interface of JSD for detection of the "Long Parameter List" smell. The results section provides the class name, function name and the parameter list of the method. It is important for the parameter list to be presented as Java supports message overloading, which makes the identification of the method easier in the code. The refactoring section of JSD provides steps to refactor the "Long Parameter List" smell that falls into the second criteria, i.e. parameter list that has no common logical object.

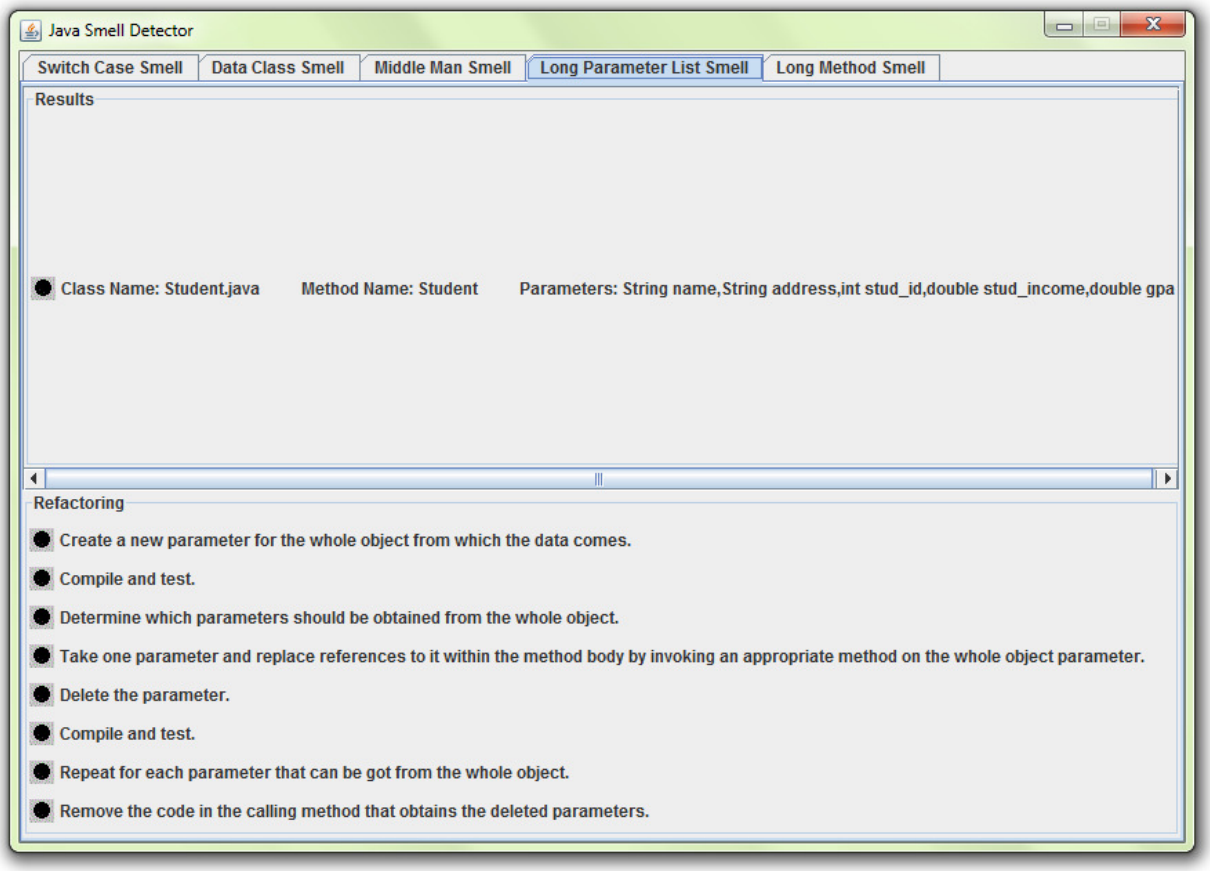


Figure 38: JSD Long Parameter List Result Interface

5.2.3.5 Long Method

Problem

A method that is too long and has a lot of responsibility is considered a "Long Method" smell.

Description

With an increase in the body size of the method, the complexity of the code increases. Such methods tend to perform multiple logics at the same time, making it more complex and increasing the chances of duplicate code. In the long run, a long method becomes more difficult to maintain, as it is hard to understand. There is no precise statistical science to detect and

classify the “Long Method” smell. So, in order to detect a “Long Method” smell, code cannot be judged by its size; rather, it is important to understand the logic of the code. Fowler & Beck (2001) mentioned that the successful way to detect “Long Method” smell is to see if the method needs a significant number of comments to explain its logic. However, Whitehead (2009) mentioned that any method consisting of more than 20 lines of code (LOC) is definitely considered as bad, and any code less than 10 lines of code is considered good. After running some sample test codes and analyzing them, we figured if a method has more than 15 lines of codes it can most likely be considered as “Long Method” smell.

For instance the method shown in Figure 39 consists of “Long Method” smell.

```
public void LongMethod ()
{
    Console.WriteLine ("I'm writting a test, and I will fill a screen with some
useless code");
    IList list = new ArrayList ();
    list.Add ("Foo");
    list.Add (4);
    list.Add (6);

    IEnumerator listEnumerator = list.GetEnumerator ();
    while (listEnumerator.MoveNext ()) {
        Console.WriteLine (listEnumerator.Current);
    }

    try {
        list.Add ("Bar");
        list.Add ('a');
    }
    catch (NotSupportedException exception) {
        Console.WriteLine (exception.Message);
        Console.WriteLine (exception);
    }

    foreach (object value in list) {
        Console.Write (value);
        Console.Write (Environment.NewLine);
    }

    int x = 0;
    for (int i = 0; i < 100; i++) {
        x++;
    }
    Console.WriteLine (x);

    string useless = "Useless String";
    if (useless.Equals ("Other useless")) {
        useless = String.Empty;
        Console.WriteLine ("Other useless string");
    }

    useless = String.Concat (useless," 1");
    for (int j = 0; j < useless.Length; j++) {
        if (useless[j] == 'u') {
            Console.WriteLine ("I have detected an u char");
            } else {
                Console.WriteLine ("I have detected an useless char");
            }
        }

        try {
            foreach (string environmentVariable in
Environment.GetEnvironmentVariables ().Keys) {
                Console.WriteLine (environmentVariable);
            }
```

Figure 39: Long Method Smell Example

(Anonymous, n.d.)

Intent

Detect if a method is introducing “Long Method” smell and suggest a refactoring technique to eliminate it.

Detection Technique & Algorithm

1. Visit each "Method Declaration" node of the AST.
2. Visit the "Block" node to access the method body and count the number of statements in the method.
3. Recursively call step 2 above to visit the "Block" node of any "Type Code" or while statements body and return the count.
4. If the count is more than 15, classify the method to have "Long Method" smell.

Refactoring Suggestion

To remove "Long Method" smell, apply the "Extract Method" refactoring technique. If the new method has a long parameter list after applying the "Extract Method" preprocessing is needed. The long parameter list occurs due to the presence of temporary variables. To eliminate these temporary variables the "Replace Temp with Query" refactoring technique can be applied. The long list of parameters can be reduced by applying "Introduce Parameter Object" or "Replace Method with Method Object."

JSD Switch Smell Result Interface

A screenshot of the JSD results for "Long Method" smell is shown in Figure 40. In the results section, JSD provides the class name, method name and Lines of Codes (LOC) for that method. The refactor section provides an interface to apply the "Extract Method" technique.

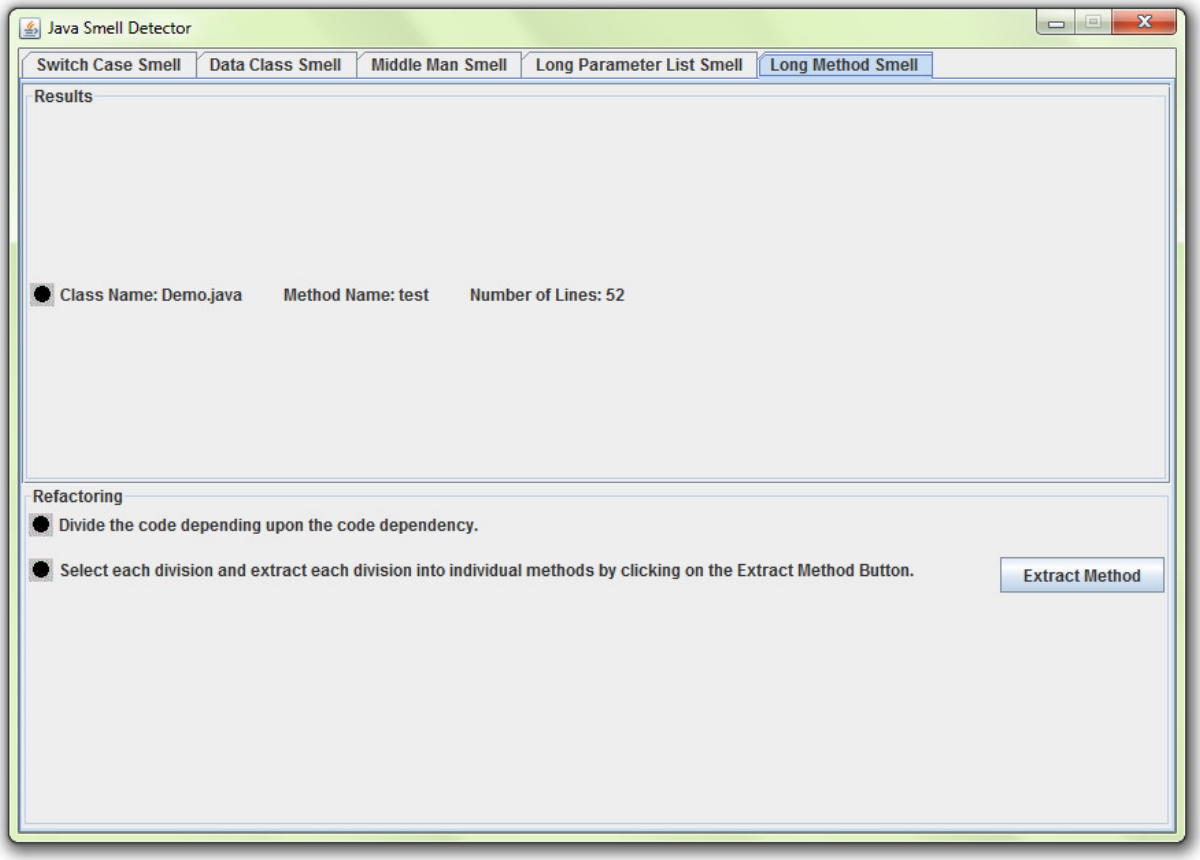


Figure 40: JSD Long Method Result Interface

6 Results

JSD was tested against 28 projects taken from the graduate students of San Jose State University. These selected students have experience of two to five years as Java developers, so the complexity of their code is considerable. Each project has an average of 13 classes. These test codes are their class assignments, hence have a good level of complexity. During the design phase, JSD interface was provided to different users from the technical as well as non-technical background to access the user-friendliness of GUI. The feedbacks were used to improvise the GUI. To test the usability, performance and the code optimization feature of JSD, three different tests were conducted.

1. Identify smells present in each project.
2. Time taken to understand code logic before and after refactoring.
3. Time taken to add functionality in the code before and after refactoring.

6.1 Identify Smells Present in Each Project

During this test, the JSD was run across each of the project and the output was recorded (whether the project contains the specific smell or not). Later they were cross-checked by the graduate students to verify correctness of the smell identified by the tool. Even other classes of the projects were skimmed through to identify other cases which the tool might have missed. The smells identified by JSD in individual projects are represented in the tabular format in Table 3. The table cell marked "Yes" represents the detected code smell in the project enlisted in column 1.

Project #	Switch Case Smell	Data Class Smell	Middle Man Smell	Long Parameter List Smell	Long Method Smell
1	Yes				Yes
2				Yes	Yes

3	Yes		Yes		Yes
4		Yes			
5		Yes			
6	Yes	Yes	Yes		Yes
7	Yes				Yes
8	Yes				Yes
9	Yes				Yes
10	Yes				Yes
11					Yes
12		Yes			Yes
13	Yes		Yes	Yes	Yes
14		Yes			Yes
15	Yes	Yes			Yes
16	Yes	Yes			Yes
17					Yes
18	Yes	Yes	Yes	Yes	Yes
19	Yes	Yes		Yes	Yes
20	Yes	Yes		Yes	Yes
21		Yes	Yes	Yes	Yes
22	Yes	Yes		Yes	Yes
23	Yes		Yes		Yes
24	Yes		Yes		Yes
25	Yes				Yes
26	Yes		Yes		Yes
27	Yes	Yes			Yes
28	Yes	Yes			Yes
Total	**21**	**14**	**8**	**7**	**26**
Percentage	**71.43 %**	**50.00 %**	**28.57 %**	**25.00 %**	**92.86 %**

Table 3: Results - Identified Smells in Each Project

As the above table shows, "Long Method" smell was found in most of the test cases (92.86 %) for which JSD provides an interactive interface to refactor. "Switch Case" smell (71.43%) and "Data Class" smell (50%) were the second and third most detected smell by JSD. JSD also provides an interactive interface to refactor the "Data Class" smell and

provides suggestions to refactor the “Switch Case” smell. JSD detected 28.6% and 25% of “Middle Man” smell and the “Long Parameter List” smell respectively.

6.2 Time Taken to Understand Code Logic Before and After Refactoring.

For this test, four Java developers were chosen ranging from two to seven years of experience. The experience of the users ensured that they had sufficient background knowledge of Java to understand the logic. Three projects (named Project 1, Project 2, and Project 3) from the 28 of the above projects were selected having different difficulty level. The details of each of the three projects are shown in Table 4.

Project #	Complexity Level	Description of Project
Project 1	1	Half Adder Circuit (Observer Observable Pattern)
Project 2	2	A console application determine if entered number is Prime or Square or Sum of square or Biggest Prime number (Master Slave Pattern)
Project 3	3	A GUI application, represents a rectangular shape which changes its size on user input

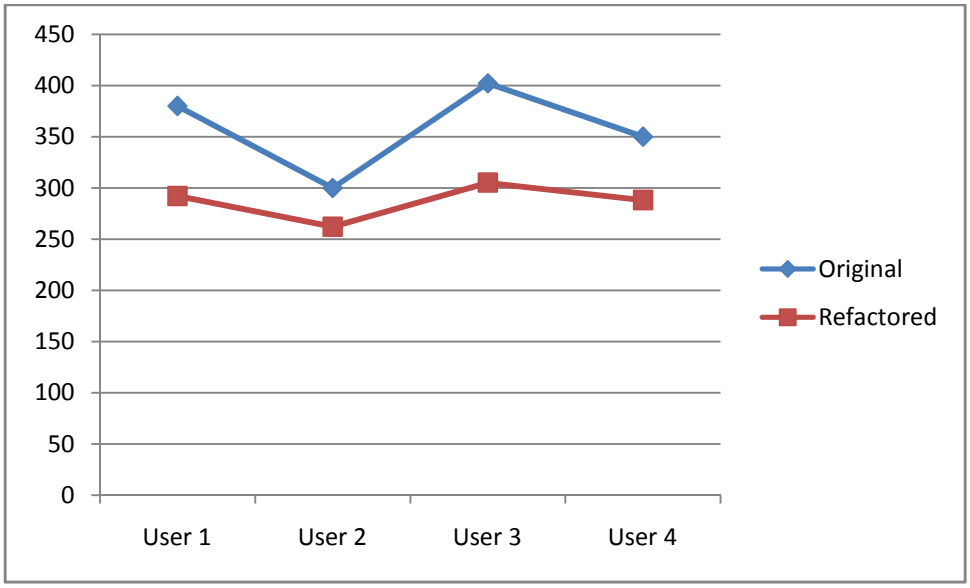
Table 4: Test Projects

Each of the projects was run across JSD and individual smells were detected. The detected “Data Class” smell and “Long Method” smell were refactored by the tool and given to the developers in random order. For Example, if the Project 1 original (non-refactored) code was given first; next time any of the refactored projects or original projects was given. The users were asked to understand the logic of the code for which they were timed. The time taken by each of the users was noted and is enlisted in Table 5.

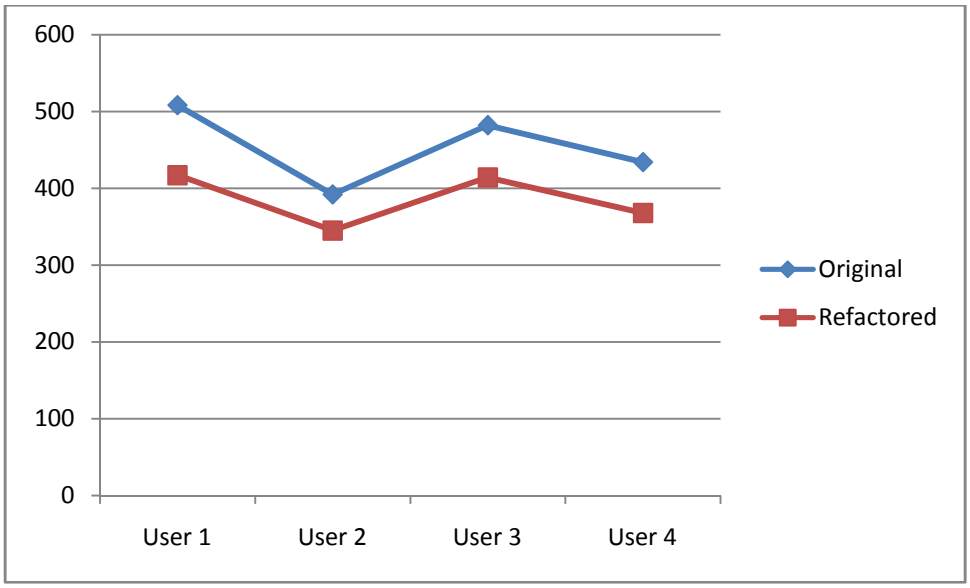
User	Time taken in seconds					
	Project 1		Project 2		Project 3	
	Original	Refactored	Original	Refactored	Original	Refactored
User 1	380	292	508	417	730	562
User 2	300	260	392	345	598	474
User 3	402	305	482	414	602	546
User 4	350	288	434	368	690	500

Table 5: Time taken - Original and Refactoring

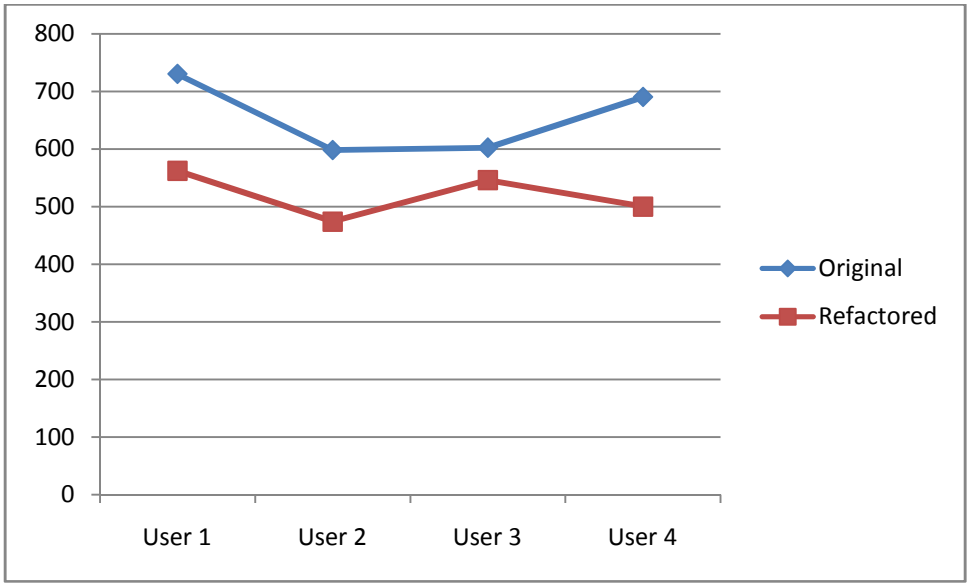
The graphical representation of time taken to understand the logic of original and the refactored code by each user is shown below. Each graph represents the time taken for individual projects.



Graph 1: Time taken to understand Project 1



Graph 2: Time taken to understand Project 2



Graph 3: Time taken to understand Project 3

The statistics above supports the fact that after refactoring; the logic of the code becomes easy to understand. However the user test results cannot be statistically used to validate the significance of results due to small number of users.

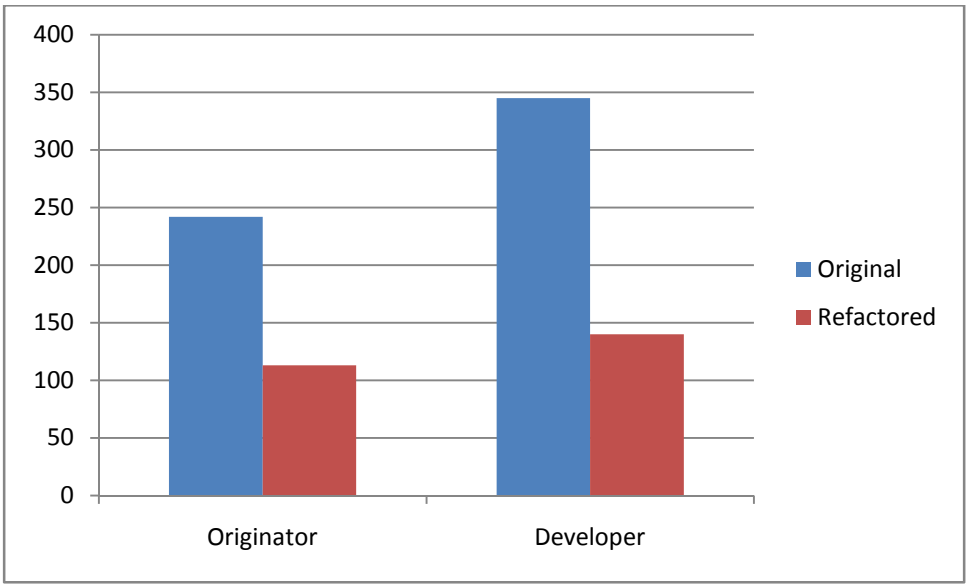
6.3 Time Taken to Add Functionality in the Code Before and After Refactoring.

This test was performed on two separate users, one of them the original writer (originator) of the code and the other an experienced Java developer. For this test, the users were asked to add functionality to the half-adder circuit project. The users were first given the original code and than refactored code one after another. The original code had "Long Method" smell which made it harder for the user to understand the code. In the refactored code, "Long Method" smell was removed by the tool. After each step, the users were asked to understand the code and add the functionality of full-adder circuit. The time taken by each of them was recorded, enlisted in Table 6.

User	Time taken in seconds	
	Half-adder to Full-adder	
	Original	Refactored
Originator	242	113
Developer	345	140

Table 6: Time taken to add functionality

The graphical representation of time difference to add the functionality to the code in original and the refactored code taken by each user is shown in Graph 4.



Graph 4: Time taken to add functionality

The statistics from this test supports that time taken to add functionality in the half-adder project is approximately half for the refactored code than the time taken for the original code. This even holds true for the originator of the code (who wrote the code) as well as for other developer. This shows that the refactored code is easy to maintain and modify than the code with smells and has nothing to do with who developed the code. However, the significance of the results cannot be trusted as the number of users is less. The research needs to be performed on a bigger set of users to trust the results.

7 Conclusion & Future Work

7.1 Conclusion

In this paper, we presented automated code smell detection processes for five of the smells of any object-oriented Java software system. We suggested refactoring techniques for all five of them and provided an interactive approach to refactor two of the five detected smells. These code smells are detected using statistical data gathered while parsing the files. In chapter two, we provided a previous research done on code smells. In chapter three and four, we elaborated about 22 code smells and refactoring techniques respectively. The statistical approach taken to develop JSD and results generated while running the test cases are presented in later chapters.

As it can be seen JSD successfully detected the code smells using the statistical analysis like Lines of Code, Method Invocation, Method Names and others. It is also seen that JSD was able to refactor the smells detected for “Data Class” smell and “Long Method” smell. The results presented in section 6.2 shows that after applying refactoring techniques the code becomes easier to understand and the results in section 6.3 shows that the code becomes easier to maintain and modify.

7.2 Future Work

Currently JSD detects code smells in Java systems only. Similar technique can also be deployed to detect the code smells in other object-oriented languages like C++, Ruby, Python, and C#. Additionally, JSD currently detects only five of the code smells, more research can be done to detect other code smells. Among these five smells the "Long Method" smell can be improved by having a better logical understanding of the method. Moreover, if JSD provides an interactive refactor approach for other code smells, it will become a powerful tool.

Bibliography

Anonymous. (n.d.). *Gendarme Rules Smells*. Retrieved April 10, 2011, from Mono-Project Website: http://www.mono-project.com/Gendarme.Rules.Smells

Bhalla, D. (2009). Automatic detection of bad smells in Java code. Long Beach: ProQuest® Dissertations & Theses .

Chaikalis, T., Tsantalis, N., & Chatzigeorgiou, A. (2008). JDeodorant : Identification and Removal of Type-Checking Bad Smells. *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference* (pp. 329-331). Athens: IEEE.

Code Smell. (2011, March 26). Retrieved April 10, 2011, from Wikipedia: http://en.wikipedia.org/wiki/Code_smell

Cusumano, M. A., & Shelby, R. W. (1995). *Microsoft Secrets.* New York: NY: The Free Press.

Fokaefs, M., Tsantalis, N., & Chatzigeorgiou, A. (2007). JDeodorant: Identification and Removal of Feature Envy Bad Smells. *Software Maintenance, 2007. ICSM 2007. IEEE International Conference* (pp. 519-520). Paris: IEEE.

Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. (2000). *Refactoring Improving The Design Of Existing Code.* New Jersey: Addison-Wesley.

Gallardo, D. (2003, September 9). *IBM Developer Works Technical Library*. Retrieved March 25, 2011, from IBM Developer Works Web Site: http://www.ibm.com/developerworks/library/os-ecref/

Holland, I. & Lieberherr, K. J. (1989). Assuring Good Dtyle of Object-Oriented Programs. *IEEE Software* , 329-331.

Lungu, M., Ersze, G., Marinescu, R., & Mihancea, P. F. (2008). Prodeoos. Timisoara, Romania .

Mantyla, M., Vanhanen, J., & Lassenius, C. (2003). A Taxonomy and Initial Emphirical Study of Bad Smells in Code. *Proceedings of the International Conference on Software Maintenance* (p. 381). IEEE.

Marinescu, R. (2005). Measurment and Quality in Object-Oriented Design. *Proceedings of the 21st IEEE International Conference on Software Maintenance.* IEEE.

Marticorena, R. ,., & Crespo, Y. (2005). Parallel Inheritance Hierarchy: Detection from a static View of the System. *Workshop on Object Oriented Reenginering (WOOR)* , 6.

Moha, N. (2007). DECOR: A Tool for the Detection of Design Defects. *ACM* .

Munro, M. (2005). Product Metrics for Automatic Identification of "Bad Smell" Design Problems in Java Source-Code. *Software Metrics, 2005. 11th IEEE International Symposium* (p. 15). IEEE.

O'Neil, J. (1999). *Teach Yourself Java.* New Delhi: Tata McGraw-Hill.

Ronquillo, U. (2009, 02 18). *I Built His Cage*. Retrieved 03 15, 2011, from I Built His cage Website: http://ibuilthiscage.com/2009/02/18/feature-envy-vs-long-parameter-list/

Roperia, N. (2009, December). JSmell : A Bad Smell detection tool for Java systems. Long Beach, CA, USA: ProQuest Dissertation & Theses.

Rutheford, K. (2010, September 12). *Github Social coding* . Retrieved Feburary 12, 2011, from Github Social coding Website: https://github.com/kevinrutherford/reek/wiki/Long-Parameter-List

Whitehead, J. (2009, January 26). *Jack Basin School of Engineering.* Retrieved July 20, 2010, from Jack Basin School of Engineering Website: http://www.soe.ucsc.edu/classes/cmps020/Winter09/lectures/refactoring-example.ppt

Zhang, M., Baddoo, N., Wernick, P., & Hall, T. (2008). Improving the Precision of Fowler's Definitions of Bad Smells. *Software Engineering Workshop, 2008. SEW '08. 32nd Annual IEEE* (pp. 161-166). Kassandra: IEEE Computer Society.

APPENDIX: Source Code

Package: - GUI

Class: - MainGUI.java

```
package GUI;

import java.awt.BorderLayout;
import java.awt.FlowLayout;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
import java.io.File;
import java.util.ArrayList;

import javax.swing.BorderFactory;
import javax.swing.BoxLayout;
import javax.swing.JButton;
import javax.swing.JCheckBox;
import javax.swing.JFileChooser;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JPanel;

public class MainGUI {

	private JFrame frame;
	private JCheckBox switchCase;
	private JCheckBox dataClass;
	private JCheckBox middleMan;
	private JCheckBox longParaList;
	private JCheckBox longMethod;

	private String path;
	private int counter = 0;
	private ArrayList<String> files = new ArrayList<String>();

	private CheckBoxListener myListener = null;

	private JButton fileChooser = null;
	private JButton refactorButton = null;

	private boolean switchCaseFlag = false;
	private boolean dataClassFlag = false;
	private boolean longMethodFlag = false;
	private boolean longParaListFlag = false;
	private boolean middleManFlag = false;

	public MainGUI() {
		frame = new JFrame("Java Smell Detector");
		configureFrame();
		createTopPanel();
```

```
            JPanel center = new JPanel();
            center.setLayout(new FlowLayout(FlowLayout.LEFT));
            createSmellCheckBoxPanel(center);
            frame.add(center, BorderLayout.CENTER);
            JPanel refactorPanel = new JPanel();
            createRefactorPanel(refactorPanel);
            frame.add(refactorPanel, BorderLayout.EAST);
            frame.setVisible(true);
      }

      private void configureFrame() {
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setSize(430, 260);
            frame.setResizable(false);
            frame.setLocationRelativeTo(null);
            frame.setLayout(new BorderLayout());
      }

      private void createRefactorPanel(JPanel refactorPanel) {
            refactorButtonListener refactorListener = new
refactorButtonListener();
            refactorPanel.setLayout(new BoxLayout(refactorPanel,
BoxLayout.X_AXIS));
            refactorPanel.setBorder(BorderFactory.createEmptyBorder(0, 0, 0,
60));
            refactorButton = new JButton("Refactor");
            refactorButton.addActionListener(refactorListener);
            refactorButton.setEnabled(false);
            refactorPanel.add(refactorButton);
      }

      private void createSmellCheckBoxPanel(JPanel center) {
            JPanel smellCheckBox = new JPanel();
            createCheckBox();
            smellCheckBox.setLayout(new GridLayout(0, 1));
            addCheckBox(smellCheckBox);
            myListener = new CheckBoxListener();
            addCheckBoxListener();
            center.add(smellCheckBox);
      }

      private void createCheckBox() {
            switchCase = new JCheckBox("Switch Case Smell");
            dataClass = new JCheckBox("Data Class Smell");
            middleMan = new JCheckBox("Middle Man Smell");
            longParaList = new JCheckBox("Long Parameter List Smell");
            longMethod = new JCheckBox("Long Method Smell");
            addToolTipToCheckBox();
      }

      private void addToolTipToCheckBox() {
            switchCase
                        .setToolTipText("occurs when a user uses lot of
switch statements. If any Switch Statement has more than 2 Switch Cases it
isconsidered to be a Switch Statements Smell.");
            dataClass
```

```
                        .setToolTipText("When a class contains only variables
and their getter and setter methods is called a Data Class Smell.");
            middleMan
                        .setToolTipText("Middle Man Smell occurs when a
method (delegate method) which puts forward the request to the client from
another method.");
            longParaList
                        .setToolTipText("When the number of parameters passed
to a method is more than what is actually required for the functionality of
the method.");
            longMethod
                        .setToolTipText("A method which contains large number
of lines and performs more than one action is considered as Long Method
Smell.");
      }

      private void addCheckBox(JPanel smellCheckBox) {
            smellCheckBox.add(switchCase);
            smellCheckBox.add(dataClass);
            smellCheckBox.add(middleMan);
            smellCheckBox.add(longParaList);
            smellCheckBox.add(longMethod);
      }

      private void addCheckBoxListener() {
            switchCase.addItemListener(myListener);
            dataClass.addItemListener(myListener);
            middleMan.addItemListener(myListener);
            longParaList.addItemListener(myListener);
            longMethod.addItemListener(myListener);
      }

      private void createTopPanel() {
            JPanel top = new JPanel();
            fileChooser = new JButton("Select Project");
            top.setBorder(BorderFactory.createTitledBorder("Input"));
            top.setLayout(new FlowLayout(FlowLayout.LEFT));
            fileChooser.addActionListener((new ActionListener() {
                  public void actionPerformed(ActionEvent e) {
                        JFileChooser chooser = new JFileChooser(".");
                        chooser.setDialogTitle("Select Project");

      chooser.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);
                        chooser.setAcceptAllFileFilterUsed(false);
                        if (chooser.showOpenDialog(fileChooser) ==
JFileChooser.APPROVE_OPTION) {
                              ifFileChooserSelectedOk(chooser);
                        } else {
                              JOptionPane.showMessageDialog(frame,
                                          "Haven't selected anything yet!!!",
                                          "Selected Project",
JOptionPane.PLAIN_MESSAGE);
                        }
                        refactorButtonConditionCheck();
                  }
```

```
                  private void ifFileChooserSelectedOk(JFileChooser chooser)
{
                        path = chooser.getSelectedFile().toString();
                        path = path.replace("\\", "/");
                        File file = new File(path);
                        fileFilter(file);
                        if (files.size() != 0) {
                              JOptionPane.showMessageDialog(frame,
                                          "The selected path is: " + path,
                                          "Selected Project",
JOptionPane.PLAIN_MESSAGE);
                        } else {
                              JOptionPane.showMessageDialog(frame,
                                          "The selected path doesn't contain
any Java Files",
                                          "Selected Project",
JOptionPane.PLAIN_MESSAGE);
                              path = null;
                        }
                  }
            }));
            top.add(new JLabel(
                        "Select the Java Source Folder to detect smell:
"));
            top.add(fileChooser);
            frame.add(top, BorderLayout.PAGE_START);
      }

      private void fileFilter(File dir) {
            File directory = dir;
            File[] filePath = directory.listFiles();
            for (File f : filePath) {
                  if (f.isDirectory()) {
                        fileFilter(f);
                  } else if (f.toString().endsWith(".java")) {
                        String temp = f.toString().replace("\\", "\\\\");
                        files.add(temp);
                  }
            }
      }

      private void refactorButtonConditionCheck() {
            if (counter > 0 & path != null) {
                  refactorButton.setEnabled(true);
            } else {
                  refactorButton.setEnabled(false);
            }
      }

      class CheckBoxListener implements ItemListener {
            public void itemStateChanged(ItemEvent e) {
                  Object source = e.getSource();
                  itemSelected(e, source);
                  itemDeselected(e, source);
                  refactorButtonConditionCheck();
            }
```

```
		private void itemDeselected(ItemEvent e, Object source) {
			if (e.getStateChange() == ItemEvent.DESELECTED) {
				if (source == switchCase) {
					switchCaseFlag = false;
				} else if (source == dataClass) {
					dataClassFlag = false;
				} else if (source == middleMan) {
					middleManFlag = false;
				} else if (source == longParaList) {
					longParaListFlag = false;
				} else if (source == longMethod) {
					longMethodFlag = false;
				}
				counter--;
			}
		}

		private void itemSelected(ItemEvent e, Object source) {
			if (e.getStateChange() == ItemEvent.SELECTED) {
				if (source == switchCase) {
					switchCaseFlag = true;
				} else if (source == dataClass) {
					dataClassFlag = true;
				} else if (source == middleMan) {
					middleManFlag = true;
				} else if (source == longParaList) {
					longParaListFlag = true;
				} else if (source == longMethod) {
					longMethodFlag = true;
				}
				counter++;
			}
		}
	}

	class refactorButtonListener implements ActionListener {

		public void actionPerformed(ActionEvent e) {
			ArrayList<Boolean> flag = new ArrayList<Boolean>();
			flag.add(switchCaseFlag);
			flag.add(dataClassFlag);
			flag.add(middleManFlag);
			flag.add(longParaListFlag);
			flag.add(longMethodFlag);
			MainTabbedPane mtp = new MainTabbedPane(flag, files);
			frame.setVisible(false);
		}

	}

	public static void main(String[] args) {
		MainGUI main = new MainGUI();
	}
}
```

Class: - MainTabbedPane.java

```
package GUI;

import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.ArrayList;

import javax.swing.BorderFactory;
import javax.swing.BoxLayout;
import javax.swing.ImageIcon;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTabbedPane;

import Main.DesignAnalyzer;
import Refactor.RefactorCode;

public class MainTabbedPane extends JFrame {

	private static JTabbedPane jtp;
	private JPanel switchCase = new JPanel();
	private JPanel dataClass = new JPanel();
	private JPanel middleMan = new JPanel();
	private JPanel longParameterList = new JPanel();
	private JPanel longMethod = new JPanel();
	private JPanel centerPanel;
	private JPanel rightPanel;
	private ImageIcon icon;
	private JButton moveButton;
	private JButton extractButton;
	private JButton inlineButton;
	private static ArrayList<String> switchCaseSmell = new
ArrayList<String>();
	private static ArrayList<String> dataClassSmell = new
ArrayList<String>();
	private static ArrayList<String> longParaListSmell = new
ArrayList<String>();
	private static ArrayList<String> longMethodSmell = new
ArrayList<String>();
	private static ArrayList<String> middleManSmell = new
ArrayList<String>();
	private RefactorListner listner = new RefactorListner();

	public MainTabbedPane(ArrayList<Boolean> flag, ArrayList<String> files)
{
		DesignAnalyzer da = new DesignAnalyzer();
		da.main(files);
		getSmellArrayList(da);
		MainTabbedPane tp = new MainTabbedPane();
```

```
            enableDisableTabs(flag);
            tp.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            tp.setVisible(true);
      }

      private void enableDisableTabs(ArrayList<Boolean> flag) {
            for (int i = 4; i >= 0; i--) {
                  if (flag.get(i) == false) {
                        jtp.setEnabledAt(i, false);
                  } else {
                        jtp.setSelectedIndex(i);
                  }
            }
      }

      private void getSmellArrayList(DesignAnalyzer da) {
            switchCaseSmell.addAll(da.getSwitchCaseSmell());
            dataClassSmell.addAll(da.getDataClassSmell());
            longParaListSmell.addAll(da.getLongParaListSmell());
            longMethodSmell.addAll(da.getLongMethodSmell());
            middleManSmell.addAll(da.getMiddleManSmell());
      }

      public MainTabbedPane() {
            setSize(850, 600);
            setResizable(false);
            setLocationRelativeTo(null);
            setTitle("Java Smell Detector");
            jtp = new JTabbedPane();
            getContentPane().add(jtp);
            createTabs(jtp);
      }

      private void createTabs(JTabbedPane jtp) {
            addTabs(jtp);
            createEachTab(switchCase, switchCaseSmell);
            createEachTab(dataClass, dataClassSmell);
            createEachTab(middleMan, middleManSmell);
            createEachTab(longParameterList, longParaListSmell);
            createEachTab(longMethod, longMethodSmell);
      }

      private void addTabs(JTabbedPane jtp) {
            jtp.add("Switch Case Smell", switchCase);
            jtp.add("Data Class Smell", dataClass);
            jtp.add("Middle Man Smell", middleMan);
            jtp.add("Long Parameter List Smell", longParameterList);
            jtp.add("Long Method Smell", longMethod);
      }

      private void createEachTab(JPanel tab, ArrayList<String> smellList) {
            JPanel result = resultPanel(tab);
            JPanel refactorDesc = refactorDescPanel(tab);
            refactorDesc.setMaximumSize(new Dimension(850, 300));
            refactorDesc.setPreferredSize(new Dimension(850, 250));
            icon = new ImageIcon("src/GUI/bullet_label.gif");
            displayResult(smellList, result);
```

```
            ArrayList<JLabel> list = new ArrayList<JLabel>();
            if (tab == switchCase) {
                  list
                              .add(new JLabel(
                                          "Extract each method to its
individual method and extract them to individual classes.",
                                          icon, JLabel.LEFT));
                  list
                              .add(new JLabel(
                                          "Change access modifier of each
method to \"public\" and rename them with same method name.",
                                          icon, JLabel.LEFT));
                  list
                              .add(new JLabel(
                                          "Extract interface from any one of
the class and create method as member of interface.",
                                          icon, JLabel.LEFT));
                  list.add(new JLabel("Implement interface for rest of the
classes.",
                              icon, JLabel.LEFT));
                  list
                              .add(new JLabel(
                                          "Declare generic Object of the
interface and it's implement it appropriately in each switch case",
                                          icon, JLabel.LEFT));
                  list
                              .add(new JLabel(
                                          "Have a generic return type outside
the switch statement and breakout from individual cases.",
                                          icon, JLabel.LEFT));
                  list.add(new JLabel("Extract switch statement in a
method.", icon,
                              JLabel.LEFT));
                  list
                              .add(new JLabel(
                                          "Create a HashMap consisting each
switch case as key and type of object as value.",
                                          icon, JLabel.LEFT));
                  list
                              .add(new JLabel(
                                          "Clean up the code by applying
Inline refactoring technique and removing unnecessary objects.",
                                          icon, JLabel.LEFT));
                  for (JLabel label : list) {
                        refactorDesc.add(label);
                  }
            } else if (tab == dataClass) {
                  refactorDesc.setLayout(new BorderLayout());
                  createPannelSetLayout();
                  createRefactorButtonAddListener();
                  rightPanel.add(moveButton);
                  rightPanel.add(new JLabel(" "));
                  rightPanel.add(extractButton);
                  centerPanel
                              .add(new JLabel(
                                          "Move method from a class which
uses these getters and setters methods of the Data Class.",
```

```
                                          icon, JLabel.LEFT));
                  centerPanel
                              .add(new JLabel(
                                          "        Make sure to check the
checkbox to keep the method delgate."));
                  centerPanel.add(new JLabel(" "));
                  centerPanel
                              .add(new JLabel(
                                          "If method is a main method or
contains other behaviours Extract Method",
                                          icon, JLabel.LEFT));
                  refactorDesc.add(centerPanel, BorderLayout.CENTER);
                  refactorDesc.add(rightPanel, BorderLayout.EAST);
            } else if (tab == middleMan) {
                  list.add(new JLabel(
                              "Make delegating object a sub class of the
delegate.",
                              icon, JLabel.LEFT));
                  list.add(new JLabel("If method signature conflicts rename
them.",
                              icon, JLabel.LEFT));
                  list.add(new JLabel(
                              "Set delegate field refer to the object
itself.", icon,
                              JLabel.LEFT));
                  list.add(new JLabel("Remove the simple delegation method.",
icon,
                              JLabel.LEFT));
                  list.add(new JLabel(
                              "Replace all other delgation with calls to the
object.",
                              icon, JLabel.LEFT));
                  for (JLabel label : list) {
                        refactorDesc.add(label);
                  }
            } else if (tab == longParameterList) {
                  list
                              .add(new JLabel(
                                          "Create a new parameter for the
whole object from which the data comes.",
                                          icon, JLabel.LEFT));
                  list.add(new JLabel("Compile and test.", icon,
JLabel.LEFT));
                  list
                              .add(new JLabel(
                                          "Determine which parameters should
be obtained from the whole object.",
                                          icon, JLabel.LEFT));
                  list
                              .add(new JLabel(
                                          "Take one parameter and replace
references to it within the method body by invoking an appropriate method on
the whole object parameter.",
                                          icon, JLabel.LEFT));
                  list.add(new JLabel("Delete the parameter.", icon,
JLabel.LEFT));
```

```
                  list.add(new JLabel("Compile and test.", icon,
JLabel.LEFT));
                  list
                              .add(new JLabel(
                                          "Repeat for each parameter that can
be got from the whole object.",
                                          icon, JLabel.LEFT));
                  list
                              .add(new JLabel(
                                          "Remove the code in the calling
method that obtains the deleted parameters.",
                                          icon, JLabel.LEFT));
                  for (JLabel label : list) {
                        refactorDesc.add(label);
                  }
            } else if (tab == longMethod) {
                  refactorDesc.setLayout(new BorderLayout());
                  createPannelSetLayout();
                  createRefactorButtonAddListener();
                  refactorDesc.add(centerPanel, BorderLayout.CENTER);
                  refactorDesc.add(rightPanel, BorderLayout.EAST);
                  centerPanel.add(new JLabel(
                              "Divide the code depending upon the code
dependency.",
                              icon, JLabel.LEFT));
                  centerPanel.add(new JLabel(" "));
                  centerPanel
                              .add(new JLabel(
                                          "Select each division and extract
each division into individual methods by clicking on the Extract Method
Button.",
                                          icon, JLabel.LEFT));
                  rightPanel.add(new JLabel(" "));
                  rightPanel.add(new JLabel(" "));
                  rightPanel.add(extractButton);
            }

      }

      private void createPannelSetLayout() {
            centerPanel = new JPanel();
            rightPanel = new JPanel();
            centerPanel.setLayout(new BoxLayout(centerPanel,
BoxLayout.Y_AXIS));
            rightPanel.setLayout(new BoxLayout(rightPanel,
BoxLayout.Y_AXIS));
      }

      private void createRefactorButtonAddListener() {
            moveButton = new JButton(" Move Method ");
            extractButton = new JButton("Extract Method");
            inlineButton = new JButton("  Inline Method  ");
            moveButton.addActionListener(listner);
            extractButton.addActionListener(listner);
            inlineButton.addActionListener(listner);
      }
```

```
	private void displayResult(ArrayList<String> smellList, JPanel result)
{
		if (smellList != null) {
			for (int i = 0; i < smellList.size(); i++) {
				JLabel label = new JLabel(smellList.get(i), icon,
JLabel.LEFT);
				result.add(label);
			}
		} else {
			JLabel label = new JLabel(
					"No Bad Smell detected in any of the files.",
icon,
					JLabel.LEFT);
			result.add(label);
		}
	}

	private JPanel refactorDescPanel(JPanel tab) {
		JPanel refactorDesc = new JPanel();
		tab.add(refactorDesc);

	refactorDesc.setBorder(BorderFactory.createTitledBorder("Refactoring"))
;
		refactorDesc.setLayout(new GridLayout(0, 1));
		return refactorDesc;
	}

	private JPanel resultPanel(JPanel tab) {
		tab.setLayout(new BoxLayout(tab, BoxLayout.Y_AXIS));
		JPanel result = new JPanel();
		result.setLayout(new GridLayout(0, 1, 1, 5));
		JScrollPane resultScroller = new JScrollPane();
		resultScroller.setViewportView(result);
		tab.add(resultScroller);
		result.setBorder(BorderFactory.createTitledBorder("Results"));
		return result;
	}

	class RefactorListner implements ActionListener {
		public void actionPerformed(ActionEvent e) {
			RefactorCode refactor = new RefactorCode();
			if (e.getActionCommand() == " Move Method ") {
				refactor.swapScreen();
				refactor.callMoveMethod();
			} else if (e.getActionCommand() == "Extract Method") {
				refactor.swapScreen();
				refactor.callExtractMethod();
			}
		}
	}
}
```

Package: - Main

Class: - DesignAnalyzer.java

```
package Main;

import java.io.File;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Scanner;
import java.util.Set;

import org.eclipse.jdt.core.dom.AST;
import org.eclipse.jdt.core.dom.ASTParser;
import org.eclipse.jdt.core.dom.CompilationUnit;

import CodeAnalyzer.CodeAnalyzerMethodDeclarationLister;
import CodeAnalyzer.FieldDeclarationLister;
import DataClassBadSmell.MethodDeclarationBlock;
import DataClassBadSmell.TypeDeclarationStatementTest;
import LongMethodBadSmell.LongMethodMethodDeclaration;
import LongParameterListBadSmell.LongParameterListMethodDeclaration;
import MiddleManBadSmell.MiddleManMethodDeclaration;
import SwitchCaseBadSmell.CountIfStatement;
import SwitchCaseBadSmell.CountSwitchStatement;

public class DesignAnalyzer {

	private ArrayList<String> globalVariableList = new ArrayList<String>();
	private ArrayList<String> globalMethodNameList = new
ArrayList<String>();

	private static ArrayList<String> switchCaseSmell = new
ArrayList<String>();
	private static ArrayList<String> dataClassSmell = new
ArrayList<String>();
	private static ArrayList<String> longParaListSmell = new
ArrayList<String>();
	private static ArrayList<String> longMethodSmell = new
ArrayList<String>();
	private static ArrayList<String> middleManSmell = new
ArrayList<String>();

	private String smell;

	public ArrayList<String> getMiddleManSmell() {
		return middleManSmell;
	}

	public ArrayList<String> getLongMethodSmell() {
		return longMethodSmell;
	}

	public ArrayList<String> getSwitchCaseSmell() {
		return switchCaseSmell;
	}
```

```
	public ArrayList<String> getDataClassSmell() {
		return dataClassSmell;
	}

	public ArrayList<String> getLongParaListSmell() {
		return longParaListSmell;
	}

	private char[] getChars(String fileName) {
		char[] result = null;
		try {
			File file = new File(fileName);
			Scanner scanner = new Scanner(file);
			String contents = "";
			while (scanner.hasNext()) {
				String nextLine = scanner.nextLine();
				if (nextLine.contains("//")) {
					int beginIndex = nextLine.indexOf("//", 0);
					String comment =
nextLine.substring(beginIndex);
					nextLine = nextLine.replaceAll(comment, "");
					if (comment != null)
						contents += nextLine;
				} else {
					contents += nextLine;
				}
			}
			result = contents.toCharArray();
		} catch (Exception e) {
			System.out.println(e);
		}
		return result;
	}

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

	private void countSwitchCase(CompilationUnit cu, String fileName) {
		CountSwitchStatement visitor = new CountSwitchStatement();
		cu.accept(visitor);
		smell = visitor.getMessage();
		fileName = fileName.substring(fileName.lastIndexOf("\\") + 1);
		System.out.println("Test: " + fileName);
		if (smell != null) {
			switchCaseSmell.add("Class Name: " + fileName + "
"
					+ smell);
		}
		CountIfStatement visitor1 = new CountIfStatement();
		cu.accept(visitor1);
		ArrayList<Integer> ifCounter = visitor1.getCounter();
		if (ifCounter != null) {
```

```
                  int i = 0;
                  while (i < ifCounter.size()) {
                        int temp = ifCounter.get(i);
                        if (ifCounter.get(i) > 2) {
                              switchCaseSmell
                                          .add("Class Name: "
                                                      + fileName
                                                      + "          contains
if else block of more than 3 cases.");
                        }
                        if (temp == 1) {
                              i = i + 1;
                        } else {
                              i = i + temp;
                        }
                  }
            }
      }

      private void longParameterList(String fileName, CompilationUnit cu) {
            ArrayList<String> smell = new ArrayList<String>();
            fileName = fileName.substring(fileName.lastIndexOf("\\") + 1);
            LongParameterListMethodDeclaration visitor = new
LongParameterListMethodDeclaration();
            cu.accept(visitor);
            smell.addAll(visitor.getExpression());
            if (smell.size() != 0) {
                  for (String temp : smell) {
                        longParaListSmell.add("Class Name: " + fileName + "
"
                                    + temp);
                  }
            }
      }

      @SuppressWarnings("unchecked")
      private void dataClassBadSmell(String fileName, CompilationUnit cu) {
            fileName = fileName.substring(fileName.lastIndexOf("\\") + 1);
            HashMap<String, Integer> methodNameLength = new HashMap<String,
Integer>();

            MethodDeclarationBlock visitor = new MethodDeclarationBlock();
            cu.accept(visitor);

            methodNameLength = visitor.getMethodNameLength();
            Set set = methodNameLength.entrySet();
            Iterator i = set.iterator();
            while (i.hasNext()) {
                  Map.Entry me = (Map.Entry) i.next();
                  if (Integer.parseInt(me.getValue().toString()) > 0) {
                        dataClassSmell.remove(fileName);
                  }
            }

            TypeDeclarationStatementTest visitor1 = new
TypeDeclarationStatementTest();
            cu.accept(visitor1);
```

```
            if (visitor1.getInterfaceFlag())
                  dataClassSmell.remove(fileName);

            if (visitor1.getAbstractFlag())
                  dataClassSmell.remove(fileName);
      }

      private void longMethodBadSmell(String fileName, CompilationUnit cu) {
            LongMethodMethodDeclaration visitor = new
LongMethodMethodDeclaration();
            cu.accept(visitor);
            smell = visitor.getSmell();
            fileName = fileName.substring(fileName.lastIndexOf("\\") + 1);
            if (smell != null) {
                  longMethodSmell.add("Class Name: " + fileName + "
"
                              + smell);
            }
      }

      private void middleMan(String fileName, CompilationUnit cu) {
            MiddleManMethodDeclaration visitor = new
MiddleManMethodDeclaration(
                        globalMethodNameList);
            cu.accept(visitor);
            smell = visitor.getSmell();
            fileName = fileName.substring(fileName.lastIndexOf("\\") + 1);
            if (smell != null) {
                  middleManSmell
                              .add("Class Name: " + fileName + "          " +
smell);
            }
      }

      private void parseCode(CompilationUnit cu) {

            FieldDeclarationLister visitor = new FieldDeclarationLister();
            cu.accept(visitor);
            globalVariableList.addAll(visitor.getGlobalVariableList());
            CodeAnalyzerMethodDeclarationLister visitor1 = new
CodeAnalyzerMethodDeclarationLister();
            cu.accept(visitor1);
            globalMethodNameList.addAll(visitor1.getGlobalMethodNameList());
      }

      public void main(ArrayList<String> files) {
            DesignAnalyzer da = new DesignAnalyzer();
            CompilationUnit cu;
            for (String fileName : files) {
                  cu = da.parse(fileName);
                  da.parseCode(cu);
                  fileName = fileName.substring(fileName.lastIndexOf("\\") +
1);
                  dataClassSmell.add(fileName);
            }
            for (String fileName : files) {
```

```
                    cu = da.parse(fileName);
                    da.countSwitchCase(cu, fileName);
                    da.dataClassBadSmell(fileName, cu);
                    da.longParameterList(fileName, cu);
                    da.longMethodBadSmell(fileName, cu);
                    da.middleMan(fileName, cu);
              }
        }
}
```

Package: - CodeAnalyzer

Class: - CodeAnalyzerMethodDeclarationLister.java

```
package CodeAnalyzer;

import java.util.ArrayList;

import org.eclipse.jdt.core.dom.ASTVisitor;
import org.eclipse.jdt.core.dom.MethodDeclaration;

public class CodeAnalyzerMethodDeclarationLister extends ASTVisitor {

      public ArrayList<String> globalMethodNameList = new
ArrayList<String>();

      public ArrayList<String> getGlobalMethodNameList() {
            return globalMethodNameList;
      }

      public boolean visit(MethodDeclaration node) {
            globalMethodNameList.add(node.getName().toString());
            return true;
      }
}
```

Class: - FieldDeclarationLister.java

```
package CodeAnalyzer;

import java.util.ArrayList;

import org.eclipse.jdt.core.dom.ASTVisitor;
import org.eclipse.jdt.core.dom.FieldDeclaration;

public class FieldDeclarationLister extends ASTVisitor {

      private ArrayList<String> globalVariableList = new ArrayList<String>();

      public ArrayList<String> getGlobalVariableList() {
            return globalVariableList;
      }
```

```
	public boolean visit(FieldDeclaration node) {
		VariableDeclarationFragmentLister visitor = new
VariableDeclarationFragmentLister();
		node.accept(visitor);

		globalVariableList.add(visitor.getGlobalVariable());
		return true;
	}
}
```

Class: - VariableDeclarationFragmentLister.java

```
package CodeAnalyzer;

import org.eclipse.jdt.core.dom.ASTVisitor;
import org.eclipse.jdt.core.dom.VariableDeclarationFragment;

public class VariableDeclarationFragmentLister extends ASTVisitor {

	String globalVariable;

	public String getGlobalVariable() {
		return globalVariable;
	}

	public boolean visit(VariableDeclarationFragment node) {
		globalVariable = node.getName().toString();
		return true;
	}

}
```

Package: - Refactor

Class: - RefactorCode.java

```
package Refactor;

import java.awt.AWTException;
import java.awt.Robot;
import java.awt.event.KeyEvent;

public class RefactorCode {

	public void callExtractMethod() {
		try {
			Robot robot = new Robot();
			robot.delay(300);
			robot.keyPress(KeyEvent.VK_ALT);
			robot.keyPress(KeyEvent.VK_SHIFT);
			robot.keyPress(KeyEvent.VK_M);
			robot.keyRelease(KeyEvent.VK_ALT);
```

```
                robot.keyRelease(KeyEvent.VK_SHIFT);
                robot.keyRelease(KeyEvent.VK_M);
            } catch (AWTException e) {
                e.printStackTrace();
            }
      }

      public void callMoveMethod() {
            try {
                Robot robot = new Robot();
                robot.delay(300);
                robot.keyPress(KeyEvent.VK_ALT);
                robot.keyPress(KeyEvent.VK_SHIFT);
                robot.keyPress(KeyEvent.VK_V);
                robot.keyRelease(KeyEvent.VK_ALT);
                robot.keyRelease(KeyEvent.VK_SHIFT);
                robot.keyRelease(KeyEvent.VK_V);
            } catch (AWTException e) {
                e.printStackTrace();
            }
      }

      public void swapScreen() {
            Robot robot;
            try {
                robot = new Robot();
                robot.delay(300);
                robot.keyPress(KeyEvent.VK_ALT);
                robot.keyPress(KeyEvent.VK_TAB);
                robot.delay(2000);
                robot.keyRelease(KeyEvent.VK_ALT);
                robot.keyRelease(KeyEvent.VK_TAB);
            } catch (AWTException e) {
                e.printStackTrace();
            }

      }
}
```

Package: - DataClassBadSmell

Class: - MethodDeclarationBlock.java

```
package DataClassBadSmell;

import java.util.HashMap;
import org.eclipse.jdt.core.dom.ASTVisitor;
import org.eclipse.jdt.core.dom.MethodDeclaration;

public class MethodDeclarationBlock extends ASTVisitor {
      private HashMap<String, Integer> methodNameLength = new HashMap<String,
Integer>();
      private int counter = 1;

      public HashMap<String, Integer> getMethodNameLength() {
            return methodNameLength;
```

```
	}

	public boolean visit(MethodDeclaration node) {
		BlockLister visitor = new BlockLister();
		node.accept(visitor);
		methodNameLength.put(counter + ":" + node.getName().toString(),
visitor.getNumberOfStatements());
		counter++;
		return true;
	}

}
```

Class: - BlockLister.java

```
package DataClassBadSmell;

import org.eclipse.jdt.core.dom.ASTVisitor;
import org.eclipse.jdt.core.dom.Block;

public class BlockLister extends ASTVisitor{

	private int numberOfStatements;

	public int getNumberOfStatements(){
		return numberOfStatements;
	}

	public boolean visit(Block node) {
		numberOfStatements = node.statements().size();

		ReturnStatementLister visitor = new ReturnStatementLister();
		node.accept(visitor);
		numberOfStatements = numberOfStatements -
visitor.getReturnStatementCounter();

		VariableDeclerationStatementLister visitor1 = new
VariableDeclerationStatementLister();
		node.accept(visitor1);
		numberOfStatements = numberOfStatements -
visitor1.getVariableDeclarationStatementCounter();

		ExpressionStatementLister visitor2 = new
ExpressionStatementLister();
		node.accept(visitor2);
		numberOfStatements = numberOfStatements -
visitor2.getAssignmentListerCounter();

		return true;
	}
}
```

Class: - ReturnStatementLister.java

```
package DataClassBadSmell;

import org.eclipse.jdt.core.dom.ASTVisitor;
import org.eclipse.jdt.core.dom.ReturnStatement;

public class ReturnStatementLister extends ASTVisitor {

	private int returnStatementCounter = 0;

	public int getReturnStatementCounter() {
		return returnStatementCounter;
	}

	public boolean visit(ReturnStatement node) {
		returnStatementCounter = 1;
		return true;
	}
}
```

Class: - VariableDeclarationStatementLister.java

```
package DataClassBadSmell;

import org.eclipse.jdt.core.dom.ASTVisitor;
import org.eclipse.jdt.core.dom.VariableDeclarationStatement;

public class VariableDeclerationStatementLister extends ASTVisitor {

	private int variableDeclarationStatementCounter = 0;

	public int getVariableDeclarationStatementCounter() {
		return variableDeclarationStatementCounter;
	}

	public boolean visit(VariableDeclarationStatement node) {
		variableDeclarationStatementCounter++;
		return true;
	}

}
```

Class: - ExpressionStatementLister.java

```
package DataClassBadSmell;

import org.eclipse.jdt.core.dom.ASTVisitor;
import org.eclipse.jdt.core.dom.ExpressionStatement;
```

```
public class ExpressionStatementLister extends ASTVisitor {

	private int assignmentListerCounter = 0;

	public int getAssignmentListerCounter() {
		return assignmentListerCounter;
	}

	public boolean visit(ExpressionStatement node) {
		AssignmentLister visitor = new AssignmentLister();
		node.accept(visitor);
		assignmentListerCounter = visitor.getAssignmentListerCounter();

		return true;
	}

}
```

Class: - AssignmentLister.java

```
package DataClassBadSmell;

import org.eclipse.jdt.core.dom.ASTVisitor;
import org.eclipse.jdt.core.dom.Assignment;

public class AssignmentLister extends ASTVisitor {

	private int assignmentListerCounter = 0;

	public int getAssignmentListerCounter() {
		return assignmentListerCounter;
	}

	public boolean visit(Assignment node) {
		assignmentListerCounter++;
		return true;
	}
}
```

Class: - TypeDeclarationStatement.java

```
package DataClassBadSmell;

import java.util.List;

import org.eclipse.jdt.core.dom.ASTVisitor;
import org.eclipse.jdt.core.dom.TypeDeclaration;

public class TypeDeclarationStatement extends ASTVisitor {

	private boolean interfaceFlag = false;
	private boolean abstractFlag = false;
```

```
	public boolean getInterfaceFlag() {
		return interfaceFlag;
	}

	public boolean getAbstractFlag() {
		return abstractFlag;
	}

	public boolean visit(TypeDeclaration node) {
		abstractFlag = false;
		List temp = node.modifiers();
		for (int i = 0; i < temp.size(); i++) {
			if (temp.get(i).toString().equalsIgnoreCase("abstract"))
				abstractFlag = true;
		}
		if (node.isInterface())
			interfaceFlag = true;
		return true;
	}

}
```

Package: - LongMethodBadSmell

Class: - LongMethodMethodDeclaration.java

```
package LongMethodBadSmell;

import org.eclipse.jdt.core.dom.ASTVisitor;
import org.eclipse.jdt.core.dom.MethodDeclaration;

public class LongMethodMethodDeclaration extends ASTVisitor {

	private int numberOfStatements;
	private String smell;

	public String getSmell() {
		return smell;
	}

	public boolean visit(MethodDeclaration node) {
		String[] methodBody = null;

		LongMethodBlock visitor = new LongMethodBlock();
		node.accept(visitor);
		numberOfStatements = visitor.getNumberOfStatements();
		methodBody = node.toString().split("\n");

		numberOfStatements = methodBody.length - 2;

		if (numberOfStatements > 15) {
			smell = "Method Name: " + node.getName().toString()
					+ "          Number of Lines: " +
numberOfStatements;
```

```
            }
            return true;
      }

}
```

Class: - LongMethodBlock.java

```
package LongMethodBadSmell;

import org.eclipse.jdt.core.dom.ASTVisitor;
import org.eclipse.jdt.core.dom.Block;

public class LongMethodBlock extends ASTVisitor {

      private int numberOfStatements;

      public int getNumberOfStatements() {
            return numberOfStatements;
      }

      public boolean visit(Block node) {
            LongMethodBlock visitor = new LongMethodBlock();
            int temp = visitor.getNumberOfStatements();
            numberOfStatements = node.statements().size() + temp;
            return true;
      }
}
```

Package: - LongParamterListBadSmell

Class: - LongParameterListMethodDeclaration.java

```
package LongParameterListBadSmell;

import java.util.ArrayList;

import org.eclipse.jdt.core.dom.ASTVisitor;
import org.eclipse.jdt.core.dom.MethodDeclaration;

public class LongParameterListMethodDeclaration extends ASTVisitor {

      private ArrayList<String> longParaList = new ArrayList<String>();
      private int numberOfParameter = 0;
      private String methodName = new String();
      private String methodBody;
      private String expression;

      public ArrayList<String> getExpression() {
            return longParaList;
      }

      public boolean visit(MethodDeclaration node) {
```

```
            LongParameterListSingleVariable visitor = new
LongParameterListSingleVariable();
            node.accept(visitor);
            numberOfParameter = visitor.getcounter();
            methodName = node.getName().toString();

            longMethodParaCheck(node, visitor);

            return true;
      }

      private void longMethodParaCheck(MethodDeclaration node,
                  LongParameterListSingleVariable visitor) {
            if (numberOfParameter > 3) {
                  methodBody = node.toString();
                  visitor.getParameters();
                  String[] lines = methodBody.split("\n");
                  for (String temp : lines) {
                        if (temp.contains(methodName) &&
!temp.startsWith("/")
                                    && temp.contains("(") &&
temp.contains(")")) {
                              int startindex = temp.indexOf("(") + 1;
                              int endindex = temp.indexOf(")", startindex);
                              expression = temp.substring(startindex,
endindex);
                              String[] te = expression.split(" ");
                              int a = te.length;
                              a = a / 2;
                              if (a > 2) {
                                    expression = "Method Name: " + methodName
                                                + "          Parameters: " +
expression;
                                    longParaList.add(expression);
                              }
                        }
                  }
            }
      }
}
```

Class: - LongParameterListSingleVariable.java

```
package LongParameterListBadSmell;

import java.util.ArrayList;

import org.eclipse.jdt.core.dom.ASTVisitor;
import org.eclipse.jdt.core.dom.SingleVariableDeclaration;

public class LongParameterListSingleVariable extends ASTVisitor{

      private int counter = 0;
      private ArrayList<String> parameters = new ArrayList<String>();
```

```
	public boolean visit(SingleVariableDeclaration node) {
		parameters.add(node.getName().toString());
		counter++;
		return true;
	}

	public ArrayList<String> getParameters() {
		return parameters;
	}

	public int getcounter() {
		return counter;
	}

}
```

Package: - MiddleManBadSmell

Class: - MiddleManMethodDeclaration.java

```
package MiddleManBadSmell;

import java.util.ArrayList;

import org.eclipse.jdt.core.dom.ASTVisitor;
import org.eclipse.jdt.core.dom.MethodDeclaration;

public class MiddleManMethodDeclaration extends ASTVisitor {

	private ArrayList<String> _globalMethodNameList;
	private boolean flag;
	private String smell;

	public MiddleManMethodDeclaration(ArrayList<String>
globalMethodNameList) {
		_globalMethodNameList = globalMethodNameList;
	}

	public String getSmell() {
		return smell;
	}

	public boolean visit(MethodDeclaration node) {
		MiddleManBlock visitor = new
MiddleManBlock(_globalMethodNameList);
		node.accept(visitor);
		flag = visitor.getFlag();
		if (flag == true) {
			smell = "Method Name: " + node.getName().toString();
			// System.out.println(node.getName().toString() +
			// " method consistes of a Middle Chain Bad Smell");
		}
		return true;
```

```
	}

}
```

Class: - MiddleManBlock.java

```
package MiddleManBadSmell;

import java.util.ArrayList;

import org.eclipse.jdt.core.dom.ASTVisitor;
import org.eclipse.jdt.core.dom.Block;

public class MiddleManBlock extends ASTVisitor {

	private ArrayList<String> _globalMethodNameList;
	private boolean flag;

	public MiddleManBlock(ArrayList<String> globalMethodNameList) {
		_globalMethodNameList = globalMethodNameList;
	}

	public boolean getFlag() {
		return flag;
	}

	public boolean visit(Block node) {
		int numberOfStatements = node.statements().size();

		if (numberOfStatements == 1) {
			MiddleManReturnStatement visitor = new
MiddleManReturnStatement(
					_globalMethodNameList);
			node.accept(visitor);
			flag = visitor.getFlag();
		}
		return true;
	}

}
```

Class: - MiddleManReturnStatement.java

```
package MiddleManBadSmell;

import java.util.ArrayList;

import org.eclipse.jdt.core.dom.ASTVisitor;
import org.eclipse.jdt.core.dom.ReturnStatement;

public class MiddleManReturnStatement extends ASTVisitor {
```

```
	private boolean flag;
	private ArrayList<String> _globalMethodNameList;

	public MiddleManReturnStatement(ArrayList<String> globalMethodNameList)
{
		_globalMethodNameList = globalMethodNameList;
	}

	public boolean getFlag() {
		return flag;
	}

	public boolean visit(ReturnStatement node) {
		MiddleManMethodInvocation visitor = new
MiddleManMethodInvocation(
				_globalMethodNameList);
		node.accept(visitor);
		flag = visitor.getFlag();
		return true;
	}

}
```

Class: - MiddleManMethodInvocation.java

```
package MiddleManBadSmell;

import java.util.ArrayList;

import org.eclipse.jdt.core.dom.ASTVisitor;
import org.eclipse.jdt.core.dom.MethodInvocation;

public class MiddleManMethodInvocation extends ASTVisitor {

	private boolean flag = false;
	private ArrayList<String> _globalMethodNameList;

	public MiddleManMethodInvocation(ArrayList<String>
globalMethodNameList) {
		_globalMethodNameList = globalMethodNameList;
	}

	public boolean getFlag() {
		return flag;
	}

	public boolean visit(MethodInvocation node) {
		if (_globalMethodNameList.contains(node.getName().toString())) {
			flag = true;
		}
		return true;
	}

}
```

Package: - SwitchCaseBadSmell

Class: - CountSwitchStatement.java

```
package SwitchCaseBadSmell;

import org.eclipse.jdt.core.dom.ASTVisitor;
import org.eclipse.jdt.core.dom.SwitchStatement;

public class CountSwitchStatement extends ASTVisitor {

	private String message = null;

	public boolean visit(SwitchStatement node) {
		SimpleNameLister visitor1 = new SimpleNameLister();
		node.accept(visitor1);
		CountSwitchCases visitor = new CountSwitchCases();
		node.accept(visitor);
		System.out.println("Test: " + visitor.getLoopCount());
		if (visitor.getLoopCount() > 3) {
			message = "Swicth Case Name: " + visitor1.getSimpleName()
					+ "          " + "# of switch cases: "
					+ visitor.getLoopCount();
		}

		return true;
	}

	public String getMessage() {
		return message;
	}

}
```

Class: - CountSwitchCases.java

```
package SwitchCaseBadSmell;

import org.eclipse.jdt.core.dom.ASTVisitor;
import org.eclipse.jdt.core.dom.SwitchCase;

public class CountSwitchCases extends ASTVisitor {

	private int switchCases = 0;

	public int getLoopCount() {
		return switchCases;
	}

	public boolean visit(SwitchCase node) {
		switchCases++;
		return true;
```

```
	}

}
```

Class: - CountIfStatement.java

```
package SwitchCaseBadSmell;

import java.util.ArrayList;

import org.eclipse.jdt.core.dom.ASTVisitor;
import org.eclipse.jdt.core.dom.IfStatement;

public class CountIfStatement extends ASTVisitor {

	private ArrayList<Integer> counterArray = new ArrayList<Integer>();
	private int counter = 1;

	public boolean visit(IfStatement node) {
		if (node.getElseStatement() != null) {
			CountElseStatement visitor = new CountElseStatement();
			visitor.setCounter();
			node.accept(visitor);
			counter = visitor.getCounter();
		}
		counterArray.add(counter);
		counter = 1;
		return true;
	}

	public ArrayList<Integer> getCounter() {
		return counterArray;
	}
}
```

Class: - CountElseStatement.java

```
package SwitchCaseBadSmell;

import org.eclipse.jdt.core.dom.ASTVisitor;
import org.eclipse.jdt.core.dom.IfStatement;

public class CountElseStatement extends ASTVisitor {

	int counter;

	public boolean visit(IfStatement node) {
		counter++;
		return true;
	}

	public int getCounter() {
		return counter;
	}
```

```
	public void setCounter() {
		counter = 0;
	}

}
```

Class: - SimpleNameLister.java

```
package SwitchCaseBadSmell;

import java.util.ArrayList;

import org.eclipse.jdt.core.dom.ASTVisitor;
import org.eclipse.jdt.core.dom.SimpleName;

public class SimpleNameLister extends ASTVisitor {

	ArrayList<String> name = new ArrayList<String>();

	public String getSimpleName() {
		return name.get(0);
	}

	public boolean visit(SimpleName node) {
		name.add(node.getIdentifier().toString());
		return true;
	}
}
```