Specification and Detection of Code Smells using OCL

Tae-Woong Kim[1], Tae-Gong Kim[2], Jai-Hyun Seu[2,1]

[1] School of Computer Engineering, Inje University,
Obang-dong 607, Gimhae, Gyeong-Nam, Korea,
ktw.maestro@gmail.com
[2] School of Computer Engineering, Inje University,
Obang-dong 607, Gimhae, Gyeong-Nam, Korea,
{ktg,jaiseu}@inje.ac.kr

Abstract. The words Code smells mean certain code lines which makes problems in source code. It also means that code lines in bad design shape or any code made by bad coding habits. There are a few studies found in code smells detecting field. But these studies have their own weaknesses which are detecting only particular kinds of code smells and lack of expressing what the problems are. In this paper, we newly define code smells by using specification language OCL and use these models in auto detection. To construct our new code smells model, we first transform java source into XMI document format based on JavaEAST meta-model which is expanded from JavaAST meta-model. The prepared OCL will be run through OCL component based on Eclipse plug-in. Finally the effectiveness of the model will be verified by applying on primitive smell and derived smell in java source code.

Keywords: Code Smells, Refactoring, OCL(Object Constraint Language), AST(Abstract Syntax Tree), Meta-model

1 Introduction

Refactoring is a kind of method to improve program behavior which includes program performance, structure, maintenance, and appearance without any changes to its original functionality. This means that it changes inner structures of source code to make it easy on maintenance and improve readability but keeps all functionalities in system at the same time[1]. To reform existing program design, we better find out what should be improved before applying refactoring to its original design[3]. Martin Fowler and Kent Beck suggested a method to discriminate certain design problems from bad smells in source code[2]. They express design problems as smells metaphorically. They merely explained which refactoring method is good to remove specific smell. A few studies were introduced to determine which refactoring method is appropriate to apply for detecting a certain code smells[3,4,5,6,7]. But these studies

[1] Corresponding author. Tel.: +82 55 320 3348; fax: +82 55 322 3107.
E-mail address: jaiseu@inje.ac.kr (J.H. Seu)

have their own weaknesses. They can not only detect limited kinds of code smells but express insufficiently for the detected code smells.

In this paper, we define code smells by using OCL[8] and use them for auto detecting. To realize this, we first transform java source code into XMI(XML Metadata Interchange)[10] format based on JavaEAST(Java Extended Abstract Syntax Tree) meta-model which is expanded from JavaAST[9] meta-model. The newly defined code smells model is made up in OCL. The prepared OCL code model will be run through OCL component(API for parsing and evaluating OCL constraints and queries)[11] based on Eclipse plug-in for auto detecting.

In chapter 2, we look into kinds of code smells and its definition. In chapter 3, we introduce a way how to detect code smells as proposed and implemented in this paper. In chapter 4, the proposed method will be verified through the example running. Finally in chapter 5, we make results and carefully suggest future study.

2 Bad Smells in Code

The word code smells mean certain code lines at any point in source code that makes problems. It is other than syntax error or warning when its' compiling. It also means that code lines in bad design shape or any code made by bad coding habits. It causes increasing expenses for software development when adding new functions or changing platform. It may also decrease efficiency of whole system. Code smells are defined for two kinds as follow.

- Primitive smell : simple smell can be detected from one class
- Derived smell : derived smell can be detected from relations between classes

Derived smell can be detected from information extracted from relation between several classes(inheritance, instances, usage of methods or fields, etc.). The following section explains studies about how to detect code smells.

3 Specification and Detection of Code Smells

The following requirements should be satisfied to detect code smells from source code. First, all syntax or grammar information of source code should be represented. Second, it should be easy to access and extract semantic information from represented source code model. Finally, it should be easy to extract relational information between classes. To satisfy these requirements, we propose following figure 1 which represents that transforming java source code into JavaEAST model and specifying code smells by using OCL and detecting method.

3.1 JavaEAST(Java Extended Abstract Syntax Tree)

The grammar information of source code will be presented as tree structure when java source code is transformed into JavaAST. It also produces a XML context for each java source code. But there is a problem to detect derived smell to analyze relationships between classes because it has only grammatical information. Consequently we propose extended AST model as figure 1. This is including binding information about fields.

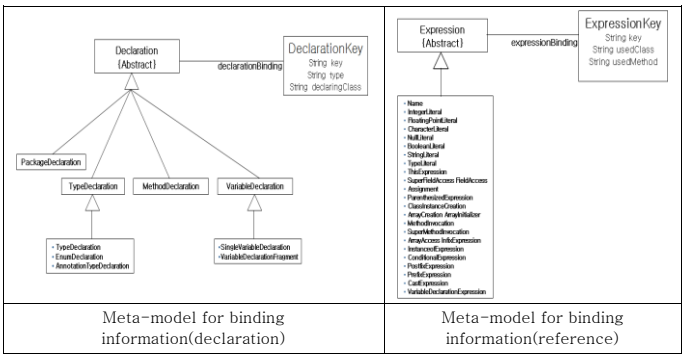

Fig. 1. JavaEAST meta-model to analyze relations between classes

3.2 Code Smells specification by using OCL

We can specify the definition of Long Parameter List among primitive smell by using OCL as follows.

```
self.parameters->notEmpty()
self.parameters->size() > maximum
```

But derived smell is gotten by analyzing relations between classes other than primitive smell. Therefore we extract this kind of information with proposed JavaEAST model. For example, we should find out the name of class, and declared properties and methods, and extracted classes in inheritance relations to detect 'Refused Bequest' among derived smell. The extractable information from relationships between classes will be represented in OCL as table 1.

Table. 1. OCL Code for Analyzing Relationships between classes.

OCL statements	meaning
context Project def:getSuperClasses() :Set(TypeDeclaration) =self.compilationUnits.types->collect(oclAsType(TypeDeclaration))- >select(subclassTypes->notEmpty())->select(bodyDeclarations- >select(oclIsTypeOf(FieldDeclaration))->notEmpty())->asSet()	Getting super class
def: getUsedClassOfSimpleName(fKey : String) : Set(String) = SimpleName.allInstances()->select(expressionBinding->notEmpty()) ->select(expressionBinding.key = fKey).expressionBinding.usedClass->asSet()	Getting name of the class that is used variables with variable key value

4 Applied example

The 'Refused Bequest' among derived smell does not use fields or methods from super class at derived class. In other words, inherited sub class refuses to be inherited with particular fields or methods from super class. Relationship between classes and fields should be formed as figure 2 to detect 'Refused Bequest' code smells.

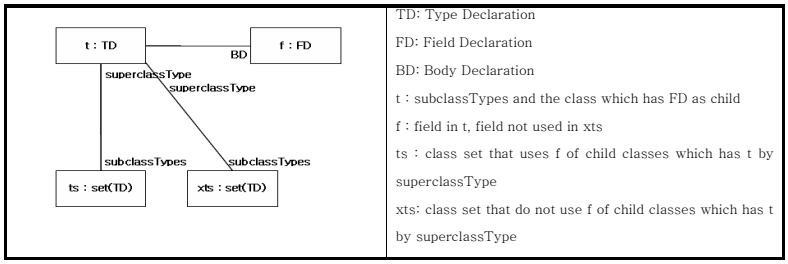

Fig 2. Specification to detect 'Refused Bequest' code smells

It finds class t and field f which is satisfied following conditions as represented context in figure 2.

A. Finding super class t which has children class and field
B. Finding f which is satisfied following conditions in super class t
   a) The fields should not be used in t
   b) The fields should not be used in xts
   c) The fields should not be used by using t
   d) The fields should not be used by using xts
   e) It should not be overriding in ts

If we specify conditions like A and B by using OCL as follows.

```
context Project
def : getRefusedBequestDetection():
Sequence(Tuple(superClass : String, field : String, subClass : String)) =
self.getSuperClasses()->collect(sType |sType.getFieldDeclarationKey()->collect( sFieldKey |
if self.getUsedClassOfSimpleName(sFieldKey)->includes(sType.getTypeKey) then null
else if self.getTypeOfQualifiedName(sFieldKey)->includes(sType.getTypeKey) then null
else if self.getUsedClassOfSimpleName(sFieldKey)->includesAll(self.getSubclassTypeKey(sType)) then null
else
  (self.getUsedClassOfSimpleName(sFieldKey)->intersection(self.getSubclassTypeKey(sType))
  ->union(self.getTypeOfQualifiedName(sFieldKey))-self.getSubclassTypes(sType)
  ->iterate(acc:TypeDeclaration ;
  result : Set(String) = Set{} | if acc.getFieldNames()->includes(self.getVariableName(sFieldKey)) then
    acc.getTypeKey->union(result) else result endif
  ))->collect(target | Tuple{superClass = sType.getClassName(), field = self.getVariableName(sFieldKey),
  subClass = self.getClassNameByKey(target)}) endif endif endif
))->excluding(null)->asSet()->asSequence()
```

If we apply above OCL code to source code as represented class diagram in figure 3 and execute it by using OCL component then we get result as figure 4.

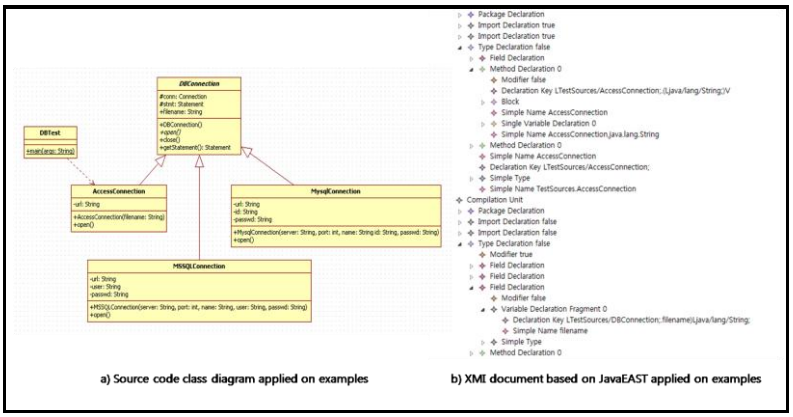

Fig 3. The only class 'AccessConnection' is using filename among 'DBConnection' classes

We assume that only 'AccessConnection' uses the filename field of 'DBConnection' among classes inherited 'DBConnection' shown in figure 3a. In this case, 'Refused Bequest' occurs. In other words, classes other than 'AccessConnection' refuse to inherit filename which is declared in super class. Figure 3b shows that XMI document which is transformed from figure 3a class diagram and expressed by JavaEAST model base which was proposed in this paper. Figure 4 shows the result of 'Refused Bequest' smell detection which OCL is applied on figure 3b model. The red box in figure 4 shows detection result of 'AccessConnection' which is the only used sub class that refuse to inherit filename field from super class 'DBConnection'.

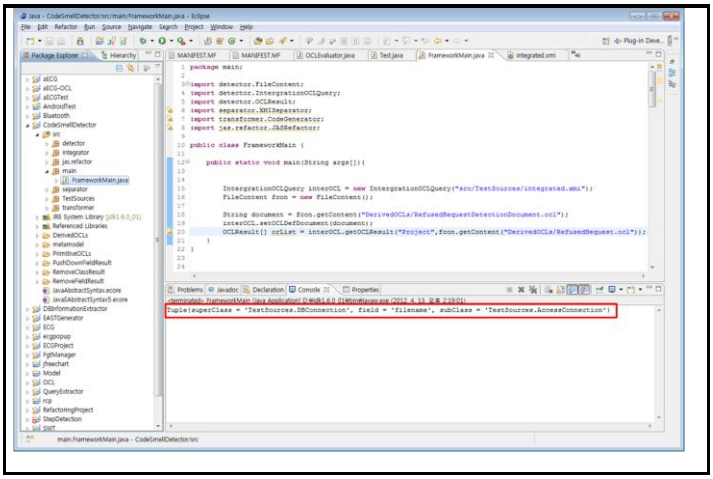

Fig. 4. It makes output Tuple as result of detecting 'Refused Bequest' code smells.

5 Conclusion

The few studies were introduced how to find out certain code smells. But these studies have their own weaknesses. They only detect limited kinds of code smells and represent lack of expression for the detected code smells. In this paper, we precisely specify code smells by using OCL and studied how to detect them automatically by running OCL component. Especially for specifying and detecting derived smell, we proposed JavaEAST model and introduced a way how to detect them. It is not only specifying bad smells in code, but also useful. In case of adding new code smell or need to modify definition of existing code smell in the future, just defining a new OCL could be enough to handle for new code smell. Moreover specification of derived smell could be reusable because it was generated from combined OCL definition which is constructed from extracting information among already defined classes.

This is a very important preceding study to develop more flexible reverse engineering tools. In the future, we need to specify and define OCL for various kinds of code smells. As a result, the studies about developing automated refactoring tools will be proceeded by using already detected and defined code smells.

References

1. Fowler, M.: Refactoring: Improving the Design of Existing Programs. Addison-Wesley (1999)
2. Fowler, M.: Refactoring: Improving the Design Existing Code. Addison Wesley (1999)
3. Slinger, S.: Code Smell Detection in Eclipse. Delft University of Technology (2005)
4. Kataoka, Y., Michael D.E., Griswold, W.G., Notkin, D.: Automated Support for Program Refactoring using Invariants. In Proc. Int. Conf. on Software Maintenance, pp.736--743. IEEE Computer Society Press (2001)
5. Ducasse, S., Rieger, M., Demeyer, S.: A language independent approach for detecting duplicated code. In: Yang, H., White, L. editors, Proc. Int'l Conf. on Software Maintenance, pp. 109--118. IEEE Computer Society Press (1999)
6. Simon, F., SteinbruUckner, F., Lewerent, C.: Metrics Based Refactoring. In Proc. 5th European Conference on Software Maintenance and Reengineering, pp.30--38. IEEE Computer Society Press (2001)
7. Murphy, H. E., Black, A. P.: An Interactive Abbient Visualization for Code Smells. ACM SOFTVIS '10 Proceedings of the 5th international symposium, pp.5--14. Software visualization (2010)
8. Object Management Group, Object Constraint Language Specification, Version 2.0, http://www.omg.org/technology/documents/formal/ocl.htm
9. MoDisco, MoDisco Tool- Java Abstract Syntax Discovery Tool, http://www.eclipse.org/gmt/modisco/toolBox/JavaAbstractSyntax/
10. Object Management Group, MOF 2.0/XMI Mapping Specification, V2.1.1, http://www.omg.org/technology/documents/formal/xmi.htm
11. Eclipse, OCL for EMF, http://www.eclipseplugincentral.com/Web_Links-index-req-viewlink-cid-200.html#, updated 2004