FUNCTIONAL OVER-RELATED CLASSES BAD SMELL DETECTION AND REFACTORING SUGGESTIONS

Jiang Dexun[1], Ma Peijun[2], Su Xiaohong[3], Wang Tiantian[4]

School Of Computer Science and Technology, Harbin Institute of Technology,Harbin, China

ABSTRACT:

Bad phenomena about functional over-related classes and confused inheritances in programs will cause difficulty in programs comprehension, extension and maintenance. In this paper it is defined as a new bad smell Functional over-Related Classes. After the analysis, the characteristics of this new smell are transformed to the large number of entities dependency relationships between classes. So after entities dependency information collection and analysis, the bad smell is detected in programs, and corresponding refactoring suggestions are provided based on detection results. The experiments results of open source programs show that the proposed bad smell cannot be detected by current detection methods. The proposed detection method in this paper behaves well on refactoring evaluation, and the refactoring suggestions improve the quality of programs.

KEYWORDS:

Function over-Related; dependency relationship; bad smell detection; refactoring

1 INTRODUCTION

Good design quality of software will ease the maintenance and reusing, and reusability, flexibility, understandability, functionality and extendibility will be improved. On the other hand, if too many bad smells exist in programs, the quality of the software would be very low.

Bad smells [1] are signs of potential problems in codes. It causes difficulty for understanding and modifying of programs. Bad smells are not mistakes or defects in codes, but may cause them. Indeed bad smells are alerting of codes. So bad smells should be removed if found. Refactoring is a programming technique for optimizing the structure or pattern of an existing body of code by altering its internal nonfunctional attributes without changing its external behavior. On the premise of preserving observed behaviors, refactoring can improve the quality through increasing the understanding and decreasing the reuse cost of programs.

Inheritance is a fundamental feature of the Object Oriented (OO) paradigm. It is used to promote extensibility and reuse in OO systems. Oppositely, in OO programs, the two classes are designed without inheritance relationships, but they should have this inheritance. This situation may cause bad results. The changing of one class will not cause the change of the other class automatically.

DOI : 10.5121/ijsea.2014.5203

But in functional request, the two classes should change together for this modification. Thus it is a bad phenomenon, because it decreases the understandability and maintenance of the program.

In this paper, a new bad smell is defined to describe the bad phenomenon above. For this new bad smell, the characteristics, classification, impact and comparison to other similar and relevant bad smells are discussed. Based on the features of this bad smell, the detection method is proposed, and corresponding refactoring suggestion is provided.

The rest of the paper is organized as follows. Section 2 presents a short overview of related work. In Section 3, the expression and characteristics are described, and the new bad smell is defined. This bad smell is classified and detected, and the refactoring suggestion is provided in Section 4. Section 5 shows the experimental results and analysis. The conclusion is provided in Section 7.

2 RELATED WORK

2.1 Classification of bad smells

Before bad smell, program analyzer use anti-pattern for the potential problems in the programs. In [2-4], some of the bad smells in [1] are classified as anti-pattern. But Luo [5] discussed the difference and correlation of anti-pattern and bad smells. Anti-pattern is the problem in design level, and bad smells are in presentation level. Anti-pattern causes bad smells.

Fowler classified 22 kinds of bad smell, gave the description and improving process. Kerievsky [6] introduced 6 new bad smells, and Abebe [7] introduced 7 more.

Wake [8] simply classified bad smells as “inner class” and “outside class”. Mantyla [9] classified the 22 bad smells in [1]: the bloaters, the object-orientation abusers, the change preventers, the dispensable, encapsulators and others. The significance of classification is that bad smells are more understandable, the relations between bad smells are more obvious, and the researches of bad smell can be further.

Raul [10] analyzed these 6 kinds of bad smell from the aspects of Granularity, Intra vs Inter-relations, Inheritance and Access Modifiers, and dividing common bad smells with relevant metrics.

2.2 Bad smell detection and refactoring

Fowler [1] introduced bad smells and gave detecting and refactoring methods. But these processes should be achieved manually.

Hayden and Ewan [11] detected bad smells with dependency graphs. Their method can detect Feature Envy, Data Dump and other bad smells. This method has shortage, and the collection of dependency relationships is not completely right, such as duplication names of classes and fields. This causes limited precision in bad smell detection.

Tourwe and Mens [12] introduced a bad smell detection method independent from programming languages. They consider the key components of programs are expressions and principles, and the principles should be redacted manually. The disadvantage of this method is the principle transformation between program languages.

Atkinson and King [13] proposed syntactic based bad smell detection approach with low-cost. The judgment of this approach is too simple, and the recall rate is low.

Simon [14] proposed a metrics based visual approach to assist the software engineer to detect bad smells in programs. The cohesion of programs can be collected with metrics and reflected as visual images. But bad smells detection needs manual judgment.

There are many open source tools for bad smell detection, such as CheckStyle1, PMD2, iPlasma [15], JDeodorant [16] and so on. There tools are convenient, automatic, easy to obtain, and have high detecting speed. But just small kinds of bad smells can be detected, such as Duplicated Code, Data Class, Feature Envy, Large Class, Long Method and so on. Furthermore, the detection processes are simple. For example, for Duplicated Code, only text duplication is considered, for Large Class, fixed thresholds are used for detection. All of these causes lower precision of the detection results.

3 FUNCTIONAL OVER-RELATED CLASSES

3.1 Bad phenomenon in codes

In object oriented language, the functions of programs are stored in different methods of classes. Through the invoking and executing related methods, the functions of programs are achieved. If the same functions are achieved in different places, the same methods should be used. Furthermore, if the more similar the methods being used are, the more similar their functions are. Therefore, the similarity of entities invoking leads to that of functions.

One typical sample of bad smells in open source program HSQLDB2.2.7 is displayed in Figure 1. There are two classes in this sample, and they have no inheritance relationships, means they are not parent-child or brothers relationships. But they all invoke some entities outside the classes, and some of the entities are same, although the usages of the same entities are different. In class SchemaObjectSet, method getSQL() invokes attribute dataImpact in class Routine, and method addAllSQL() invokes attribute name and method schema(). In class ParserRoutine, method compileRepeat() invokes attribute name in class HsqlName, and method readLocalTableVariableDeclarationOrNull() invokes attribute name and method schema() in class HsqlName. Other entities are both invoked by these two classes SchemaObjectSet and ParserRoutine, but they are not listed here because of space cause.

[1] checkstyle.sourceforge.net
[2] pmd.sourceforge.net

```
public class SchemaObjectSet {
    String[] getSQL(OrderedHashSet resolved, OrderedHashSet unresolved) {
        ...
        if (type == SchemaObject.FUNCTION || type == SchemaObject.PROCEDURE) {
            ...
            while (it.hasNext()) {
                for (int i = 0; i < routineSchema.routines.length; i++) {
                    Routine routine = routineSchema.routines[i];
                    if (routine.dataImpact == Routine.NO_SQL|| routine.dataImpact == Routine.CONTAINS_SQL) {
                    ...
                    }
                }
            }
        }
        ...
    }
    static void addAllSQL(OrderedHashSet resolved, OrderedHashSet unresolved,
                          HsqlArrayList list, Iterator it,
                          OrderedHashSet newResolved) {
        while (it.hasNext()) {
            ...
            for (int j = 0; j < references.size(); j++) {
                HsqlName name = (HsqlName) references.get(j);
                switch (name.type) {
                    ...
                    case SchemaObject.COLUMN : {
                        if (object.getType() == SchemaObject.TABLE) {
                            int index = ((Table) object).findColumn(name.name);
                            ....
                        }
                    }
                    case SchemaObject.CHARSET :
                        if (name.schema == null) {
                            continue;
                        }
                    ...
                }
            }
        }
    }
}
```

```
public class ParserRoutine extends ParserDML {
    …
    private Statement compileRepeat(Routine routine,StatementCompound context,
                                    HsqlName label) {
        ...
        if (isSimpleName() && !isReservedKey()) {
            ...
            if (!label.name.equals(token.tokenString)) {
                throw Error.error(ErrorCode.X_42508, token.tokenString);
            }
            ...
        }
    }
    Table readLocalTableVariableDeclarationOrNull(Routine routine) {
        ...
        if (token.tokenType == Tokens.TABLE) {
            ...
            HsqlName name = super.readNewSchemaObjectName(SchemaObject.TABLE,false);
            name.schema = SqlInvariants.MODULE_HSQLNAME;
            ...
        }
        ...
    }
    Statement compileSQLProcedureStatementOrNull(Routine routine,
            StatementCompound context) {
        ...
        try {
            switch (token.tokenType) {
                case Tokens.OPEN : {
                    if (routine.dataImpact == Routine.CONTAINS_SQL) {
                        throw Error.error(ErrorCode.X_42602,
                                          routine.getDataImpactString());
                    }
                    ...
                }
                ...
                default :
                    return null;
            }
            ...
        }
    }
}
```

Figure 1 Bad phenomenon example existing in open source programs

If the number of the same entities invoking relationships is high in the total invoking relationships between two classes, theoretically the two classes should be designed in the same inheritance tree, or bad smells occur.

3.2 Definition of new bad smell

Definition 1 (Entity): the entity is the attribute a or the method m in one class, which is signed as E.

Definition 2 (Dependency relationship): if in a method $m1$ attribute a or method $m2$ is used, it is called $m1$ invokes a or $m2$. Maybe a or $m2$ is in the same class with $m1$, or the opposite. When entity $E1$ invokes entity $E2$, dependency relationship occurs between them.

The dependency relation contains two forms: one is to instantiate other classes, one is to parameterize other classes. The two forms are shown in Figure 2.

```
class class_A
{
    static int aA1;
    static int aA2;
    public static void mA1()
    {
        class_B b=new class_B();
        b.aB1=0;
        ...
    }
```

```
class class_A
{
    static int aA1;
    static int aA2;
    public static void mA1(class_B b)
    {
        b.aB1=0;
        ...
    }
```

Figure 2 Sample codes of dependency relationship

If methods invoke attributes or methods in same classes, the form should be “this.attrbute” or “this.method()”. If methods invoke attributes or methods in father classes, the form should be “super.attrbute” or “super.method()”.

The dependency relationship is represented as $I(E_1, E_2)$, and E_2 and E_1 are the entities. Invoking is directed, but dependency relationship is undirected, so $I(E_1, E_2) = I(E_2, E_1)$.

Dependency relationship is the undirected chain structure with two points.

Definition 3(Property set): The property set of an entity is the set of all the entities which have dependency relationships with this entity. The entity itself is the element of its property set. The property set of entity E is signed as $P(E)$.

Definition 4 (Dependency chain): Dependency chain is the chain structure composed of invoking relationships. If there are two dependency relationships $I(E_1, E_2)$ and $I(E_2, E_3)$, the length of chain $E_1 - E_2 - E_3$ is 2. In this chain $E_1 \neq E_3$.

Definition 6 (Function related): two classes are function related, if dependency relationship occurs between entities in respective classes.

Definition 6 (Function over-Related Classes): two classes are not in the same inheritance tree. If there are dependency relationships between the entities in these respective classes, and the rate α is high, the two classes are function over-related. And it is seen as a new bad smell, named "Functional over-Related Classes", FRC for short. And the equation of α is:

$$\alpha = \frac{\text{number of the entities with dependency relationships}}{\text{number of all the entities in the class}}$$

3.3 Features and comparison of FRC

3.3.1 What is functional related?

There are two food factories which separately produce chicken sausages and chicken tines. Different foods are in accordance with the tastes of different people, but the food materials are both live chicken. On the other hand, these two factories are "related" but not "same". Because of the same materials, the two factories can be considered to locate together, such as the chicken farms or places with convenient transportation.

The function related classes judging are based on whether they have dependency relationships between them, or they have dependency relationships to the entities in a third class. But they may invoke the entities for different using, and it is not considered in this paper.

3.3.2 Classification of FRC bad smell

FRC bad smell can be classified into direct related and indirect related.

（1）Direct related bad smell

There is dependency relationship between E_A and E_B, and E_A is in Class A, E_B is in class B. If this kind of invoking behaviors is frequent, and the share of the dependency relationships is high, it is considered to be direct related FRC bad smell. In practice there are two kinds of situations, one is that too many entities in Class A invoke entities in class B, as shown in Figure 3(a); the other is that entities in Class A invoke too many entities in class B, as shown in Figure 3(b).

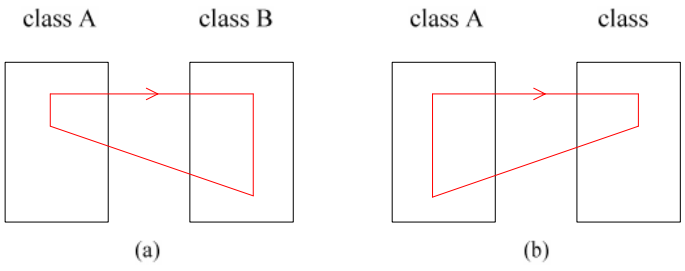

Figure 3 Direct related FRC bad smells

（2）Indirect related bad smell

There are dependency relationships between E_A and E_C, and also between E_B and E_C. E_C is in class C. So it is a dependency chain between class A and B, with the length of 2. If there are other dependency chain between class A and B, and the share is high, it is considered to be indirect related FRC bad smell, as shown in Figure 4.

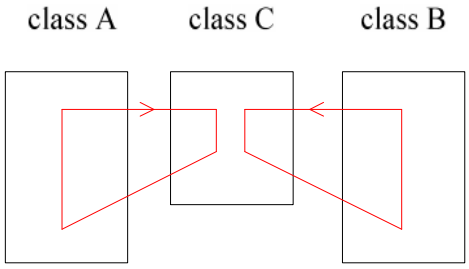

Figure 4 Indirect related FRC bad smell

In this paper the dependency chain with only length 2 is considered in indirect related bad smell. If the dependency chain is longer, the correlation between classes and entities is decreased.

3.3.3 Features and impacts of FRC bad smell

After analysis, the features are listed as follows:

(1)This bad smell occurs between two classes, and the dependency relationships are between entities form different classes. The two classes are named original class.

(2)The entities in one original class have dependency relationships with entities outside the class, or the entities in both of the original classes do.

(3)If there are dependency relationships between entity E_A in class A and entity E_B in class B, the intersection of entity E_A's property set and entity E_B's property set is not empty. For example, entity E_A in class A invokes entity E_C in class C, and entity E_B in class B also invokes E_C. Then $P(E_A) \cap P(E_B) = E_C \neq \varnothing$.

(4)The share of the element number in the intersection is high. The most extreme case is that all the entities in one class are invoked by entities in another class. And it is obviously a wrong design, for the class being invoked should be a superclass.

And the negative effects of FRC bad smell are:

(1) It is consumed that the FRC bad smell occurs at Class 1 and Class 2. When function request changes, the program should be modified at Class 1 and Class 2. The dependency relationship causes similar modifications at Class 1 and Class 2. And this will increase the workload of codes

modification. If Class 1 and Class 2 are in same inheritance hierarchy, the modification process would be simplified.
(2) If there is dependency relationship between Class 1 and Class 2, but they are not in the same inheritance hierarchy, this program will be difficult to understand, extend and reuse.

4 FRC BAD SMELL DETECTION AND REFACTORING

The detection of FRC bad smell is based on the analysis of this bad smell. The features of FRC bad smell is collected as the standard of static analysis and bad smell detection. The total algorithm of FRC detection is shown in Figure 5.

The inputs of the algorithm are: every class C_i in the programs being detected, the entities E_{ik} in C_i, and the two-dimensional array Inv recording all the dependency relationships of E_{ik}. The gain process of these input data is provided in [17], so the same will not be repeated here.

Algorithm: FRC bad smell detection
Input: C_i, E_{ik}, Inv
Output: FRC bad smell classes group (C_m, C_n)
Begin
  foreach (C_i)
    Threshold η_i of C_i in computed
    foreach (other class C_j in the program)
      the rate α_{ij} of C_i and C_j is computed
      if ($\alpha_{ij} > \eta_i$)
        C_i and C_j are pushed on the classes group(C_m, C_n)
      endif
    endfor
  endfor
End

Figure 5 Total algorithm of FRC bad smell detection

For different kinds of FRC bad smells, the detection and refactoring processes are not the same. The main approach of refactoring is adding new inheritance and changing existing inheritance. But other simple refactoring operations are also necessary for the rationality and readability of programs, such as changing names about the classes and entities.

4.1 Direct FRC

First the detection approach and refactoring suggestion about directed related FRC bad smell are represented. In some cases, indirect bad smell may disappear along with removing direct bad smell.

Just as shown in Figure 3(a), if some entities in class B is invoked by methods in class A, the rate ξ_1 is computed for bad smell detection:

ξ_1 = number of entities in class B which are invoked by methods in class A

And the thresholds should be added.

There are dependency relationships between entities. In the class if all the entities with dependency relationships are clustered, finally there are one or several clustering groups. For example, e_1, e_2 and e_3 are in one cluster group. If there is one dependency relationship between e_1 and e_4, e_4 is pushed on this cluster group. In class A, the number of all the entities is E_A, and the entities number of the largest cluster group is Y_A. Similar values in class B are E_B and Y_B.

The attributes and methods in classes may be private or protected. The number of private/protected attributes and methods in class A is M_A. Similar value in class B is M_B.

Then two thresholds should be computed for bad smell detection.

$$\eta_1 = \frac{{E_A}^2 \cdot (Y_B - Y_A)}{E_B \cdot (E_A - Y_A) - {E_A}^2}$$

$$\eta_2 = \frac{Y_A \cdot E_B + M_A \cdot E_B - E_A \cdot M_B}{E_A}$$

And $Y_B > Y_A$.

ξ_1 is compared with η_1 and η_2.

a) When $0 < \xi_1 < \eta_1$, there is no bad smells;

b) When $\eta_1 < \xi_1 < \eta_2$ and $Y_A + M_A < E_A$, FRC bad smell exists. And the refactoring operation is to extract relevant entities to create a new class, as the superclass of both class A and B. The situation is shown in Figure 6(a);

c) When $\xi_1 > \eta_2$, FRC bad smell exists. And the refactoring operation is to set class B as the superclass of A, as shown in Figure 6(b).

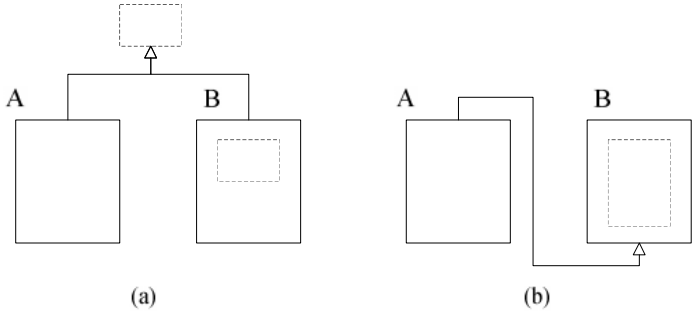

Figure 6 Refactoring suggestion of direct related FRC bad smell

4.2 Indirect FRC

The methods in class A and B both invoke the entities in class C, just shown as Figure 4. Then this rate should be computed:

ξ_2 =entities number invoke by methods in both A and B

The entities number in class A, B, C are separately E_A, E_B and E_C. The entities number of the largest cluster group is Y_A. The number of private/protected attributes and methods in class A is M_A. Similar values in class B are Y_B and M_B. Then two thresholds are computed.

$$\eta_3 = Y_A \cdot \frac{E_C}{E_A}, \quad \eta_4 = Y_B \cdot \frac{E_C}{E_B}$$

When $\xi_2 > \eta_3$, $\xi_2 > \eta_4$ and $Y_A + M_A + Y_B + M_B < E_A + E_B$, indirect FRC exists. Then the refactoring suggestion is to set class C as the superclass of class A and B, and it is shown in Figure 7.

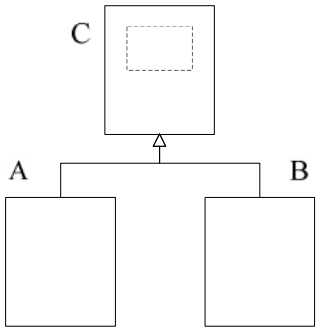

Figure 7 Refactoring suggestion of indirect related FRC bad smell

5 EXPERIMENTS AND ANALYSIS

5.1 FRC bad smell detection

According to the approach of FRC bad smell detection described in Section 4, a FRC bad smell detection tool FRC Detector is built in .net environment. Besides FRC bad smell detection, this tool can provide refactoring suggestions for existing FRC bad smell. The working process of FRC Detector is represented in Figure 8.

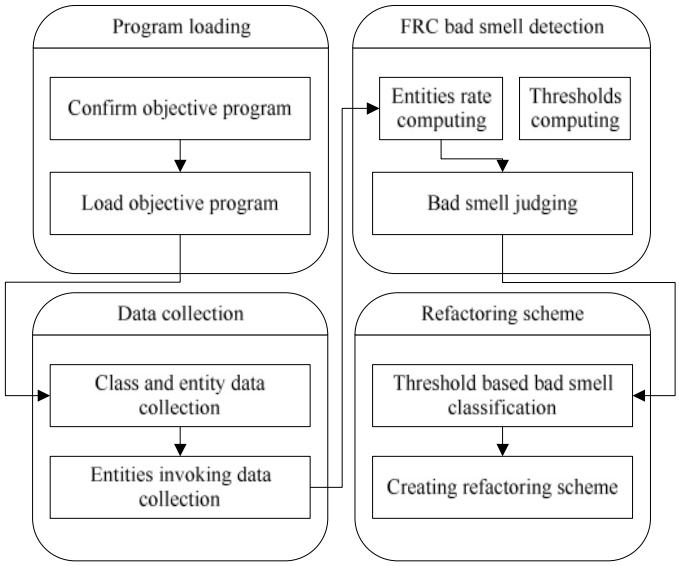

Figure 8 Working process of FRC Detector

Open source programs HSQLDB, Tyrant, ArgoUML and JFreeChart are the object programs for FRC bad smell detection. HSQLDB3is a Java database. The version is 2.2.8(2012-4-18), and the codes are over 130,000 lines. Tyrant4 is a game, and the version is 0.96. ArgoUML5 is a open source UML model tool, with the version 19912. JFreeChart6 is a free chart library for the Java platform, with the version 1.0.13.

Meanwhile, bad smells detection tools are used towards same programs. Smelly7 is used for the detection of Data Class, God Class, Long Parameter List; Code Bad Smell Detector8 is used for the detection of Data Clumps , Switch Statements, Speculative Generality, Message Chains, Middle Man; PMD9 is used for Duplicate Code; the method in [17] is used for Feature Envy; the

[3] hsqldb.org
[4] sourceforge.net/projects/tyrant
[5] argouml.tigris.org/source/browse/argouml
[6] sourceforge.net/p/jfreechart/code
[7] smelly.sourceforge.net
[8] cbsdetector.sourceforge.net
[9] pmd.sourceforge.net

method in [18] is used for Parallel Inheritance Hierarchy; FRC Detector proposed in this paper is used for FRC bad smell.

The experimental results are displayed in Table 1.

Table 1 Result of FRC bad smell detection

Detecting tools	Bad smells	Number of bad smells			
		HSQLDB	Tyrant	ArgoUML	JFreeChart
Smelly	Data Class	1	1	5	4
	God Class	0	0	7	3
	Long Parameter List	2	0	7	4
Code Bad Smell Detector	Data Clumps	1	2	0	12
	Switch Statements	6	4	32	15
	Speculative Generality	0	0	1	5
	Message Chains	3	4	9	8
	Middle Man	0	0	5	1
PMD	Duplicate Code	1	0	48	33
Method in [17]	Feature Envy	3	2	0	16
Method in [18]	Parallel Inheritance Hierarchy	1	0	1	1
FRC Detector	FRC	4	6	32	5

FRC bad smell cannot be detected by Smelly, Code Bad Smell Detector and PMD. Meanwhile, the programs without other bad smells still have FRC bad smells. Therefore, FRC bad smell cannot be detected by existing bad smell detection tools. And the improvements of other bad smells cannot remove the decrease of FRC bad smell in programs design quality.

5.2 Refactoring impact analysis

5.2.1 Refactoring suggestions of FRC bad smell

FRC Detector can provide corresponding refactoring suggestions for existing FRC bad smells. One suggestion is composed of one or more refactoring operations. Contents of the refactoring suggestions are created through the approach in this paper by FRC Detector, and the executing of the refactoring suggestions is achieved manually. The refactoring operations of partly existing FRC bad smells are shown in TABLE 2.

Table 2 Refactoring operations of existing FRC bad smells

Program	Class	Type	Refactoring operations
HSQLDB	*RangeIteratorBase* , *RangeVariable*	Direct	*RangeIteratorBase* is changed as the subclass of *RangeVariable*
ArgoUML	*StateBodyNotationUml*, *TransitionNotationUml*, *OperationNotationUml*	Indirect	Part of the entities are extracted a new class as superclass

The refactoring operations of all the existing FRC bad smells are not listed entirely because of space cause.

5.2.1 Program quality property computing

In this paper, the QMOOD (Quality Model for Object Oriented Design) model in paper [19] is used for refactoring impact analysis. The metrics of DesignSize, Hierarchies, Abstraction, Encapsulation, Coupling, Cohesion, Composition, Inheritance, Polymorphism, Messaging, and Complexity are extracted from the programs before and after refactoring. Through the QMOOD model, six program quality properties are computed.

According to paper [20], six properties about programs quality can be measured for total quality analysis:

- (Reusability)：Object oriented design characteristics that allow a design to be reapplied to a new program without significant effort
- (Flexibility)：Characteristics that allow the incorporation of changes in a design.
- (Understandability)：Properties of the design that enables it to be easily learned and comprehended, which relates to the complexity of the design structure.
- (Functionality)：Responsibilities assigned to the classes of a design, which are made available by the classes through their public interfaces.
- (Extendibility)：Characteristics refer to the presence and usage of properties in an existing design that allow for the incorporation of new requirements in the design.
- (Effectiveness)：Characteristics refer to the design ability to achieve the desired functionality and behavior using object oriented design concepts and technique.

The structure, relationships and functions of programs are reflected by these design quality properties. The description of design quality metrics are displayed in TABLE 3.

Table 3 Description and computation of design quality metrics

Design quality property	Description
DesignSize	The count of the total number of classes in the program
Hierarchies	The count of the number of class hierarchies in the program
Abstraction	Number of classes along all paths from the root classes to all classes in an inheritance structure.
Encapsulation	The ratio of the number of private/protected entities to the total number of entities declared in the class.
Coupling	Count of the different number of classes that a class is directly related to.
Cohesion	The summation of the intersection of dependency relationships of an entity with the maximum clustering set of all entities in the class.
Composition	The count of the number of data declarations whose types are user defined classes.
Inheritance	The ratio of the number of entities inherited by a class to the total number of entities accessibly by member methods of the class.
Polymorphism	The count of the methods that exhibit polymorphic behavior.
Messaging	The count of the number of public methods in a class
Complexity	The count of all the methods defined in a class.

According to paper [19], the values of design quality properties can be computed by design quality metrics. The formulas are listed in TABLE 4.

Table 4 Computing formula of design quality properties

Design quality property	Formula
(Reusability)[]	-0.25*Coupling+0.25*Cohesion+0.5*Messaging+0.5*DesignSize
(Flexibility)[]	0.25*Encapsulation-0.25*Coupling+0.5*Composition+0.5*Polymorphism
(Understandability)	-0.33*Abstraction+0.33*Encapsulation-0.33*Coupling+0.33*Cohesion-0.33*Polymorphism-0.33*Complexity-0.33*DesignSize
(Functionality)	0.12*Cohesion+0.22*Polymorphism+0.22Messaging+0.22*DesignSize+0.22*Hierarchies
(Extendibility)	0.5*Abstraction-0.5*Coupling+0.5*Inheritance+0.5*Polymorphism
(Effectiveness)	0.2*Abstraction+0.2*Encapsulation+0.2*Composition+0.2*Inheritance+0.2*Polymorphsm

The standard value of each design quality metric in the formulas is 1. Then the values of each design quality property are 1 expect the understandability property, and it is -1.

5.2.2 Quality properties results before and after refactoring

The real values of each design quality metric before and after refactoring are computed and shown in Table 5. And “Before” means the program before refactoring, and “After” means that after refactoring.

Table 5 Real values of each design quality metric before and after refactoring

Design quality metrics	HSQLDB		Tyrant		ArgoUML		JFreeChart	
	Before	After	Before	After	Before	After	Before	After
DesignSize	110.000	111.000	116.000	118.000	1873.000	1881.000	583.000	587.000
Hierarchies	5.000	6.000	1.000	1.000	12.000	24.000	9.000	13.000
Abstraction	1.088	1.102	1.817	1.860	1.507	1.610	2.253	2.299
Encapsulation	0.610	0.723	0.480	0.566	0.710	0.728	0.630	0.753
Coupling	6.209	5.789	4.399	3.669	3.281	3.135	5.276	4.799
Cohesion	0.228	0.255	0.199	0.220	0.263	0.275	0.260	0.266
Composition	2.103	1.986	1.112	1.195	2.111	2.184	0.496	0.541
Inheritance	0.318	0.336	0.530	0.546	0.349	0.371	0.065	0.065
Polymorphism	2.076	2.196	2.891	2.748	3.047	2.765	2.892	2.762
Messaging	33.000	35.000	37.000	39.000	22.000	22.000	33.000	36.000
Complexity	172.000	204.000	379.000	434.000	6551.000	7175.000	876.000	1002.000

The measure units of each design quality metric are not unified. For example, Design Size represents the number of classes in a program, so it is a value larger than 1 (actually it is always much larger than 1). But some metrics like Encapsulation and Inheritance are the ratio so the value is between 0 and 1. So the design quality metrics need standardizing for property computing. After simplification, the design quality metric before refactoring is set 1, and the design quality metric after refactoring is the ratio of original after value and before value. The results after standardization are displayed in Table 6.

Table 6 Standard values of each design quality metric before and after refactoring

Design quality metrics (standardization)	HSQLDB		Tyrant		ArgoUML		JFreeChart	
	Before	After	Before	After	Before	After	Before	After
Design Size	1.000	1.009	1.000	1.017	1.000	1.004	1.000	1.007
Hierarchies	1.000	1.200	1.000	1.000	1.000	2.000	1.000	1.444
Abstraction	1.000	1.013	1.000	1.024	1.000	1.068	1.000	1.020
Encapsulation	1.000	1.185	1.000	1.179	1.000	1.026	1.000	1.195
Coupling	1.000	0.932	1.000	0.834	1.000	0.955	1.000	0.910
Cohesion	1.000	1.116	1.000	1.103	1.000	1.045	1.000	1.023
Composition	1.000	0.944	1.000	1.074	1.000	1.034	1.000	1.091
Inheritance	1.000	1.057	1.000	1.032	1.000	1.063	1.000	1.012
Polymorphism	1.000	1.058	1.000	0.951	1.000	0.907	1.000	0.955
Messaging	1.000	1.061	1.000	1.054	1.000	1.000	1.000	1.091
Complexity	1.000	1.186	1.000	1.145	1.000	1.095	1.000	1.144

According to the formulas in Table 4, the values of design quality properties are computed and listed in Table 7.

Table 7 Design quality properties of each program

Design quality property	HSQLDB		Tyrant		ArgoUML		JFreeChart	
	Before	After	Before	After	Before	After	Before	After
Reusability	1.000	1.081	1.000	1.103	1.000	1.025	1.000	1.077
Flexibility	1.000	1.064	1.000	1.099	1.000	1.008	1.000	1.094
Understandability	-0.990	-0.956	-0.990	-0.887	-0.990	-0.976	-0.990	-0.930
Functionality	1.000	1.064	1.000	1.005	1.000	1.095	1.000	1.099
Extendibility	1.000	1.098	1.000	1.086	1.000	1.042	1.000	1.039
Extendibility	1.000	1.052	1.000	1.052	1.000	1.020	1.000	1.055
Amount	4.010	4.402	4.010	4.457	4.010	4.213	4.010	4.434

Take HSQLDB 2.2.8 as the example, the design quality properties changing is shown in Figure 9.

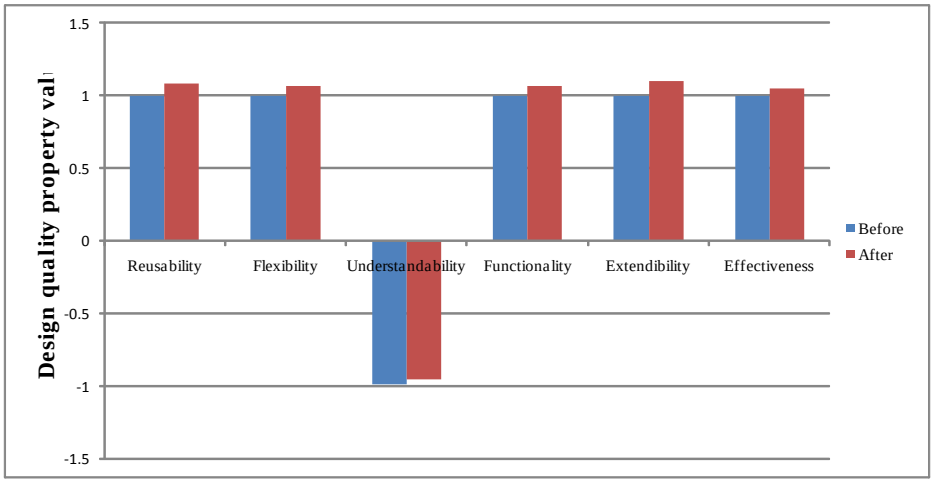

Figure 9 Design quality properties changing of HSQLDB 2.2.8

The refactoring suggestions of FRC bad smell are usually creating new classes then new inheritance between new and original classes, or adding new inheritance between existing classes. From Table 5, after refactoring the metrics of DesignSize, Hierarchies, Abstraction, Encapsulation, Cohesion, Inheritance and Complexity increased; Coupling decreased; the metrics of Composition, Polymorphism, Messaging are mainly changeless. From the formulas of Table 4, the changing of all the metrics above increases the value of quality properties except Understandability property. The increasing of Design Size, Abstraction and Complexity would decrease Understandability. But in another hand the removing of FRC bad smell increases Cohesion and decreases Coupling, so totally the value of Understandability is increased.

Thus, after FRC bad smell detection and corresponding refactoring operation, the values of each design quality properties are increased, so the total quality of the programs would be improved.
In large scale codes of over 300,000 lines and 1873 classes (ArgoUML), FRC bad smell is detected to exist in 32 places. Just executing these 32 refactoring operations (about 108 relevant classes should be changed), the quality standard value of the program is changed from 4.010 to 4.213, and the increasing percentage is about 5%. It is seen that the cost of FRC refactoring is lower, and benefit is higher.

6 CONCLUSION

In open source programs, the phenomena of function over related between classes without inheritance relationships, express the essence of frequently invoking between entities of classes. Actually, some public functional sources (data or operations) can be used by every member, and this is achieved by inheritance in program design. The classes which should have inheritance relationships but have no relationships, may cause the difficulty of understanding and maintenance, and decrease the program quality.

In this paper, the bad code phenomena above are defined as a new bad smell called FRC. This bad smell exists much in codes and decreases the quality of programs. It is necessary to study the detecting method and refactoring suggestion for this bad smell.

In this paper, the detection approach of FRC bad smell is proposed. The basic process is to collect the dependency relationships between classes, and compute the invoking rates and compare with dynamic thresholds. Finally the bad smell is judged to exist or not. Furthermore, according to the detection results, refactoring suggestions are provided.

The limitation of this paper is the thresholds in FRC bad smell detection. The preset thresholds decrease the veracity of detection results. In this paper dynamic thresholds are used for bad smell detection, and different collection and computing process of thresholds are executed according to the different situation of programs. Therefore the subjectivity is lower, and the detection is more accurate.

ACKNOWLEDGEMENT

This research is supported by the National Natural Science Foundation of China under Grant No.61173021 and the Research Fund for the Doctoral Program of Higher Education of China (Grant No. 20112302120052).

REFERENCE

[1] Fowler M, Beck K. Refactoring: improving the design of existing code. Addison-Wesley Professional. 1999.
[2] Abbes M, Khomh F, Gue W G, Antoniol, G. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. in Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on. IEEE, march 2011, pp:181 -190
[3] Counsell S. The ‘deception’ of code smells: An empirical investigation. International Conference on Information Technology Interfaces(ITI). 2010, pp:683-688
[4] Counsell S, Hierons RM, Hamza H, Black S, Durrand M. Exploring the eradication of code smells: An empirical and theoretical perspective. In: Proc. of the Advances in Software Engineering, 2010.
[5] Luo Yixin. An Ontological Identification of Relationships between Anti-Patterns and Code Smells. Aerospace Conference. 2010, pp:1-10
[6] J. Kerievsky. Refactoring to patterns. Addison-Wesley , 2004
[7] Abebe S L, Haiduc S, Tonella P, Marcus A. Lexicon bad smells in software. in Proc. Working Conf. on Reverse Engineering. IEEE, 2009, pp: 95–99
[8] Wake W C. Refactoring Workbook. Addison-Wesley, 2003.
[9] Mantyla M. Bad smells in software: a taxonomy and an empirical study. Ph.D. dissertation, Helsinki University of Technology, 2003.
[10] Raul M, Carlos L, Yania C. Extending a Taxonomy of Bad Code Smells with Metrics. WOOR’06, Nantes, 4th July, 2006.
[11] Hayden M, Ewan T. Identifying refactoring opportunities by identifying dependency cycles. In Proceedings of the 29th Austrasian Computer Science Conference - Volume 48, ACSC '06. 2006.
[12] Tom Tourwe and Tom Mens. Identifying refactoring opportunities using logic meta programming. In Proceedings of the 7th European Conference on Software Maintenance and Reengineering(CSMR '03), Benevento, Italy, March 26{28, pages 91{100. IEEE Computer Society, Los Alamitos, California, March 2003.
[13] Atkinson, D.C., King, T., 2005. Lightweight detection of program refactorings. In: Proceedings of the 12th Asia–Pacific Software Engineering Conference. IEEE CS Press, Taipei, Taiwan, pp:663–670.
[14] Simon, F., Steinbr, F., Lewerentz, C., 2001. Metrics based refactoring. In: Proceedings of the 5th European Conference on Software Maintenance and Reengineering. IEEE CS Press, Lisbon, Portugal, pp:30–38.

[15] Cristina M, Radu M, Petru M, Daniel R. iPlasma: An integrated platform for quality assessment of object-oriented design. In: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 2005). IEEE CS Press, Budapest, Hungary, pp: 77–80.

[16] Nikolaos Tsantalis, Alexander Chatzigeorgiou. Identification of extract method refactoring opportunities for the decomposition of methods. Journal of Systems and Software, 2011, 84(10):1757–1782.

[17] Jiang Dexun, Ma Peijun. Detecting Bad Smells with Weight Based Distance Metrics Theory. Proc. of 2nd International Conference on Instrumentation, Measurement, Computer, Communication and Control (IMCCC). 2012:299-304

[18] Marticorena, C López, Y Crespo. Parallel Inheritance Hierarchy- Detection from a Static View of the System. 6th International Workshop on Object Oriented Reengineering (WOOR), Glasgow, UK. 2006,6.

[19] Jagdish Bansiya, Carl G Davis. A Hierarchical model for object-oriented design quality assessment. IEEE Transactions on Software Engineering.2002, 28(1):4-17.

[20] IEEE. Standard Glossary of Software Engineering Terminology 610. 12-1990, Vol. 1. Los Alamitos: IEEE Press , 1999

.

INTENTIONAL BLANK