JSMELL: A BAD SMELL DETECTION TOOL FOR JAVA SYSTEMS

A THESIS

Presented to the Department of

Computer Engineering and Computer Science

In Partial Fulfillment

of the Requirements for the Degree

Master of Computer Science

Committee Members:

Michael Hoffman, Ph.D. (Chair)
Arthur Gittleman, Ph.D.
Kenneth James, Ph.D.

College Designee:

Kenneth James, Ph.D.

By Naveen Roperia

B.E., 2006, Maharishi Dayanand University

May, 2009

ABSTRACT

JSMELL: BAD SMELL DETECTION TOOL

FOR JAVA SYSTEMS

By

Naveen Roperia

May 2009

Bad-Smells in software are patterns of poor design and bad programming. These patterns can be removed from the software system by using refactoring techniques which improve the readability, maintainability and comprehension of the software system. This thesis describes an automated bad smell detection process in the Java source-code by static analysis of code-structure.

A prototype tool, JSmell, has been developed to detect the bad smell. Once the bad smell has been detected by the tool, various refactoring techniques are suggested to eliminate the smell. The developers can then focus on the elimination of poor design constructs. The analysis and detection process involves software-metrics evaluation and parsing technique, respectively. The success rate for bad smell detection process is 85%-90% when tested against seven different test cases for a single version of system. JSmell also represents the structural analysis of source code in terms of classes, methods and data fields which facilitates the developers in understanding the high level system architecture.

Keywords: Bad-smell, Code Smell, Refactoring, Object-Oriented Design, Software Metrics, Taxonomy, Design defects, Source-code, Software Maintenance, Reverse Engineering, Lexer, Parser

ACKNOWLEDGMENTS

I would like to thank the God for giving me the strength and patience to complete this thesis. While researching on this thesis I have worked with a great number of people whose contribution in assorted ways to the research and the making of the thesis deserved special mention.

It is a pleasure to convey my gratitude to them all in my humble acknowledgment. I take this opportunity to record my sincerest gratitude to my advisor Dr. Michael Hoffman, for his supervision, advice, and guidance from the very early stage of this research as well as for giving me extraordinary experiences through his valuable mentoring at every possible step. He was a constant oasis of ideas and passions which exceptionally inspired and enriched my growth as a student and as a researcher while doing this thesis. I am indebted to him more than he knows.

I gratefully thank Dr. Kenneth James and Dr. Arthur Gittleman for being a part of my thesis committee, and providing me with necessary suggestions and valuable comments to make this thesis stronger. I am extremely thankful to all those people, whom I had any sort of conversation relating to topics of this thesis. The valuable exchange of technical and non-technical thoughts has helped me to a great extent in my path to complete this research.

My parents deserve special mention for their inseparable support and prayers. I would like to present this work to my loving parents, Satbir Roperia and Bimla

Roperia, who believed in me and my goals, and who continues to be a source of inspiration in my life.

In the end I would like to give special thanks to my friend Deepti Kundu who supported and encouraged me to pursue and finish this degree in the possible time limits.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

TABLE Page

LIST OF FIGURES

FIGURE Page

FIGURE Page

CHAPTER 1

INTRODUCTION

The quality of a software system tends to degrade as the system undergoes transformation in its life cycle. As a result, this change may introduce various undesired design-flaws in the system. During the source-code inspection, developers lay more emphasis in finding and fixing bugs rather than improving the system design. Bad smells are design flaws which are introduced in the system during this maintenance life cycle. "Bad smell" are also known as "code smell" and the terms are used interchangeably in this thesis.

Martin Fowler and Kent Beck [1] listed 22 bad smells in source code of Object-Oriented systems. This thesis describes the detection of 11 bad smells. The bad smell detection process has been defined as a pre-work to the refactoring process. Refactoring applied at any level depends on the type of design defect found in the system and has a direct influence on the software maintenance cost [2]. A catalog of 72 possible refactorings has been represented and classified into five sections. It also states how, when and where to apply these refactoring(s). For example, to apply refactorings for data-class bad smell, we need to apply *encapsulate fields* first and then apply *move method* for the client class. A consistent technique is required to detect the accurate code smell type so that corresponding refactoring can be applied to the source code.

In order to be able to improve the design process, improvement efforts must be made measurable [3]. Software metrics provides a very promising way to achieve this goal. This method of smell detection has resolved the problem to a satisfactory level. Researchers and software practitioners emphasize the implementation of automated tools to detect bad smells in the system and these tools are quite appealing as the size of the system increases.

In this research work, the focus is on the code smell detection process. To achieve this goal, a prototype tool, JSmell, has been developed that applies a set of software metrics on Java system and the results are interpreted as design defects in the system. ANTLR, a parser generator, generates code that is used for parsing the input code. The bad smell symptoms are captured and highlighted in the JSmell. For each smell, the parsing and metrics analysis techniques are used to detect the code smell instance from the source code and highlight it in JSmell. The detection process provides additional evidence by locating these bad smells (in location such as class, method) in the source code.

The main benefit of using the software metrics approach is that the proposed results can be easily verified by manual detection. The proposed refactoring in the tool helps in visualizing the problem and solution for the type of design defect detected in the system.

A higher level view of system architecture is as shown in Figure 1.

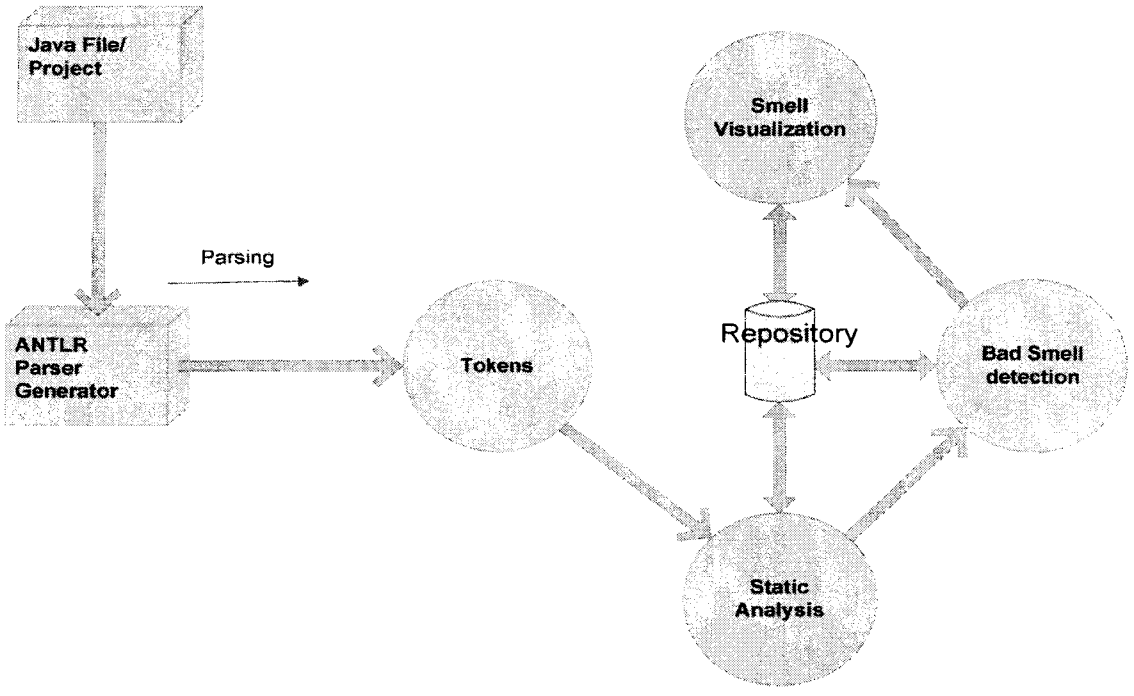

FIGURE 1. High Level System Architecture.

The work done by various researchers and the tools available for smell detection are discussed in chapter 3. Chapter 4 briefly explains the functionality provided by the JSmell. Analysis and Detection approaches applied in the prototype tool will be discussed in chapters 5 and 6. Chapter 7 describes the input validation and result verification. Chapter 8 includes the conclusion and the related future work to be done in this field. In the next chapter, we will discuss a detailed background and classification of the bad smells

CHAPTER 2

BAD SMELLS OVERVIEW AND REFACTORING PROCESS

The concept of bad smell becomes critical while modifying complex software. The presence of bad smells serves as an indicator to the developers of the need to improve software design by refactoring. The quality attribute of a software largely depends on its design [3]. Refactoring eases the code readability and maintainability. Fowler and Beck [1] originated the idea to pursue the code smell detection for refactoring purpose. The research work in regard to code smell and their classification has been specified primarily in [3, 4]. Other researchers and their aspect of bad smell detection are also referred and cited in this thesis. The resource for code smell can be found in "*Refactoring: Improving the Design of Existing Code [1]*", by Martin Fowler with contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts. This book has served as primary resource for referral in this thesis.

This chapter first introduces the classification and definition of bad smells and then discusses how refactoring helps in removing code smell from the system. The code smells are represented in groups of related behaviors [3]. Another approach to classify code smells is by location of the smell in the system. In this taxonomy, we classify the bad smells as found within the class, such as *Long method, Long Parameter List, Comment, duplicate code,* or outside the class, such as *Data Class,*

Data Clumps, Primitive Obsession. The taxonomy followed in this chapter is based on the listing provided in [3].

Bad Smell Classification

Since several smells are closely related, they are organized into several categories. The taxonomy discussed in this chapter provides a relevant name to each category based on the relationship amongst the bad smells in each group. All the smells under these categories and those that do not fit any of these groups are discussed below.

Bloaters

A bloater is a group that represents something in code which has grown so large that it cannot be effectively handled [3]. This group includes the following bad smells.

1. *Long Method*: A method with large number of lines, performing more than one action is referred to as a *Long Method.* A method greater than 20 LOC can be considered as a Long Method [5].

2. *Large Class*: A Large Class is one which is trying to do too much, that is, when a class has too many responsibilities and thence large number of instance variables and methods in the system, it is termed as Large Class.

3. *Primitive Obsession*: Primitive data types are the building blocks in a programming language. When these data types are used excessively rather than representing a group of such data type as a class, it is referred to as Primitive Obsession.

4. *Long Parameter List*: When the number of parameters passed to a method is more than actually required for the functionality of the method, such an instance in the system highlights the existence of Long Parameter List.

5. *Data Clumps*: Existence of similar types of data items together in a number of places such as fields in classes, parameters in many methods is called as Data Clumps.

Object-Oriented Abusers

This category of smell includes the cases where solution does not fully exploit the possibilities of Object-Oriented design. These include:

1. *Switch Statements*: Existence of the same switch statement scattered throughout the program in the source code is identified as a bad smell termed as Switch Statements. It causes code duplication in the system. Polymorphism provides a better alternative to be applied on such switch type code in object-oriented programming.

2. *Temporary Field*: When an instance variable is used only in certain circumstances and is difficult to track, it represents the existence of Temporary Field.

3. *Refused Bequest*: When sub-classes inherit unnecessary data and methods from their parent classes, it indicates the presence of Refused Bequest bad smell in the source code.

4. *Alternate Class with Different Interfaces*: Alternate Class with Different Interfaces bad smell is said to be existing when there is an instance of methods with similar behavior appearing with different signature in the system.

5. *Parallel Inheritance Hierarchies*: Parallel Inheritance Hierarchies exists when creating subclass of a class compels us to make subclass of another.

Change Preventers

The Change Preventers is the group of bad smells that hinder the process of software modification. This group includes the following bad smells:

1. *Divergent Changes*: In this smell one class changes in different ways for different reasons. To illustrate further, whenever we intend to provide an additional functionality in a class, we have to make changes in multiple methods in the class. Again if need for such modification occurs time and again, the methods are changed eventually every time. This is not a good instance of Object oriented programming. The expected behavior from the system is to have a single clear point to make any change in the system functionality.

2. *Shotgun Surgery*: This smell is similar to Divergent Changes but the behavior is exactly opposite. When a change is made to an operation, we have to make changes to a number of different classes.

Dispensable

Bad-smells which contain unnecessary code, such as duplicity, fall in this category. These include:

1. *Lazy Class*: A class which is not doing enough and resides in the system because of changes that were planned but have not been implemented yet is referred to as Lazy class. In other words, classes which are included in the software by taking future perspectives into consideration but they have no responsibility at present.

2. *Data Class*: Classes that have only fields and their corresponding getter and setter methods is termed as Data Class. These methods are used by other classes and do not have any behavior of their own

3. *Duplicate Code*: Existence of same code structure at multiple locations in the software system is identified as Duplicated Code. The presence of same expression in two methods of the same class or that of same expression in two sibling subclasses represents a Duplicated Code bad smell.

4. *Speculative Generality*: This smell exists in source code when the system has some code for a functionality that is expected to be required in future but has no use according to the present requirement specifications.

Encapsulators

This category of code smell includes the presence of bad smells which deal with data communication mechanism or encapsulation. These include:

1. *Message Chain*: When one client asks one object for another object, which in turn the client asks for another object and again for yet another object, Message Chain smell exists in the code.

2. *Middle Man*: Middle Man is the delegate methods that can only put forward the requests required by other methods. In such instances, the methods should not delegate the responsibility. Instead the request should go directly to the client class.

Couplers

Bad smells related to coupling include the following:

1. *Feature Envy*: When one method of a class seems to be more interested in some other class rather than the one which contains it, it indicates the presence of Feature Envy.

2. *Inappropriate Intimacy*: When two classes are tightly coupled to each other and are interested in digging too much in other class's private data, it indicates the existence of Inappropriate Intimacy.

Others

Other bad smells which do not fit well into above categories include:

1. *Incomplete Class Library*: This smell exists when the amount of functionality provided by the system library classes are either more or less than the required.

2. *Comments*: Comments are not exactly a bad smell and are actually considered as a sweet smell. The comments are expected to be present in code but excessive use of comments is not considered as a good programming practice and hence is enlisted as a bad smell.

Out of the above defined bad smells, six have been implemented in the JSmell. A detailed analysis of five more bad smells is presented and explained why these were not implemented through software metrics approach.

To improve the design of an existing system we need to apply refactoring to the detected code smells. In such a scenario, we need to know where and when to

apply these refactoring in the system. The refactoring should be applied when we add a new function, fix a bug in the system or do a code review [1].

Refactoring Techniques

We will now discuss various refactoring techniques and describe how these techniques help in improving the system design by removing code smell.

Refactoring for Data Class

As mentioned earlier in this chapter, classes that have only fields and their corresponding getter and setter methods is termed as Data Class. In the below example, the data class, Category, contains variable fields (*category, catid* and *cat_value)* and the getter setter methods for these fields. There is nothing else in the class except the fields and functions specified above.

```
public class Category {
private String category;
private int catid;
public cat_value;

public void setCategory (String category)
{
 this.category = category;
}
public void setCatid(int catid)
{
 this.catid = catid;
}
public String getCategory()
{
return category;
}
public int getCatid()    {
return catid;
}
}
```

FIGURE 2. Data Class Example (*Category* Class).

MaterialDBase class uses the getter method *getCatid()* defined in C*ategory* class in Figure 3.

```
import Java.sql.*;
import Java.util.*;
public class MaterialDBAO
{
ArrayList<Scategory> scat;
ArrayList<Category> cat;
ConnectionManager CM ;
 public MaterialDBAO() {
       scat = new ArrayList<Scategory>();
       cat = new ArrayList<Category>();
       CM = new ConnectionManager();
}
synchronized public String getCategory(int catid)  {
      String cat="";
      try
      {
      String sql = "SELECT * FROM CATEGORY WHERE CATID='"+catid+"' ";
      System.out.print(sql);
      ResultSet rr = CM.executeQuery(sql);
      rr.next();
      cat = rr.getString(2);
      cat = rr.getCatid().getCategory();
      }
      catch(Exception e)
      {
            System.out.println("The Error Occurred in getting name of
CATEGORY-"+ e.getMessage());
      }
      return cat;
 }
synchronized public ArrayList<Scategory> getSubcategories(int catid) {
      try   {
      String sql = "SELECT * FROM SUBCATEGORY WHERE CATID='"+catid+"'
ORDER BY SUBCATEGORY";
      System.out.print(sql);
      ResultSet rs = CM.executeQuery(sql);
      while(rs.next())
      {
            Scategory s = new Scategory();
            s.setScatid(rs.getInt(1));
            s.setScategory(rs.getString(2));
            scat.add(s);
      }
       }
       catch(Exception e)      {
            System.out.println("The Error Occurred in getting
SUBCATEGORIES-"+ e.getMessage());
       }
      return(scat);     }
}
```

FIGURE 3. Data Class Example (*MaterialDbase* Class).

For refactoring we will apply *Encapsulate fields* technique to the public field - . C*at_value*. This means that we will encapsulate cat_value field to private field.

Additionally, we will use *Remove Setting Methods* wherein *catid* will be made final since it does not alter the object of Category class. Also, we can use *move method* to move the behavior in Category class.

The effect of applying Move Method refactoring on Category class is shown in Figure 4.

```
public class Category {
private String category;
private final int catid;
private cat_value;

public void setCategory(String category)        {
 this.category = category;
}

public String getCategory(int catid)            {
      try   {
            String cat = "";
      String sql = "select * from CATEGORY where catid='"+catid+"' ";
      System.out.print(sql);
      ResultSet rr = CM.executeQuery(sql);
      rr.next();
      cat = this.catid;
      }catch(Exception e)
      {
            System.out.println("The Error Occured in getting name of
CATEGORY--"+ e.getMessage());
      }
      return cat;
}
}
```

FIGURE 4. Category class after applying refactoring.

Result

The Data class did not have any responsibility before applying refactoring techniques and its data members were used by another class. By using *Move Method*, the getCategory () method has been moved to the data class *Category* from *MaterialDbase* and new responsibility has been assigned to it. Moreover, the *Category* class is not used by *MaterialDbase* class unnecessarily, hence less coupling.

Refactoring for Speculative Generality

The following code shows the existence of *Speculative Generality* in the following Java code. This smell exists when we have such functionality in the code which is either no longer required or is intended for a future purpose. Here, we have a class *person* having attribute - *name* and method - *getTelephoneNumber()*.

```
public class Category {
private String category;
private final int catid;
private cat_value;

public void setCategory(String category)        {
 this.category = category;
}

public String getCategory(int catid)            {
      try    {
            String cat = "";
      String sql = "select * from CATEGORY where catid='"+catid+"' ";
      System.out.print(sql);
      ResultSet rr = CM.executeQuery(sql);
      rr.next();
      cat = this.catid;
      }catch(Exception e)
      {
            System.out.println("The Error Occured in getting name of
CATEGORY--"+ e.getMessage());
      }
      return cat;
}
}
```

FIGURE 5. Speculative Generality Example (*person class*).

Another class TelephoneNumber has attributes named *areaCode* and *number* and *getTelephoneNumber()* method.

```
class TelephoneNumber {
private String _areaCode;
private String _number;
public String getTelephoneNumber() {
return ("("+ _areaCode+") " + _number;              }
String getAreaCode() {
return _areaCode;
}
String getNumber () {
return _number;
}
```

FIGURE 6. Speculative Generality Example (TelephoneNumber *class*).

In this case, *TelephoneNumber* class is not doing much and hence we can merge it to the class person by applying *Inline class* refactoring wherein all the attributes will be moved to the person class.

```
Public class person {
 String _name;
 String _number;
 String area_code
 public String getTelephoneNumber(){
  return area_code +" "+_number;
}
 Public String getInfo(String name) {
 _name = name;
 return _name + "\r\n"+ getTelephoneNumber();
} }
```

FIGURE 7. Person class after refactoring.

Result

Now person class has all the required attributes. This class has the responsibility required to carry out name and telephone information.

In the above two examples, we tried to establish how refactoring simplified the design problem for Data Class and Speculative Generality bad smells. We will discuss the refactoring types used to correct the code smell in chapter 5. In the next chapter, we will look at the existing work done by various researchers in this field and the software tools available to solve the code smell detection problem.

CHAPTER 3

RELATED RESEARCH WORK

The literature provides many contributions in the field of refactoring. This chapter mentions the research in refactoring and then describes some of the existing software tools which help in the analysis and detection of code smells in the Object Oriented systems.

Current Literature

Matthew James Munro focus primarily on the characteristics of a bad smell [2]. These bad smells have been presented M. Fowler in his book Refactoring [1]. An incremental approach has been tied strongly with the eXtreme Programming (XP). Matthew followed the software metrics approach, defined in [6], and detected the code smells such as *Lazy Class* and *Temporary field.* In his work, the author created bad smell interpretation rule templates and captured those code smells which fit best into the templates. This work is partially based on Matthew's work [2]. The interpretation of software metrics such as LOC (Lines Of Code), NOM (Number Of Methods), CBO (measurement of amount of Coupling) and WMC (Weighted Methods per Class) constraint the smell within the periphery of design and eliminates the possibility of detection of false-positive results as an outcome of interpretation. An overview of the system defined by Matthew [2] is shown in Figure 8.

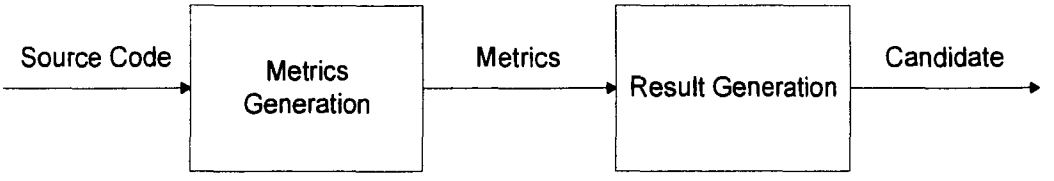

FIGURE 8. Detection Process in [2].

His approach is a great contribution in refining the process of software metrics for future work.

Angelo Lozano, Michel Wermelinger and Bashar Nuseibeh emphasized on understanding the relationship between bad smells and design principle violations [7]. The relationship between maintainability and code smell is based on the extent to which bad smell affect the ease to maintain, adapt or extend the code. It describes a cost-effective relationship between code smell and the violation of design principle. Various other authors have also defended their arguments in this context. This work has evaluated results for code-cloning and found that cloning is not always bad. This derivation is based on the fact of relationship among clones. Kim found about to what extent cloned fragments change in the same manner [8]. This relationship was detected and the changes in behavior of the code smell were found to be very impressive. It was found that most of the clones are used as template to implement new behavior in the system.

D.L. Parnas described a relationship which states that hiding details insulating the change reduces its negative effect and thus is cost-effective in the system comprehension and support the abstraction process in Object-Oriented languages [9].

The work extract sets of bad smells that appear frequently in close locations and try to relate the bad smell from diverse source of information such as dependency relation or common vocabulary.

Mika Mantyla, Jari Vanhanen and Casper Lassenius describes the approach for code smell detection process is based on software quality [4]. This is a progress work of what they had defined previously in [3]. The research aimed at critically evaluating, validating and improving the understanding of subjective indicators of design quality.

Tool Support

There are numerous tools and IDE available for code smell detection and refactoring process. These include jBuilder [10] for Java, csharprefactory for C#, Eclipse [11] for Java. We will discuss about some of the commonly used tools and their approach towards bad smell detection process.

JDeodorant [12, 13] tool identifies two kinds of bad smells, namely "*Feature Envy*" and "*Type Checking*" bad smells. This tool uses the ASTParser API of Eclipse Java Development Tool to detect the bad smell from the source code and ASTRewrite API to apply refactoring. The underlying technique for detecting *Feature Envy* bad smell is based on the notion of distance between methods (entities) and system classes. This bad 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 [12]. This bad smell violates the principle of high cohesion. Regarding *Type Checking* the underlying aim is to implement polymorphism by using Refactoring. The methodology for identification of bad smell involves two cases [13]. In the first case, there can be a

field which represents state (switch case or if-else-if loop) and depending on its value, corresponding conditional branch is executed. In 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 [13].

After bad smell identification, the *Feature Envy* problems are automatically resolved by "Move Method" and "Extract and Move Method" refactoring. T*ype Checking* problems are automatically resolved by "Replace Conditional with Polymorphism" and "Replace Type code with State/Strategy" refactoring [12, 13].

Eclipse plug-in, Metrics [11], is used to detect *Parallel Inheritance Hierarchy* bad smell. In this approach, various metrics are included such as DIT (Depth Inheritance Hierarchy), NOC (Number Of Children), and results are evaluated based on data mining technique. With Metrics plug-in, the process defined in a case study [11] evaluates satisfactorily results. Considering the test case for the JUnit framework, the interpretations are shown in Figure 9.

Cluster	Num. Classes	%	Mean DIT	St. Dev	Mean NOC	St. Dev
0	29	63%	1.8397	0.8793	0	0.7200
1	12	26%	5.6438	0.4825	0.0001	0.0110
2	5	11%	1.5712	0.7282	1.8568	0.6394

FIGURE 9. Evaluation of Approach used in [11].

Prodeoos [14] detects design flaws from C++ and Java programs. The analysis in this tool is based on the *Software Metrics* detection strategy. Prodeoos queries a database which contains the model of the analyzed system in order to calculate all the necessary software metrics. Applying a detection strategy, it creates a report containing all the design entities which are suspected to be affected by the quantified flaw.

DÉCOR [15] detects the code smell constrained to the domain specific language. It generate design-defect detection algorithm. The smell detection rules are specified using the structural and lexical properties and structural relationship. This tool involves three-step process to detect code smell. The first step involves parsing the source code using JFlex and JavaCup. The second step includes reification which is a specification of defects based on meta-model of the target system. Reification is done in order to manipulate the defects and store them in a repository of high-level defects. In the last step, the generation of detection algorithm is implemented as visitors on the meta-model. The algorithm generation process uses the services of SAD (Software Architectural Defects) framework and is based on the templates which are excerpts of Java source with well-defined tags to be replaced by concrete code.

Research work discussed in this chapter provides the direction and basic concepts applied in the functionality of JSmell. The tool implements the functionality using the existing work as an initial step and modifies algorithms discussed in the existing research work to enhance the performance of the tool. The enhancements and the algorithms are described in details in the coming chapters. In the following chapter we will discuss the functionality of the JSmell in details.

CHAPTER 4

JSMELL--BAD SMELL DETECTOR

Introduction

Bad smell detection is a complex procedure. For bad smell detection we need to scrutinize the code files and examine the design of the software system. Manual detection of code smell becomes unmanageable with an increase in the size and complexity of code. JSmell is a software tool that helps the programmers to automatically detect bad smells in the Java source code.

JSmell Functionality

This chapter outlines the functionality and features provided by JSmell. The tool provides the functionality to analyze the code structure of the target system and detects the bad smells in the code, explained in chapter 5. In the tool, once the bad smell is detected, we can view the exact location of code smell in the source code. This tool also provides a tree view of the source code which is a structural breakdown of the system being analyzed. This helps the user in understanding the code structure at high level. Structural breakdown includes the details of the imports, package, classes, methods and variables in each class.

JSmell is designed to support code smell detection, provide refactoring suggestions, and at the same time also to help the user in understanding the system better in terms of its architecture. JSmell is relatively small tool which parses the

input Java source code using classes generated by ANTLR parser generator. The additional feature that JSmell provides when compared to similar tools is that it provides the refactoring suggestion based on the type of smell detected. JSmell can be modified to extend its functionality to support other languages in future by defining templates for each type of language such as C++, C#, Cobol, and Python.

A high level interface of JSmell is shown in Figure 10.

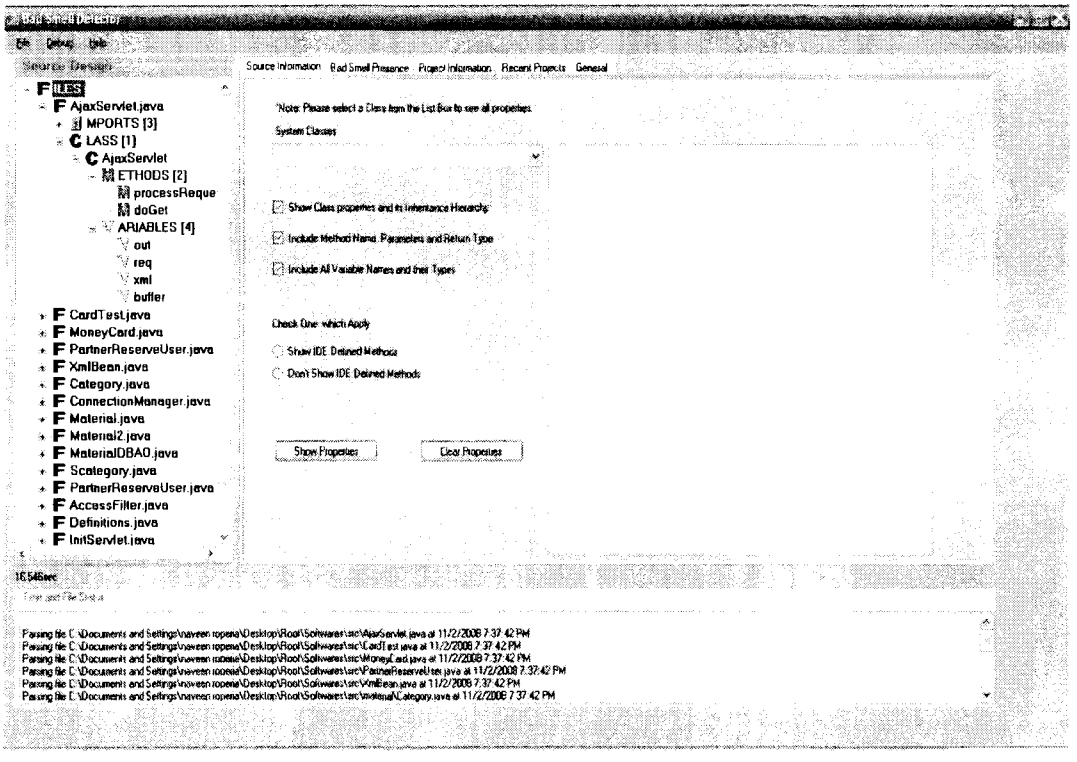

FIGURE 10. High Level Overview of JSmell Software System.

For the implementation of bad smell detection process, the tool implements various data-structures. The data structures mentioned below are used in JSmell for

facilitating the detection of bad smell through Metric evaluation process. An overview of the responsibilities of some of these data structures is given below:

1. An array list which stores the type of the object referenced object in a class.
2. An array which stores the accessor and mutator methods which help in *Data Class* bad smell detection process.
3. An array list containing abstract classes which are utilized in the source code. This is required to eliminate the abstract classes which are not used in the target software system.
4. An array containing data variables which are of Primitive types. These Primitive types are data types which are defined in the Java System.
5. An array list which stores the methods which are the victims of message chain code smell.
6. Arraylist for methods and variables defined in a class.

Bad Smells Detected by JSmell

Automated smell detection process involves the use of data structures described in the last section, metrics count and the information generated from the parser. JSmell detects the following code smells:

1. Data Class
2. Message Chain
3. Primitive Obsession
4. Speculative Generality
5. Parallel Inheritance Hierarchy
6. Duplicated Code

7. Comment

JSmell Performance

JSmell is built on the basis of Object-Oriented programming practices. During smell detection process, algorithms and related metrics count are evaluated using information generated by the parser as well as from the database to solve the problem quickly and efficiently. The criteria used for results analysis include accuracy and the time taken to detect the smell. While testing software having about 460 classes, it was found that the time taken to parse the whole system and save the related metrics information was approximately 4.64 sec.

In the next chapter we will discuss the analysis technique and algorithm used to detect the six different code smells as detected by the JSmell.

CHAPTER 5

BAD SMELL DETECTION

This chapter explains the process for identifying bad smells mentioned in chapter 4. There is a separate algorithm to detect each code smell. The idea behind the notion of algorithms is to use static analysis technique for code analysis. Then, based on the type of code smell, metrics are applied to detect the type of code smell.

Data Class

Problem Definition

A class having only fields, accessors and mutators should be eliminated. Such classes serve the purpose of dumb data holders. Their details are used by other classes.

Description

A method is called an *accessor* when it has a return statement for any of the class fields. An accessor is a method that accesses the contents of an object and returns them but does not modify that object. Methods that alter variables are called *mutators*. In other words, a mutator performs some computation on a data field and assigns a new value to it. Data classes contain only fields, accessor and mutator methods for the fields and nothing else [1]. For example, say in any banking software, we have a class *Account* that contains data fields named *accountName* and *accountType*. The class also contains the accessor and mutator methods for these

fields. There is no other method in this class which defines any kind of functionality. These classes are dumb data holders and fulfill the purpose of other classes. These classes do not have any responsibility and behavior other than getting/setting the values of specified attribute(s). So, these classes are not in accordance to the Object-Oriented design principles.

Intent

Detect the presence of data class code smell and to suggest the corresponding refactoring process to remove it.

Detection Technique

Since the existence of code smell is due to the symptoms defined in the problem definition section above, our interest is in the detection of metrics related to these symptoms. Number of Variables (NOV) per class are required to match the variables returned by the methods. A heuristic approach is applied to count the probable number of Line of Code (LOC_{prob}) for data fields, methods, imported packages, and the package declaration.

This information is matched with the actual LOC_{actual} to detect the probability of true-positive results. Metrics analysis is applied on the static information which include NOV, LOC_{prob}, LOC_{actual}. If any other method exist other than the accessors and mutators, the tool will not detect it as a data class. This is because the class will have some responsibility if any other method exists. Also, the abstract classes, structs and unions are not considered as data classes.

Algorithm

1. For each class, check if any method is returning a declared data field.
2. Check the existence of method where any declared data field value has been reset.
3. Check if the returned data field exists in the defined list of declared variables because returned field can be a variable local to the function defined in the class.
4. Calculate the actual line of code for the class, LOC_{actual}.
5. Calculate the heuristic number of line of code, $LOC_{prob,}$ for the existence of data class which includes LOC for data fields, methods and other declared imported packages.
6. Capture the classes which satisfy above constraints.

In Figure 11, *displayNumber (String attribute)* method in *JSmellParser* class counts the number of methods, data fields, global variables (scope global to the class and not confined to the methods). Since *data class* contains getter and setter methods (referred to as accessors and mutators) on global variables, variables local to the method are not taken into consideration in this count. Now *methodReturnVal* collection contains the variables returned by methods defined in the class.

If the returned data fields matches the global variables to the class, then it is considered as a *hit*. On each hit the class information is saved in *showReport(string file_location)* method. The informaton captured on each *hit* is again filtered in *matrixCalculation(String file)* method. This method returns either a true or false

value, claiming the presence or absence of data class presence in the target Java class, respectively.

Process Flow

```
if((parse.displayNumber("methods")>= parse.displayNumber("globalvar")))
{
 foreach (string retVariables in parse.methodReturnVal.AllKeys)    {
   foreach (string variableDeclared in parse.variableReturnVal.AllKeys)
{
   if (retVariables.ToString().Equals(variableDeclared.ToString())
{
    if (parse.GlobalVarArrayList.Contains(retVariables.ToString()))
         {
          hits++;
          varArrayList.AddRange(parse.GlobalVarArrayList);          }
     }
  }
}
if((fileCount <= matrixCount)&& fileCount >
parse.displayNumber("globalvar"))
            {
                this.Result = true;
            }
            return this.Result;
```

FIGURE 11. Data Class Detection process.

Suggested Refactoring

In order to remove *data class bad smell*, one must apply *Encapsulate Field* on the getter and setter methods. A lookup is performed for the location of these getter and setter methods that are being used by other classes. Use of *Move Method* is suggested to move behavior into the data class. If it is not feasible to move the whole method, in such case, Extract method can be used to create a method that can be moved.

Message Chain

Problem Definition

The chain of object call should be eliminated. This happens when a client has to use one object to get another, and then use that one to get to another. Any change to the intermediate relationships causes the client to have to change.

Description

The *Message Chain* is a code smell violating the *Law of Demeter* [16]. The Law of Demeter is a simple, programming language independent style rule for designing the object-oriented systems. This code smell is contrary to the law of Demeter, according to which messages can only be sent to the following closely related units [16]:

1. a parameter of the method, including the enclosing object (this or self).
2. a global object (for pragmatic reasons).
3. an object that a method called on the enclosing object returns, including attributes of the enclosing object.
4. an element of a collection which is an attribute of the enclosing object.

The essence of this rule is "*Only talk to immediate friends*" [16]. In general it means that each attribute should have limited knowledge about the external world and should communicate only with the objects *closely related* to it. The actual cause of this smell is high coupling of objects with other objects and low cohesion within the same class. These message chains are problematic because they expose unnecessary dependencies and may introduce data access bugs [17]. This is a result of poor design

of the inter-object associations. For example, the following Figure 12 shows an example of message chain:

```
public void Method (Person person)
{
 person.GetCulture.GetCountry.Language.ToFrench ("Hello world");
}
```

FIGURE 12. Message Chain Example.

Intent

Detect *message chain* code smell from the target source code. Also, suggest the corresponding refactoring process to remove this design defect from the code.

Detection Technique

While parsing message chains, filter the message chain method call with respect to the methods defined in the classes. Hence, the system defined method calls can be eliminated. The object call has been restricted to the maximum of one method only. More than one object call violates the *law of Demeter*. The metric calculation involves number of Method calls (NOM_{calls}).

Algorithm

1. For each class, obtain the methods defined in the target application code.
2. Check the method calls where object interaction count is more than two, that is, object calls having $NOM_{calls} > 2$.

3. From the selected method, filter only those affected method calls which are defined in the application. There are methods in the libraries of IDE which may contains message chain. But we are restricted to find it in target application.

4. For each smell detected, consider it as a hit.

5. Each hit is saved in *msgChainArrayList*. Inspect the presence manually to confirm the result.

Process

```
foreach (string str in parse.MsgVarChainArrayList)
{
 if (AllMethodsInProject.Contains(str.ToString()))
 {
 msgChainArrayList.Add(str.ToString());
 hits++;
 }
}
if (hits >= 1)
this.Result = true;
return this.Result;
```

FIGURE 13. Message Chain Detection Process.

Suggested Refactorings

Hide Delegate refactoring process is suggested in order to remove this code smell. When a client calls a delegate class of an object, the delegate references the appropriate objects/methods as per client requirements by hiding the functional details from the client. Another method that can be used is *extract method* followed by *move method* to move it down the chain similar to as explained in [1].

Primitive Obsession

Problem Definition

Excessive use of Primitive data types in a class is a bad design issue.

Description

The use of Primitive data types is important to represent the class as a component. If we deal with the normal use of an object, unnecessary Primitive data-types should be removed from the class and new object should be created with its behavior representing the responsibility assigned to this new class. Excessive use of Primitive data type is considered to be a design defect and is called Primitive Obsession. The number of Primitive data types declared in a class should be based on the responsibility assigned to the class.

Intent

Detect the occurrence of classes which are victims of *Primitive obsession*, that is, classes with superfluous usage of Primitive data types. Suggest appropriate refactoring methods to remove this code smell and hence helps in removing this design defect.

Bad Smell Detection Technique

We need to capture classes having extortionate number of Primitive data types. As the first step, the Numbers of Variables (NOV) per class are obtained. An average number of variables count V_{avg} is calculated, which is the mean of number of variables in a class. This average count is considered as the standard number of data fields per class. In order to count the excessive use, only classes with above-average Primitive data types are obtained during parsing. Classes which are not used as components are

filtered, that is, those classes having more than average number of Primitive data types and not been utilized in the application because of planning for future enhancement of those classes in the software. In the last step, a set of metrics are applied on the detected classes. The classes that satisfy the metrics conditions are considered as the design defects/ bad smells in the code.

Algorithm

1. Obtain the class metric values of NOV (Number of Variables per class) for each one of the classes.
2. Observe the average count on the number of variables obtained, V_{avg}.
3. Find all the classes having NOV greater than V_{avg}.
4. From the selected group, obtain all the parents and child classes for each class, if any.
5. Obtain a list of defined types (classes) declared in each class of the target application.
6. From the list of selected group, along with their child and parent, obtain the types whose instance is not created in any other class.
7. Capture and display the classes which satisfy the above metrics.

Process Flow

```
Average_Primitive_variables = JSmell.Num_variables/JSmell.Num_classes;
average_Primitive_variables = average_Primitive_variables + 1;
average_variables_in_class =
parse.displayNumber("Primitiveglobalvariables");
   if (average_variables_in_class > average_Primitive_variables)
     {         CurrentClassName = parse.CurrentClass.ToString();
        insertPrimitiveClass(CurrentClassName);
        updateClassesHierarchy();
        selectChildClasses();
      foreach (object childClass in childClasses)          {
       if (parse.NonPrimitiveArrayList.Contains(childClass.ToString()))
                        this.Result = false;
                    else
                        this.Result = true; }
}

return this.Result;
}
```

FIGURE 14. Primitive Obsession Detection Process.

The detection process is shown in Figure 14. The variable *average_Primitive_variables* returns the number of data fields that are global to the class. The arraylist *NonPrimitiveArrayList* contains defined class types so that only Primitive data types can be filtered during parsing. Then *updateClassesHierarchy()* method returns the parent and child classes. Finally, those filtered classes are marked as Primitive obsessions either whose parent or child classes are not utilized in the target application.

Suggested Refactorings

If a separate representation is required for some fields that are listed in a class, for example if we have a class named "*Order*" that has two fields referring to the field *name* and *telephone* for some customer of the underlying system and also some other

fields related to order specifications, then we can use *Replace Data Value with Object* and move name and telephone fields to new class and associates *order* class with new class.

Alternatively, if we have a group of fields that should go together, the use of Extract Class is recommended, that is create a new class containing group of fields which should go together. These fields should represent the attributes necessary to represent the responsibility of the new class. If we have a lot of parameters as Primitive data types, use Introduce Parameter Object refactoring process.

Speculative Generality

Problem Definition

The existence of classes and methods which are not utilized in the software but are still present in the source code increase the software complexity and degrades the system comprehension. These classes and methods exist in the source code because of some future enhancement in the software system. These methods and classes should be removed from the source code.

Description

Brian Foote suggested that this smell exists when we say "Oh, I think we need the ability to this kind of thing some day" [1]. So primarily, if there are any classes/modules which are included in the software and are not being currently utilized, it is identified as a bad smell. Although, this is done to add flexibility to the software for future wherein developers think that they might need this kind of functionality in the software at certain future point of time yet it is not advisable because unused code increase difficulty in code comprehension. Addition of

unnecessary code often adds bugs in the system architecture. Emphasis should be on resolving current problems and satisfying current requirements. The subsequent problems that shall happen in future should be handled when they are actually encountered. As stated in [18], developers can save time by avoiding writing the code which is not required.

Intent

Detect *speculative generality* from the target source code wherein we have to detect unused abstract classes and methods from the source code. Appropriate refactoring process needs to be suggested in order to get rid of this design defect.

Bad Smell Detection Technique

The presence of unused abstract classes in the source code, which might have been defined with intention of being used in future indicate the presence of speculative generality. These classes that are not used anywhere in the target system, are parsed by the parser and information is extracted for such unused classes.

Algorithm

1. Obtain the set of abstract classes present in the target source code.
2. While parsing, capture the abstract classes that have not been implemented anywhere in the system.
3. Apply metrics to locate these unimplemented classes in the system.

Process Flow

```
if ( JSmellGramParser.ImplementedAbstractClass.Count != 0)
 {
  if (!parse.AbstractClass.ToString().Equals(""))
  {
  if (!JSmellGramParser.ImplementedAbstractClass.Contains(
      parse.AbstractClass.ToString()))
             {
              specGenArrayList.Add(parse.AbstractClass.ToString());
              hits++;
              }
    }
  }
  if (hits > 0)
  this.Result = true;
  return this.Result;
```

FIGURE 15. Speculative Generality Detection Process.

As shown in Figure 15, data field *"AbstractClass"* returns the current abstract class during parsing process. If *ImplementedAbstractClass* array list does not contain the specified abstract class, it will be captured. The presence of this abstract class just detected corresponds to the presence of unimplemented abstract class (s).

Suggested Refactorings

To alleviate this bad smell from the code, it is suggested that the abstract classes that are not doing much should use *Collapse Hierarchy*, that is, when superclass and subclass are not handling anything much different separately, they should be merged using collapse hierarchy. If you have methods with unused parameters, use *Remove parameters*.

Parallel Inheritance Hierarchy

Problem Definition

Existence of parallel hierarchy forces the duplicity in code. It should be eliminated from the source code.

Description

Parallel inheritance hierarchy is a case of relative dependency. When one hierarchy is dependent on the other, then for every change made in one hierarchy, we have to make corresponding changes in the other hierarchy also [6]. It is a special case of shotgun surgery.

Intent

Detect *parallel inheritance hierarchy* from the source code and suggest the possible refactorings which can help in elimination of this design flaw.

Bad Smell Detection Technique

A simple way to detect this code smell is to find the number of classes (NOC), called nodes, at each level, called Depth Inheritance Hierarchy (DIT). This hierarchy level is defined as the level of classes in the hierarchy where root node is super class at level zero, subclass at level one and so on. One criterion to find this smell is that only those hierarchies are taken into consideration that are having level greater than or equal to two. Each node is visited once and name of the class, its root class and level are obtained. This information is stored in the database. Now, the hierarchies which have similar number of nodes at each level are detected at this point. These are the root classes in the hierarchy.

Algorithm

1. Obtain the class metric values of DIT, Number of Children (NOC) at each level for all classes.

2. Find DIT having depth greater than two.

3. From the selected cluster of hierarchies, select the classes having similar DIT and NOC values at corresponding levels.

Suggested Refactorings

The use of *Move Method* and *Move Field* is suggested which will combine the hierarchies into one [1]. With the use of this refactoring technique this code smell shall disappear.

Duplicated Code

Problem Definition

Occurrence of same code structure in more than one place indicates the presence of duplicated code smell. This phenomenon occurs very frequently in large software systems.

Description

Code duplication is considered as bad programming practice as it affects the maintenance and evolution of a software system [19]. It is difficult to track the fragments of similar code scattered throughout the system. The presence of large volume of duplicated code makes it difficult to track bugs in the system. It is complex to handle bug fixing in case of duplicated code because it requires fixing at multiple locations which in turn increases the coding and testing times. Code duplication

introduces long and repetitive sections of code in the system which violates the principle of reuse in Object Oriented Systems.

Intent

Detect the presence of *duplicated code* smell and to suggest corresponding refactoring process to remove it.

Detection Technique

The existence of duplicated code smell is detected using the variation of O (ND) algorithm, provided by Eugene Myers in [20]. The algorithm follows the divide-and-conquer implementation of the longest common-subsequence (LCS) algorithm. In this algorithm, two files are compared based on the identical code found in them. Each line of code is transformed into a hash number. This hash number is computed by storing each new line into a hash table and the corresponding line number is the hash value in it. These hash values are stored in an array corresponding to each file. The algorithm compares two arrays of numbers and the implementation is done in the *DiffCodes* method. D*iffText* method returns the hash values where any difference is detected in two files. This difference is ignored and the similarity between the two files is captured in the *return_duplicate* method. This method returns the captured similar code found in each file.

Algorithm

1. For each file, assign a hash number to every line.
2. Store the hash values in two different arrays.
3. Check the difference in the sequence of hash values appearing in both the files.

4. Obtain the hash number of the code line where the first difference in code was detected. Also capture the corresponding number of lines for which no other similarity could be detected in the two files.

5. Obtain the next similarity in code by adding the line number at which the last difference was found and the number of lines for which no other similarity was found, which were obtained in step 4.

6. Discard the differences found in hash numbers and the rest of the lines of code are identified as duplicated.

Process Flow

In Figure 16, *return_duplicate* method returns the duplicated string captured between the files mentioned, *fileA* and *fileB*. *showDuplication* method implements *return_duplicate* and update the hash values returned by the *DiffCodes*.

```
public string return_duplicate(string fileA, string fileB, int min, int
current_change_in_first, int current_change_in_sec, int
total_line_changein_first, int total_line_changein_sec)
        {
            int current_end_index = 0, track_line_num = 0;
            string read_data = "", return_data = "";
            StreamReader sr = new StreamReader(fileA);
            current_end_index = current_change_in_first;
            while ((read_data = sr.ReadLine()) != null)
            {
                if (track_line_num >= min && track_line_num <
current_end_index)
                {
                    return_data += read_data.ToString() + "\n";
                }
                track_line_num++;
            }
            return return_data.ToString();
        }
```

FIGURE 16. Duplicated Code Detection process.

On each update in the hash values, the similar code is captured between *min* value of hash and the *current_change_in_first*, total changes in current value of the hash.

Suggested Refactoring

In order to remove *duplicated code bad smell*, one must apply *Extract Method* in two sibling classes. Then using *pull Up Method*, move the method from sibling classes to the parent class. If two methods have same functionality but using different algorithm, then choose the best algorithm and use *Substitute Algorithm.*

Comments

Problem Definition

Excessive use of comments is a bad design process. It should not be present in the source code.

Description

Comment isn't basically a bad smell; instead it is a sweet smell [1]. The reason comment is a bad smell is when thickly commented code is written in order to explain the structure of the code [1].

Intent

Detect the comment smell from the source code.

Bad Smell Detection Technique

For pointing out the presence of this code smell we check the number of methods in the classes. The constraint is that only two comments are allowed per method to describe its functionality. The parsing of the comments is done at class as well as method level. In Figure 17, p*arse.displayNumber ()* method calculate the

number of methods in a class. For a class, the number of comments assumed here are equal to the number of methods defined in the class. An extra comment is added in the condition to allow one comment to define class responsibility. Any *count* greater than the specified condition is considered as a code smell.

Process Flow

```
if (matches.Count > (2*parse.displayNumber("methods")) + 1)
{
    foreach (Match match in matches)
      {
        commentArrayList.Add(match.ToString());
       }
     this.Result = true;
}
return this.Result;
```

FIGURE 17. Comment Detection Process.

Suggested Refactorings

The suggestion for the refactoring says that change the name of classes and methods in such a way that their functionality can be clearly understood from the names. R*ename Method* can be used to change methods' name.

The next chapter explains other five code smells which cannot be detected automatically through. These code smell detection process involves behavior analysis. Manual process is highly recommended to find these smells.

CHAPTER 6

BAD SMELLS BEYOND THE SCOPE OF JSMELL

As we saw in last chapter, JSmell detects some code smells automatically. It is of interest to mention here that few bad smells can even be detected through human intuition. The process of such manual smell detection involves behavior analysis of the source code. This chapter discusses four such bad smells which can be inspected through this technique.

Divergent Changes

Problem Definition

The existence of such design strategy in which code modification in a single class takes place in different ways for multiple reasons is known to be *Divergent Change* smell.

Description

Software is meant to be soft [1]. We should be able to reach to any single point of concern for making any change in the system design. There should be least dependency between components so that while making any changes in the software, we are not forced to make huge numbers of other changes. For example, let's consider a banking software system where we have separate classes for various account types. Now, if the bank plans to change business logic for an account type, the changes that need to be applied to a method would also need modifications in various related data

fields. It might be possible that an existing database also needs to be changed which might be affecting the existing business rules. In order to communicate with other objects, the class may need to add a separate delegate class. Additionally, it can also happen that class might be containing some unrelated functionality. The changes thus taken place affects the existing design of the system and as a consequence decreases the system efficiency which is a matter of concern. The design should be such that for any changes made in source code, it is easy to jump to a single point of concern.

Why Software Metrics not used for detection?

As stated earlier, JSmell detects code smell using metrics evaluation technique. This code smell requires the detection of functionality change among different versions of the system under observation. It is beyond the scope of Metrics technique to capture such statistics. Such prognosticative analysis requires keeping different versions of the system which cannot be achieved with static analysis and metric evaluation techniques.

Shotgun Surgery

Problem Definition

Existence of design strategy in which one kind of change in the code forces the developer to make lots of other little changes in different classes is identified as Shotgun Surgery.

Description

Shotgun Surgery is similar to *divergent changes* with the difference that it is totally opposite in intent. Any change to the new functionality may require widespread changes in the source code. When a change at one place requires lot of

other changes to complete the requirement, it is difficult to track places where changes are required. It is possible that an important place that needs change may be missed which leads to serious code defect. This design defect also makes the software difficult to maintain and understand and demands significant number of man-hours for incorporating a small change in the system. Such changes in the software development help in short-term improvement at the expenditure of long-term maintainability and stability.

Why Software Metrics not used for detection?

The strategy for detecting *shotgun surgery* requires that we should trace all those data fields and methods that need change in their behavior to implement new functionality. Also we are required to know how many classes are affected when any such change is required. It is not possible to determine this through static analysis because software metrics does not provide any way to capture such statistics.

Alternative Classes With Different Interfaces

Problem Definition

In a system if methods in different classes are performing similar actions but are having different signatures, it is referred to as Alternative Classes with Different Interfaces.

Description

If two classes are similar in functionality, but otherwise they look different from outside, they can be modified to share a common interface [21]. This implies that classes containing methods which have the same behavior but are different in their signatures should be moved to a common interface. This helps in removing code

redundancy. It is also expected that the naming methodology of the methods should be in accordance with the functionality that they intent to deliver.

Why Software Metrics not used for detection?

This code smell requires detecting the methods providing same functionality. The software metrics analysis technique does not equip us with the ability to capture such dynamic functionality of the code. Thus this bad smell needs human analysis for successfully interpreting the existence of such bad smell.

Incomplete Class Library

Problem Definition

Existence of library classes that lack the intended functionality, that is, reusability has not been fully implemented in the software.

Description

Class libraries are built in the object oriented languages in order to implement reuse of code. There is no border line which can distinguish in between which functionality should be included in the library classes and which should be implemented in concrete classes. This is decided during the design phase of the software and, more precisely, depends on the software requirements. For example, if an application requires that it should contain some kind of sorting algorithm, then it will be decided during the design time to include the algorithm in the library class. This is totally a manual process. Metric evaluation and static analysis techniques stand to be of no help in this process.

Why Software Metrics not used for detection?

The detection process of this code smell requires the involvement of software designers as well as software developers. The reason why this code smell cannot be detected using software metrics is that there is no possibility of tracking the reusability requirement specifications using this technique.

CHAPTER 7

INPUT VALIDATION IN JSMELL

This chapter describes the validation test for the algorithms used in code smell detection process in chapter 5. Java source code is given as input to the JSmell to validate the detection process. JSmell was tested on three different sample programs which include ANTLR open source IDE containing 408 classes, one web-based application containing 17 classes and a small desktop application containing 11 classes. The algorithms described in chapter 5 successfully detect the presence of code smell in the source code. The outputs are then verified manually against the type of code smell detected. 90% of the detected smells found shows true positive results. Since metrics evaluation technique involves assumptions of the metrics values for code smells, it is not possible to generate an output of 100%.

JSmell reads the Java source code and extract the static information iteratively for each class. The structural breakdown of classes, methods and defined variables is displayed in the form of tree node where each node may represent a file, a class, method, data fields. The tree structure is helpful in understanding the system design. Now the user has option to check any of the six code smells listed in chapter 5. Once the user select the option to check the presence of code smell, JSmell runs the algorithm associated to that smell and display detected code to the user, if found. If no code smell is detected by the JSmell, it simply displays a message showing that code

smell was not found. Although JSmell does not provide 100% output, it helps the developers to constraint their search for required changes to specific blocks of code.

JSmell not only display the smell detection output, but also save the information for users' convenience. At any time user can save the output information and retrieve it when required. User Interface of the JSmell is shown in Figure 18. User can select the input file/project using the File menu followed by *debug* menu and selecting *Parse File*. The next section will describes the input test code given to JSmell and the output generated for the type of code smell.

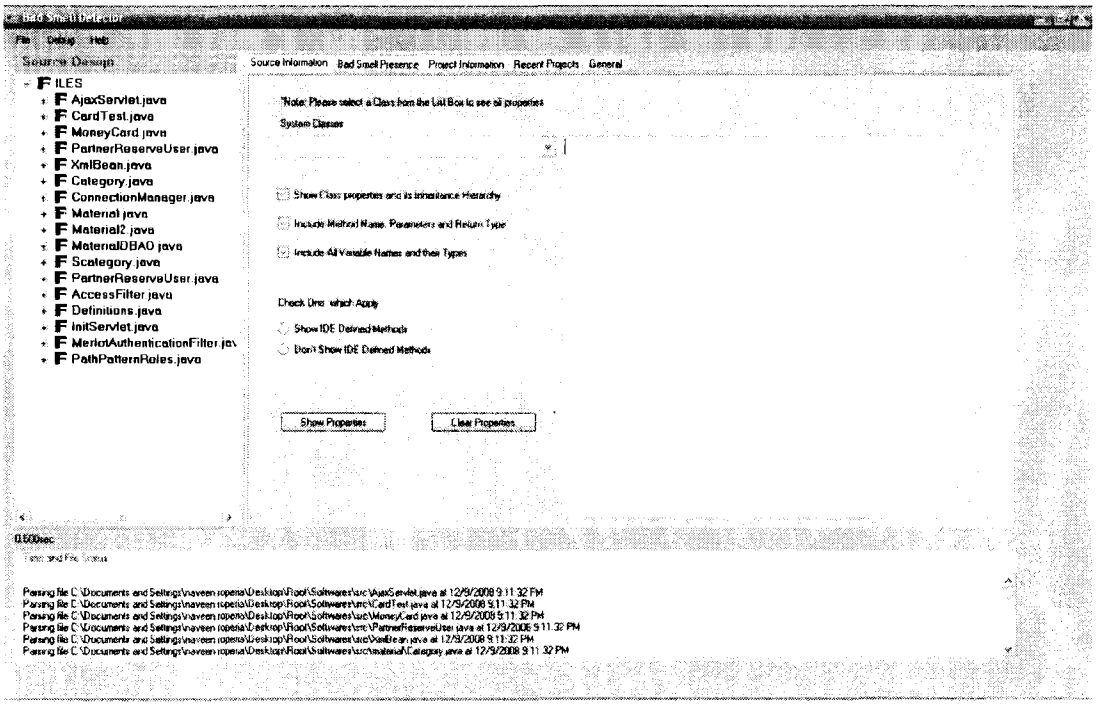

FIGURE 18. Initial User Interface of JSmell.

Input Validations for Code Smells

Input Validation for Data Class (Case1)

The first case scenario includes a web based project scenario which contains 17 classes. This project uploads the files (materials). For this purpose, files are organized in different categories based on their types. Each category of file represents related contents. The category class serves the purpose of data class in this case. It deals with setting a new category of material to be uploaded or retrieve the category name during catalog display. The project contains various classes necessary for its functionality. The source code described in Figure 19 is for data class only.

The data class detection algorithm captures *category* class because it satisfies all conditions mentioned in the algorithm. These conditions include the presence of accessor and mutator methods (*getSetMethods*), returning globally declared data variables (*methodReturnVal*). Here global means data fields global to the class and not local to the methods. C*atid* and *category* data fields are declared at class level and *setCategory* and *getCategory* are the mutators and accessors, respectively. These methods alter and return the values of *category* data field. Same is the case with *catid* data field. The output shown confirms the true-positive result for the sample input in case1.

```
package material;
public class Category
{       //Variables
String category;
int catid;

//Methods: getter and setter only
public void setCategory(String category)
{
 this.category = category;
}

public void setCatid(int catid)
{
 this.catid = catid;
}

public String getCategory()
{
return category;
}
public int getCatid()
{
return catid;
}
}
```

FIGURE 19. Sample Input in Case1 for Data Class.

When the project input is given to the JSmell, it detects the above shown class as data class code smell. The output displayed for data class is shown in Figure 20.

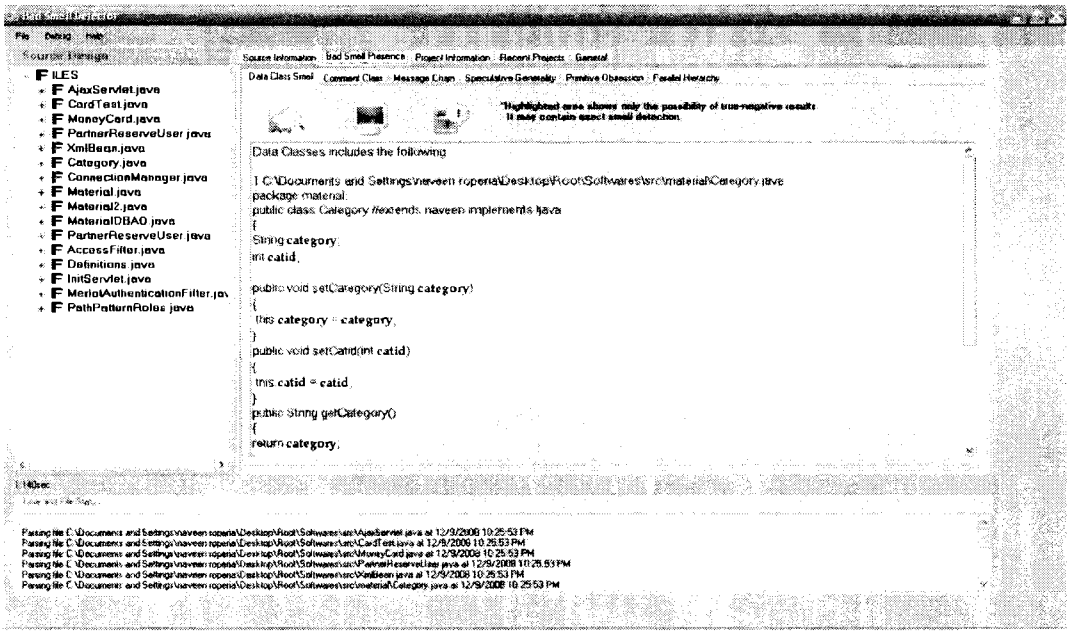

FIGURE 20. Output for Data Class smell detection in Case1 Scenario.

Input Validation for Data Class (Case2)

In case2, sample Input of an application having 11 classes is given to the JSmell. In this case, class *Contact* has five data fields and corresponding accessors and mutators. This case also provides a true-positive result for the sample input. So far in this case, the output is 100%. The output is shown in Figure 21.

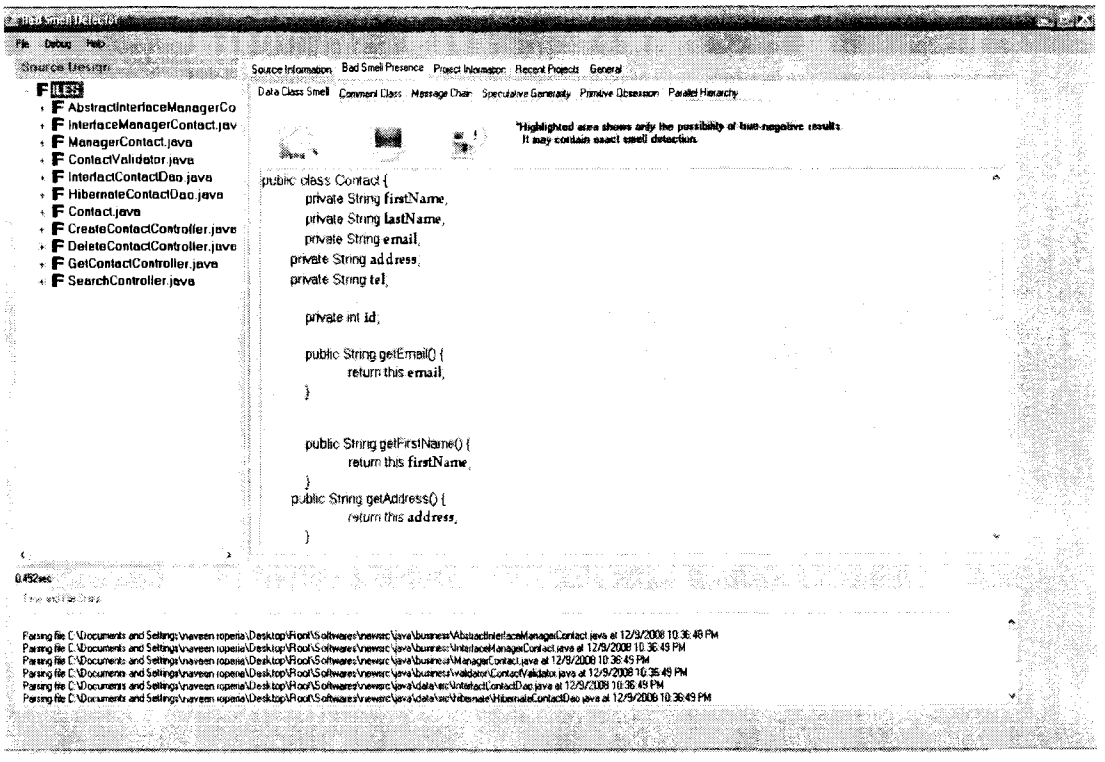

FIGURE 21. Output for Data Class smell detection in Case2 Scenario.

Input Validation for Message Chain

The existence of message chain code smell is detected in the project having 11 classes. This is a make-up scenario where methods are added in chain for test purpose. The code shown in Figure 22 (shown in blue) contains one instance of message chain which is successfully detected in JSmell.

The *MsgVarChainArrayList* array list in JSmell contains a list of all methods having a metod call chain of more than two instance. These methods are then filtered in *AllMethodsInProject* arraylist because JSmell only allowed message chain detection for methods defined in the project. It eliminates all instances of method chains which contains methods defined in the IDE Library.

```
import Java.sql.*;
import Java.util.*;
public class MaterialDBAO
{
ArrayList<Scategory> scat;
ArrayList<Category> cat;
ConnectionManager CM ;
 public MaterialDBAO() {
        scat = new ArrayList<Scategory>();
        cat = new ArrayList<Category>();
        CM = new ConnectionManager();
}
synchronized public String getCategory(int catid)  {
       String cat="";
       try
       {
       String sql = "SELECT * FROM CATEGORY WHERE CATID='"+catid+"' ";
       System.out.print(sql);
       ResultSet rr = CM.executeQuery(sql);
       rr.next();
       cat = rr.getString(2);
       cat = rr.getCatid().getCategory();
       }
       catch(Exception e)
       {
              System.out.println("The Error Occurred in getting name of
CATEGORY-"+ e.getMessage());
       }
       return cat;
 }
synchronized public ArrayList<Scategory> getSubcategories(int catid) {
       try   {
       String sql = "SELECT * FROM SUBCATEGORY WHERE CATID='"+catid+"'
ORDER BY SUBCATEGORY";
       System.out.print(sql);
       ResultSet rs = CM.executeQuery(sql);
       while(rs.next())
       {
              Scategory s = new Scategory();
              s.setScatid(rs.getInt(1));
              s.setScategory(rs.getString(2));
              scat.add(s);
       }
        }
        catch(Exception e)      {
             System.out.println("The Error Occurred in getting
SUBCATEGORIES-"+ e.getMessage());
        }
       return(scat);      }
}
```

FIGURE 22. Sample Input for Message Chain Case Scenario.

The output for message chain is shown in Figure 23. The message chain is highlighted in color. The participating classes are shown in the display area.

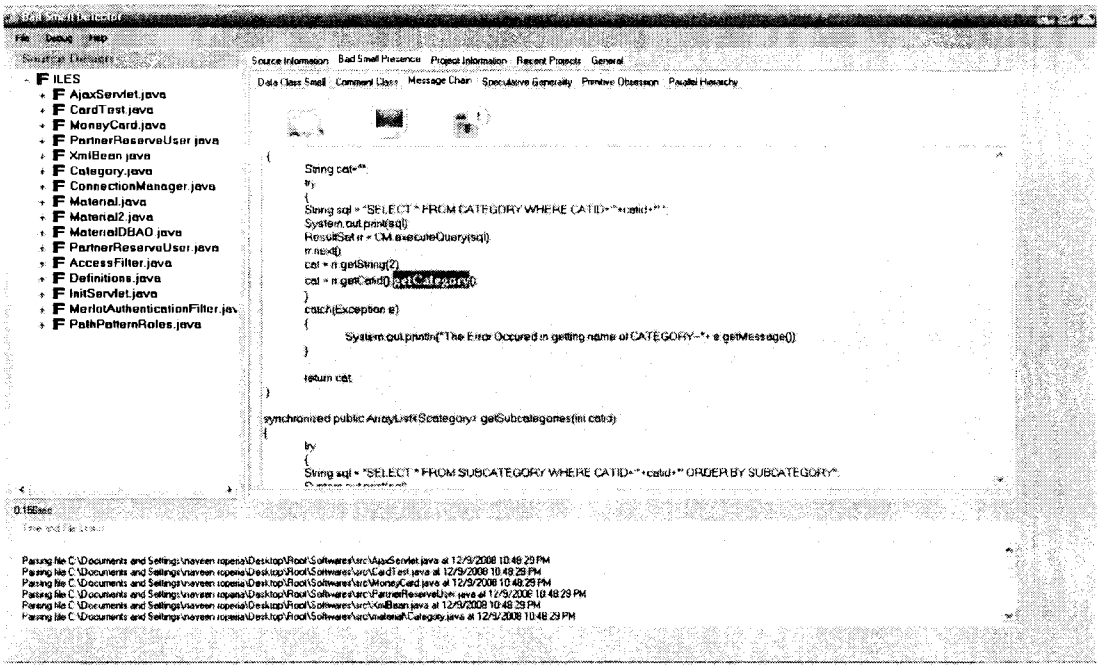

FIGURE 23. Output for Message Chain Case Scenario.

Input Validation for Primitive Obsession (Case1)

The input code for validation, shown in Figure 25, displays the presence of Primitive obsession code smell. The code in Figure 24 is partial code for the class. It seems to be a data class but not actually. It contains other methods also but not shown here as the class has 200 LOC.

This code smell is detected from the web based application containing 17 classes. The initial decision for Primitive obsessive class is that the number of declared data variables (global to the class) should exceed the number of average declared data fields per class, V_{avg}. Also, *Material2* is a make-up class whose object is not instantiated anywhere in the application. These two conditions are fulfilled as defined in the algorithm.

Input Validation for Primitive Obsession (Case2)

Now, for the same sample input, if we create an instance of *Material2* no class has been detected as Primitive Obsession class. This is in accordance to the Primitive obsession algorithm defined in chapter 6. The output in JSmell shows true positive results. This is shown in Figure 26. The sample input code for Primitive Obsession is shown in Figure 24.

```
public class Material2 extends Material
{
      public String name;
      String author_fname;
      String author_lname;
      String description;
      String location;
      String date;
      int catid, scatid;
      String filesize;
      String fileformat;
      String submitter;
      String view1;
      String view2;
      String view3;
      String view4;

            public void setName(String name)  {
        this.name = name;
            }
             public void setSubmitter(String submitter)  {
        this.submitter = submitter;
            }
             public void setFilesize(String filesize)  {
        this.filesize = filesize;
            }
             public void setFileformat(String fileformat)
            {
        this.fileformat = fileformat;
            }
             public void setAuthor_fname(String author_fname)  {
                  this.author_fname=author_fname;
             }
             public void setAuthor_lname(String author_lname) {
                  this.author_lname=author_lname;
             }
            public void setDescription(String desc) {
                  this.description = desc;
            }
            public void setLocation(String loc) {
                  this.location = loc;
            }
            public void setDate(String date) {
                  this.date = date;
            }
            public void setCatid(int catid)
            {
                  this.catid = catid;
               }
}
```

FIGURE 24. Sample Input Code for Primitive Obsession.

The output for Primitive Obsession is shown in Figure 25.

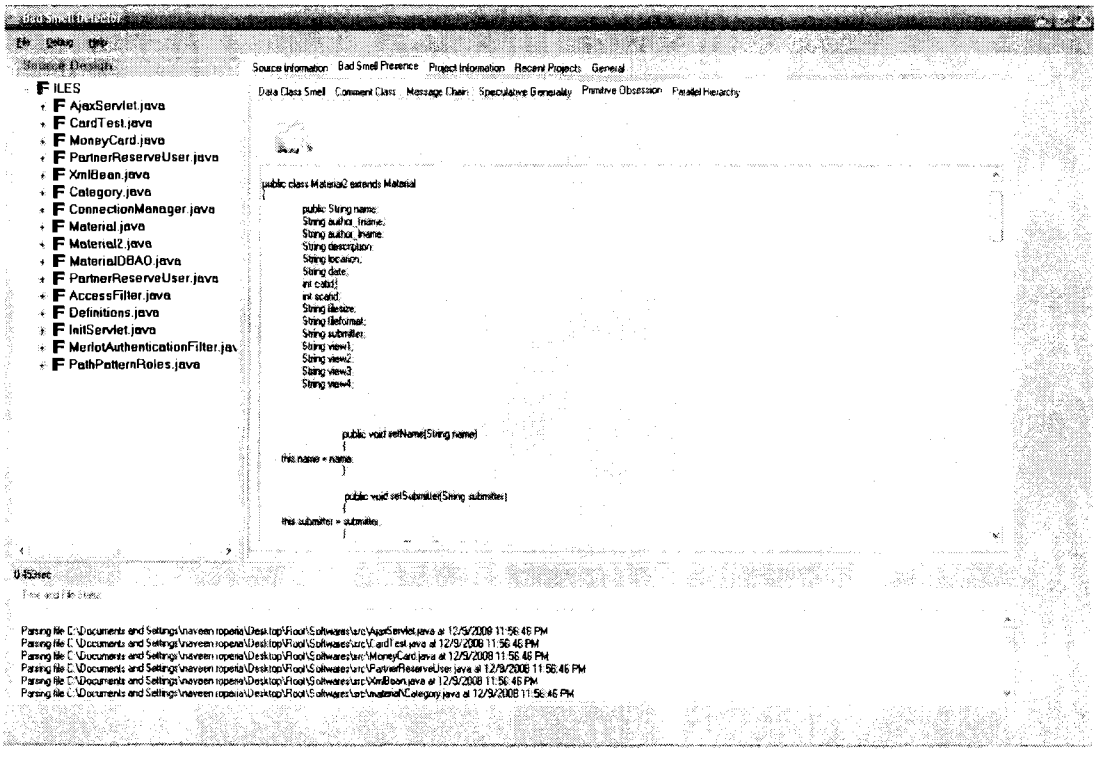

FIGURE 25. Output for Primitive Obsession in case1.

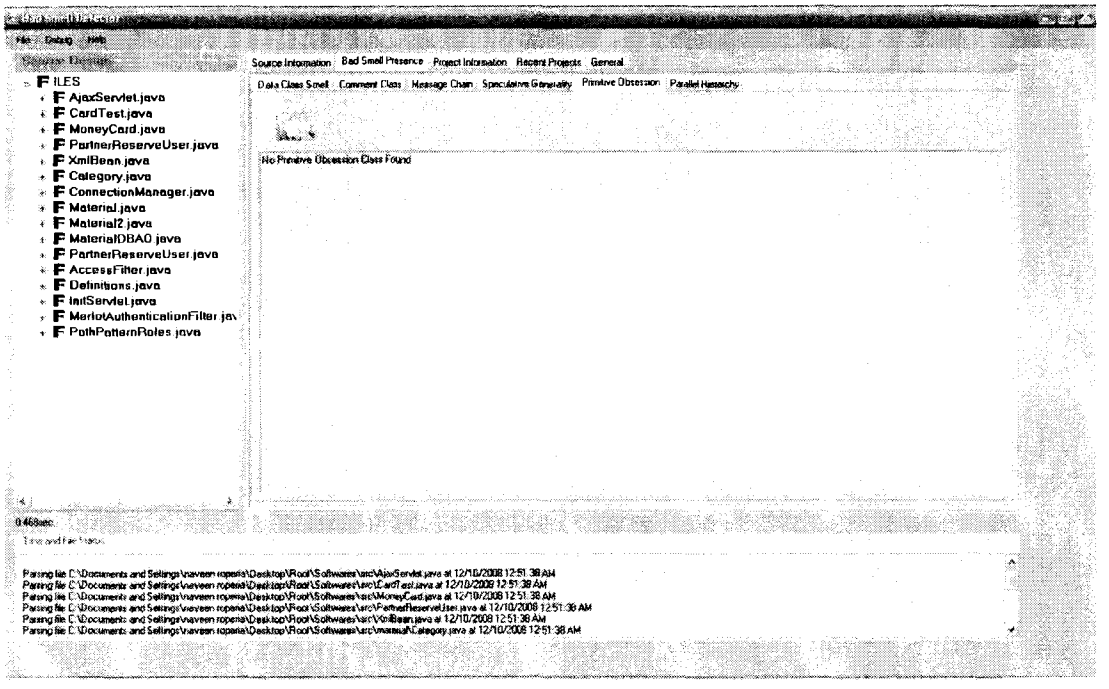

FIGURE 26. Output for Primitive Obsession in case2.

Input Validation for Speculative Generality

The speculative generality detection algorithm looks for unused classes, methods and parameters. Our algorithm reported true positive results for the sample inputs given to the JSmell. One perspective for detecting speculative generality is to capture the abstract classes which are defined in the application but not used anywhere. These classes are created by taking future functionality into consideration. The sample input code contains abstract class which is not utilized in the application and detected in the JSmell. A sample input for Speculative generality is shown in Figure 27.

The abstract Class *AmountCard* is not inherited in the web based application with 17 classes. This class is created for future use in credit card transaction but not

implemented anywhere in the application. Now, for test purpose, if this class is inherited by any other sub class, JSmell does not consider it as speculative generality code smell. This confirms the true positive result for test input.

```
package card.estateworld;
public abstract class AmountCard {
    private int amount;
    private int cardType;
    private String label;

    public AmountCard (String label, int amount, int cardType){
        this.label = label;
        this.amount = amount;
        this.cardType = cardType;
    }
    public int getCardType() {
        return cardType;
    }

    public String getLabel() {
        return label;
    }
}
```

FIGURE 27. Sample Input for Speculative Generality.

The result for speculative generality is shown in Figure 28. The *ImplementedAbstractClass* arraylist contains all the abstract clases which are implemented in the application. The algorithm detects the abstract classes not present in this arraylist. A*mountCard* class has been detected in this case.

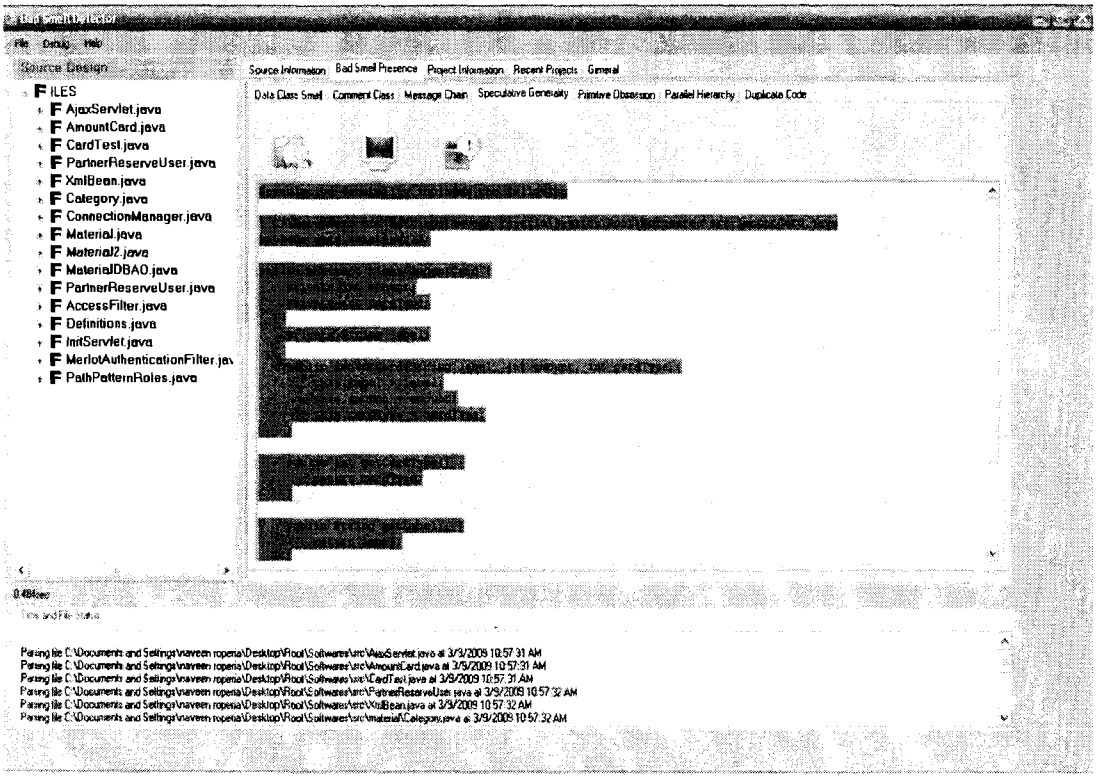

FIGURE 28. Output for Speculative Generality.

Input Validation for Parallel Inheritance Hierarchy

The presence of instance of parallel inheritance hierarchy is detected for the ANTLR IDE source code. This is an open source IDE containing 408 classes. This code smell detection process involves ensuring 3 things from the information extracted by JSmell. The number of levels in the inheritance hierarchy should be more than 2. At each level, the number of nodes must be the same. The depth of inheritance tree must be same. To verify the existence of parallel hierarchy, it is confirmed that the one parallel hierarchy does not instantiate the object of the other. The output for this code smell is shown in Figure 29.

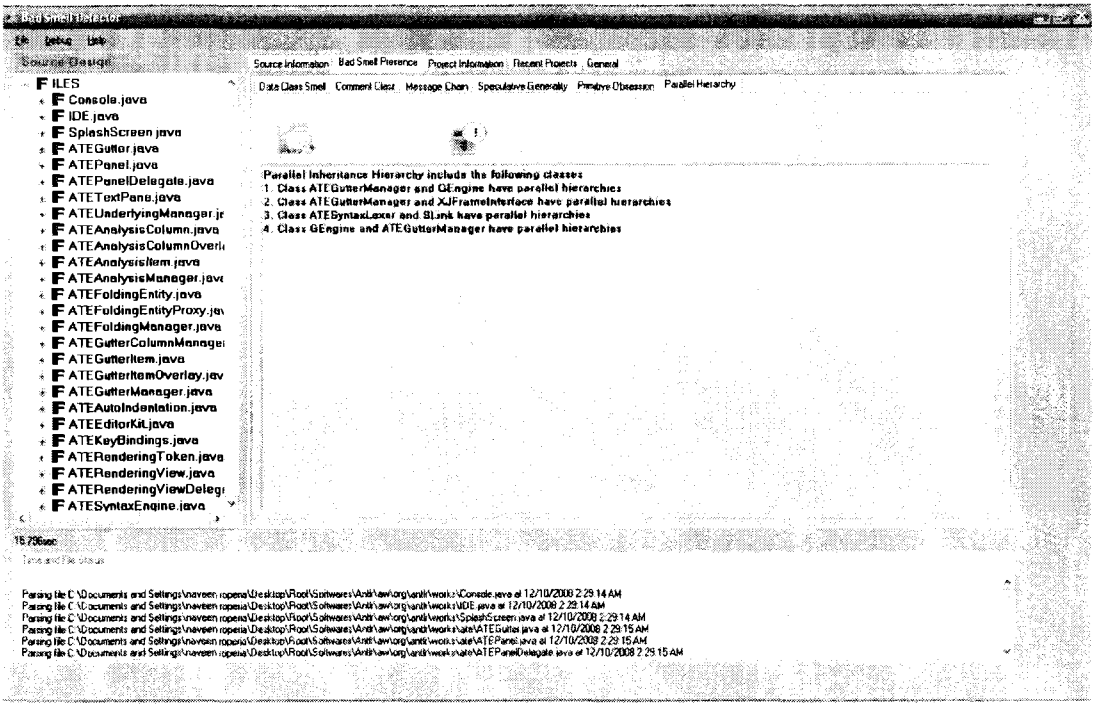

FIGURE 29. Output for Parallel Inheritance Hierarchy.

Input Validation for Comments

The detection approach in comments is straightforward. Figure 31 demonstrates the detection of comment code smell. This is a true positive case scenario. JSmell detects the presence of comment only in classes where the number of comments in the class exceeds the one greater than the number of methods in that class. This is because it is presumed that methods should be named as what they intend to. One comment is allowed for each method and one for the class.

Figure 30 shows instances of comment smell in the source code. Presence of more than one comment line for one method validates the presence of comment code smell.

```
public class AjaxServlet extends HttpServlet {
//int a,b,c,d=7,e,f;
    /** Processes requests for both HTTP <code>GET</code> and
<code>POST</code> methods.
     * @param request servlet request
     * @param response servlet response
     */
    protected void processRequest(HttpServletRequest req,
HttpServletResponse resp)
    throws ServletException, IOException {

        // Sets the content type made to xml specific
        response.setContentType("text/xml");
        response.setHeader("pragma","no-cache");
        response.setHeader("Cache-Control","no-cache");
        response.setHeader("Cache-Control","no-store");

       //Initializing PrintWriter
        PrintWriter out = response.getWriter();
        System.out.println("Came here in Ajax Servlet");

        String req = request.getParameter("req");
        System.out.print("This the values of Request:"+req);
        // Creating an instance of XmlBean Which generates XmlData From
Business Logic specified
        XmlBean xml = new XmlBean(req);

        // Method to Generate XMLDATA
        String buffer = xml.getXmlData(req);

        // If Close of DB connections are succefull then write the
content to the printstream

        if(xml.close() == true)
            out.write(buffer);

        out.flush();
        out.close();
        this.doGet(a,b);
    }
    protected void doGet(HttpServletRequest request,
HttpServletResponse response)
    throws ServletException, IOException {
// Calls a Method called processRequest Which Processes the request
which was made by the Browser Page
        processRequest(request, response);
    }
}
```

FIGURE 30. Input Sample for comment smell.

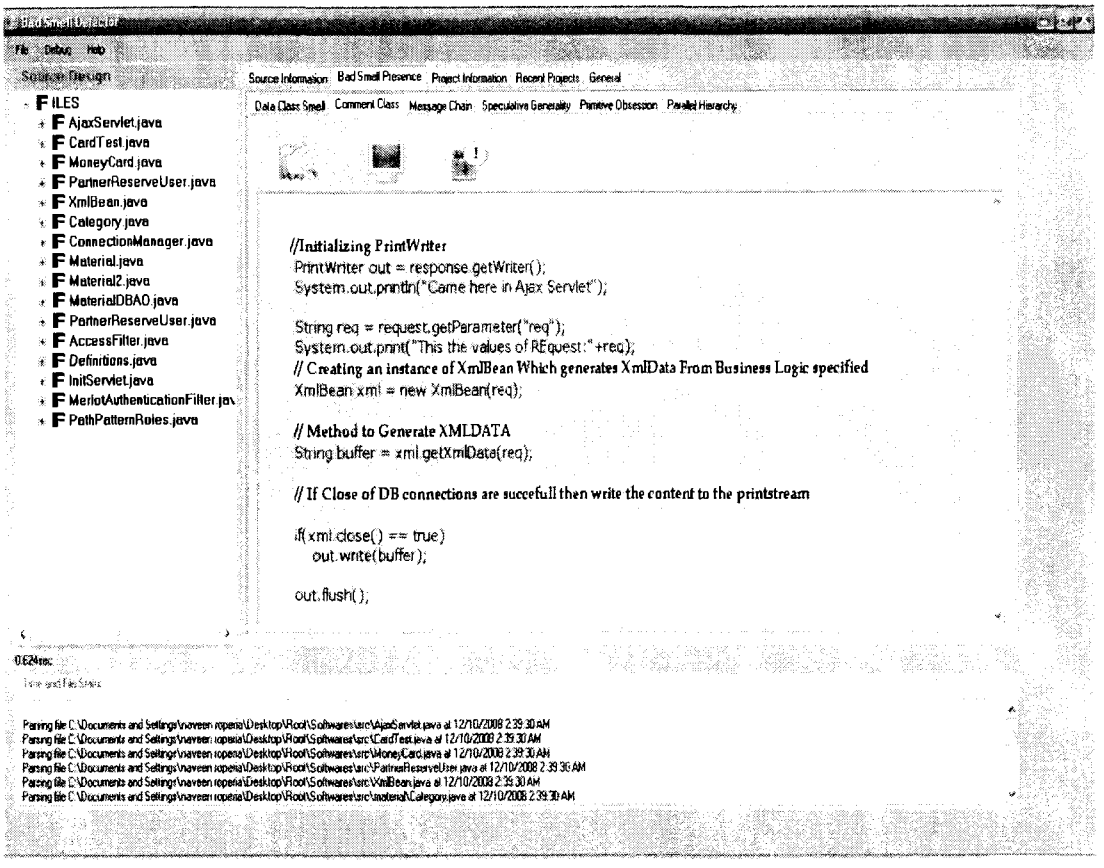

FIGURE 31. Output for comment smell.

Input Validation for Duplicated Code

The duplicated code detection scenario includes the web based project which contains 17 classes. A new class is added with some duplicity from an existing class. A new file, *CategoryTest*, is added which contains some duplicated code copied from class materialDBAO, Figure 24, and the *Category*, Figure 19. Figure 32 display the input source code from the file *CategoryTest*. This class contains four different variables and four different methods copied from *materialDBAO* class and four methods from class *Category*.

```
public class CategoryTest
{
      String view1;
      String view2;
      String view3;
      String view4;
public void setCategory(String category)  {
 this.category = category;
}
public void setCatid(int catid) {
 this.catid = catid;
}
public String getCategory()         {
return category;
}
public int getCatid() {
return catid;
}
//Copied from MaterialDBAO class
public String getView1(){
            return view1;
      }
  public String getView2(){
                  return view2;
            }
   public String getView3(){
                  return view3;
            }
   public String getView4(){
                  return view4;
            }
}
```

FIGURE 32. Input Sample for duplicated code smell.

The duplicated code detection process detects the duplicate code present in the *Category* and the *CategoryTest*. The duplicated code detection algorithm captures the same code copied between these two classes because it satisfies all conditions mentioned in the algorithm. Figure 33 shows the result of same code present in two different classes.

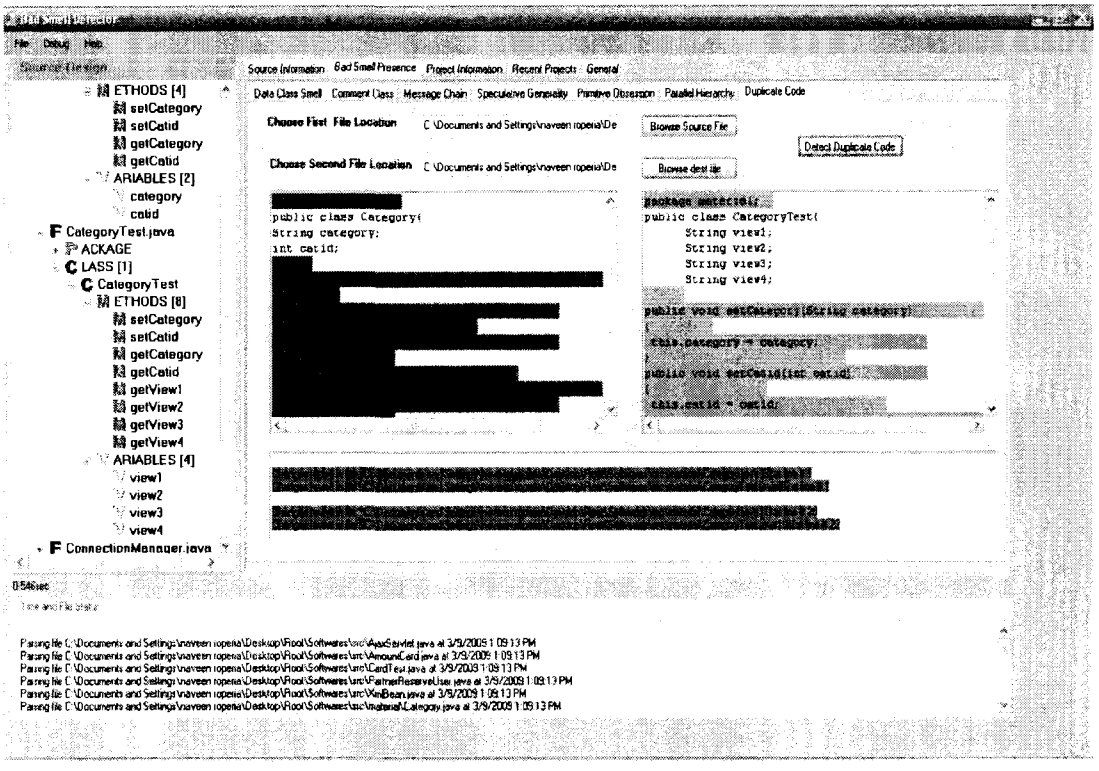

FIGURE 33. Output for Duplicated Code smell detection.

CHAPTER 8

CONCLUSION AND FUTURE WORK

Conclusion

The research work in this thesis has presented code smell detection processes to automatically detect seven different bad smells, using static analysis of any Object Oriented Java software system. All code smells are discovered using metrics and data storage techniques with the presumption that only one version of the system exists. The concepts of bad smell are defined in the beginning of the thesis and the previous research work has also been stated. We have improved algorithms from previous work in this field and have reduced the number of false positive results.

As the thesis establishes, JSmell successfully carries out the bad smell detection by means of analyzation of static information for example Number of Classes, Number of methods, Depth Inheritance Tree, Lines of Code and others.

Future Work

The current version of the JSmell detects code smell in Java systems only. Further this process can be enhanced by employing this bad smell detection for other languages such as C, C++, Python, and COBOL. Software metrics such as V_{avg}, NOV, DIT, LOC and NOC are widely applied in this detection process and is very helpful in future work for Object Oriented languages.

Additionally, this thesis currently detects only seven bad smells and can be enhanced to detect and suggest refactoring for other bad smells. Needless to say, if all the bad smells known are implemented for this tool, it would result in making it a powerful tool for enhancing the design of the Object Oriented Java software systems.

APPENDICES

APPENDIX A

ACCOUNT CLASS AS DATA CLASS SMELL

```
package edu.washington.cs.money;

public class BankAccount {

	private String accountName;
	private String accountType;


	public BankAccount(String name, String acc_type) {

		accountName = name;
		accountType = acc_type;
	}

	public void setAccountName(String name) {
		accountName = name;
			}

	public String getAccountName() {
		return accountName;
	}

	public void setAccountType(String acc_type) {
		accountType = acc_type;
			}
public String getAccountType() {
		return accountType;
			}
}
```

FIGURE 34. Account Class as an example of Data Class Smell.

APPENDIX B

PARALLEL INHERITANCE HIERARCHY DATABASE

TABLE 1. Class_ou_data

Field Name	Type	Description
Class_name	Varchar	Name of the class
Package_name	Varchar	Package name of the concerned class
Class_level	Int	Position of the class in a hierarchy, if exists
Root_class	Varchar	Root class in a hierarchy
Parent_class	Varchar	Parent of specific class
implementedInterface	Varchar	Name of the interface, if any

TABLE 2. Inner_ou_data

Field Name	Type	Description
class_name	varchar	Name of the class
Package_name	varchar	Package name of the concerned class
inner_class	varchar	Inner Class of the concerned class

TABLE 3. Select_duplicate

Field Name	Type	Description
Root_class	varchar	Root class in a hierarchy
class_level	int	Level of the concerned class
total _class	int	Total classes at specific level

REFERENCES

REFERENCES

[1] M. Fowler with contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts, *Refactoring: Improving the design of existing code*. Indiana: Addison-Wesley, 1999.

[2] Matthew James Munro, "Product Metrics for Automatic Identification of "Bad Smell" Design Problems in Java Source-Code," in *Proceedings of the 11th IEEE International Software Metrics Symposium*, 2005, pp.15.

[3] Mika Mantyla, Jari Vanhanen and Casper Lassenius, "A Taxonomy and an Initial Empirical Study of Bad Smells in Code," in *Proceedings of the International Conference on Software Maintenance*, 2003, pp. 381.

[4] Mika Mantyla, Jari Vanhanen and Casper Lassenius, "Bad Smells – Humans as Code Critics," in *Proceedings of the 20th International Conference on Software Maintenance*, 2004, pp. 399-408.

[5] Jim Whitehead, "Home page – Prof. Jim Whitehead," Jan. 2009. Available: http://www.soe.ucsc.edu/classes/cmps020/Winter09/lectures/refactoring-example.ppt. [Accessed Feb.1, 2009].

[6] R. Marinescu and D. Ratiu, "Quantifying the Quality of Object-Oriented Design: the Factor-Strategy Model," in *11th Working Conference on Reverse Engineering*, 2004, pp.192-201.

[7] Angelo Lozano, Michel Wermelinger and Bashar Nuseibeh, "Assessing the impact of bad smells using historical information," in *Ninth International Workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*, 2007, pp. 31-34.

[8] Miryung Kim, Vibha Sazawal, David Notkin and Gail C. Murphy, "An empirical study of code Clone genealogies," in *Proceedings of the European Software Engineering Conference*, 2005, pp. 187-196.

[9] D.L. Parnas, "Designing software for ease of extension and contraction," in *Proceedings of the 3rd International Conference on Software Engineering*, 1979, pp. 264-277.

[10] Embarcadero Technologies, "JBuilder 2008: Enterprise Java Development Just Got Easier," Sep. 23, 2008. [Online]. Available: http://www.codegear.com/products/jbuilder. [Accessed: Oct. 24, 2008].

[11] Ra'ul Marticorena, Carlos L'opez and Yania Crespo, "Parallel Inheritance Hierarchy: Detection from a static View of the System," in *6th International Workshop on Object Oriented Reenginering (WOOR)*, 2005, pp. 6.

[12] Marios Fokaefs, Nikolaos Tsantalis and Alexander Chatzigeorgiou, "JDeodorant: Identification and Removal of Feature Envy Bad Smells," *IEEE International Conference on Software Maintenance*, pp. 519-520, Oct. 2007.

[13] Theodoros Chaikalis, Nikolaos Tsantalis and Alexander Chatzigeorgiou, "JDeodorant : Identification and Removal of Type-Checking Bad Smells," *12th European Conference on Software Maintenance and Reengineering,* pp. 329-331, Apr. 2008.

[14] Mircea Lungu, Gabriel Ersze, Radu Marinescu and Petru Florin Mihancea, "Prodeoos," *LOOSE Research Group*, 2008. [Online]. Available: http://loose.upt.ro/prodeoos.html. [Accessed: Oct. 26, 2008].

[15] Naouel Moha, "Home page - Naouel Moha," DECOR: A Tool for the Detection of Design Defects, November 2007. [Online]. Available: http://www.naouelmoha.net/DECOR/DECOR.htm [Accessed: Nov. 29, 2008].

[16] K.J. Lieberherr and I.M. Holland, "Assuring good style for object-oriented programs," *IEEE Software*, vol.6, no.5, pp. 38-48, Sep. 1989.

[17] Chris Parnin, Carsten Gorg and Ogechi Nnadi, "A Catalogue of Lightweight Visualizations to Support Code Smell Inspection," in *Proceedings of the 4th ACM symposium on Software Visualization*, 2008, pp. 77-86.

[18] Kiel Hodges and Ron Jeffries, "You Aren't Gonna Need It," *xp.c2.com*, Nov.10, 2005. [Online]. Available: http://xp.c2.com/YouArentGonnaNeedIt.html. [Accessed: July. 14, 2008].

[19] Matthias Rieger and Stephane Ducasse, "Visual Detection of Duplicated Code*," *Workshop ion on Object-Oriented Technology*, p.75-76, July 1998. [Online]. Available: portal.acm.org [Accessed Oct. 20, 2008].

[20] Eugene W. Myers, "An O(ND) Difference Algorithm and Its Variations*," xmailserver.org, Algorithmica, vol. 1, pp. 251-266, 1986. [Online]. Available: http://www.xmailserver.org/diff2.pdf. [Accessed Feb. 25, 2009].

[21] Scott F. Smith, "Home page – Prof. Scott Smith," Oct. 2008. Available: http://www.cs.jhu.edu/~scott/oose/lectures/design.shtml. [Accessed Dec. 11, 2008].

[22] Arun Mukhija, "Estimating Software Maintenance," Requirements Engineering Resource Group, Institut für Informatik, Universität Zürich, Sem. Rep. WS02/03, Jan. 2003.

[23] C.Y. Baldwin, and K.B. Clark, "Design Rules: Volume1, The Power of Modularity," vol. 1, December 2005. [Online]. Available: http://www.people.hbs.edu/cbaldwin/DR2/BaldwinChinaPrefacev1_2.pdf. [Accessed Oct. 12, 2008].

[24] Jimmy Cerra, "Parallel Inheritance Hierarchy," Cunningham & Cunningham, Inc., 2004. [Online]. Available: http://c2.com/cgi/wiki?ParallelInheritanceHierarchies [Accessed Aug. 12, 2008].

[25] Joshua Kerievsky, *Refactoring to Patterns*. Indianapolis, IN: Addison-Wesley Signature Series, 2004.

[26] Matthias Hertel, "An O(ND) Difference Algorithm," October 08, 2008. [Online]. Available: http://www.mathertel.de/Diff/ViewSrc.aspx [Accessed Feb. 27, 2009].

[27] Stefan Slinger, "Code Smell Detection in Eclipse," M.S. Thesis, Delft University of Technology, Delft, Netherlands, 2005.

[28] Terrence Parr, *The Definitive ANTLR Reference Building Domain Specific Languages*. Version 2007-7-29 Texas: The Pragmatic Bookshelf, 2007.