Specification and Automated Detection of Code Smells using OCL

Tae-Woong Kim[1], Tae-Gong Kim[2] and Jai-Hyun Seu[3]

School of Computer Engineering, Inje University,
Obang-dong 607, Gimhae, Gyeong-Nam, Korea

[1]ktw.maestro@gmail.com, [2]ktg@inje.ac.kr, [3]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 [3-7]. But these studies 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 analyze existing studies and point out the problems about related studies. 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. Related Studies

In this chapter, we look into kinds of code smells and its definition. In addition, we analyze existing studies about detecting code smells.

2.1. Bed 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. Table 1 shows such a example [3].

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

Table 1. Examples of Code Smells Type and Kind

Type	Kind	explanation
Primitive smell	Long Parameter List	Too many numbers of parameters
	Switch Statements	It may causes redundant
	Too Many Fields	There are too many fields
	Message Chain	It may needs go through too many classes to use particular class
Derived smell	Refused Bequest	Properties and methods in super classes are not used in sub classes
	Feature Envy	Methods are executed with data from other 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.

2.2. Other Studies

Van Emden's study [12]. Its' study about JCosmo which is used as standard tool to detect code smells in java source code. It inspects the java context to detect code smells. This method cannot be used to detect code smells from java structure or induced code smells. This study results some weakness that software should be constructed again from the beginning when code smells are add.

Richard C. Holt's study [13]. This study is about detecting code smells by using script language named Grok as known as related algebra tool based on information extracted from source code. It gathers facts from variety of information included in source code. These facts will be stored in facts repository. The code smells will be detected from this repository by using related algebra tool. The weakness in this study is that facts should be gathered and facts repository will be reconstructed again when source code is modified.

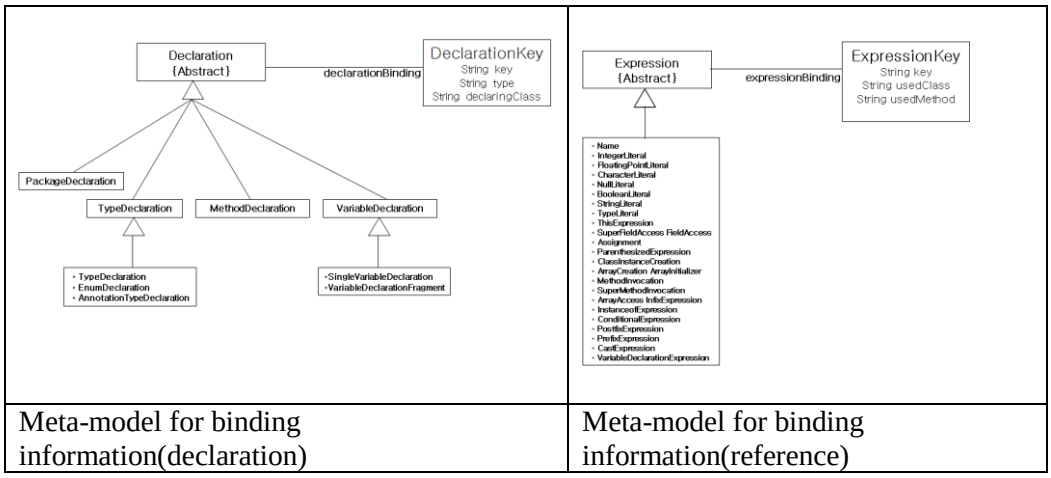
Stefan Slinger's study [3]. He developed the code smells detecting tool as plug-in based on eclipse. This study proposed that program should be coded by using proposed five steps. Especially in case of derived code smell, it is detected by using script language Grok after gathering facts like Richard C. Holt's study. This will result weakness that complex plug-in should be developed additionally and facts repository will be reconstructed again when new code smells are defined.

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.



Meta-model for binding information(declaration)	Meta-model for binding information(reference)

Figure 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 2.

Table 2. OCL Code for Analyzing Relationships between Classes

OCL structure	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
context TypeDeclaration def:getFieldNames():Set(String) =self.bodyDeclarations->select(oclIsTypeOf(FieldDeclaration))->collect(oclAsType(FieldDeclaration)).fragments.name.fullyQualifiedName->asSet()	Getting names of fields from current class
context Project 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.

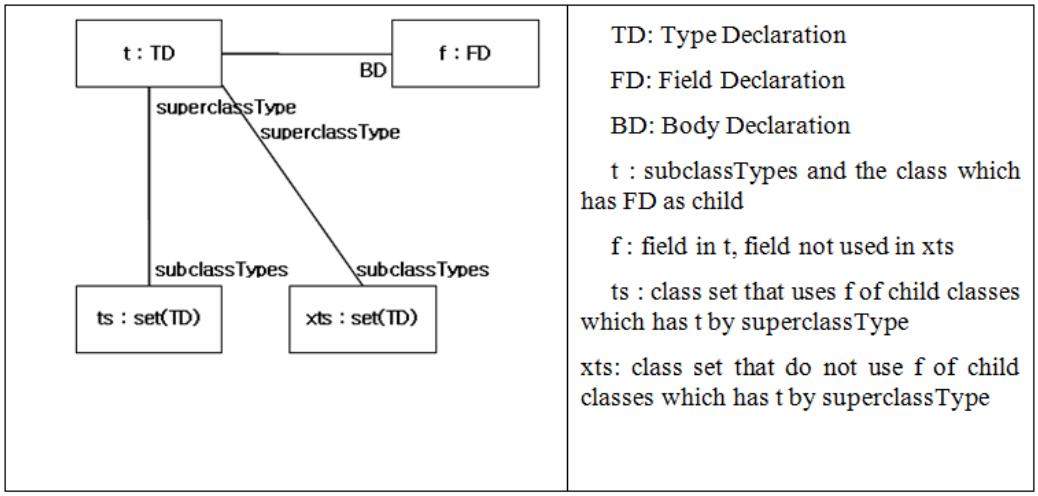


Figure 2. Specification to Detect 'Refused Bequest' Code Smells

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, they will be 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.

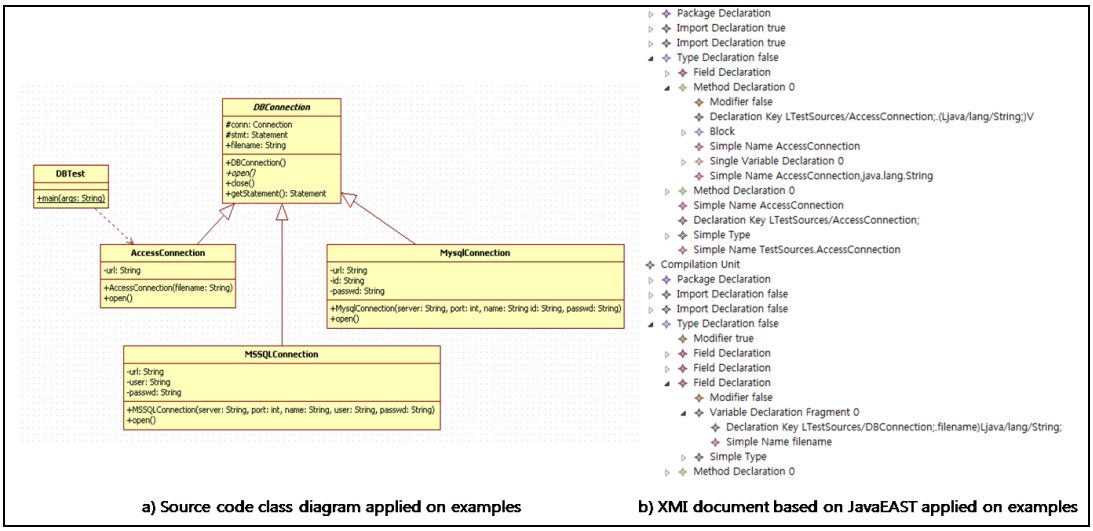


Figure 3. The Only Class 'AccessConnection' is using Filename among 'DBConnection' Classes

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’.

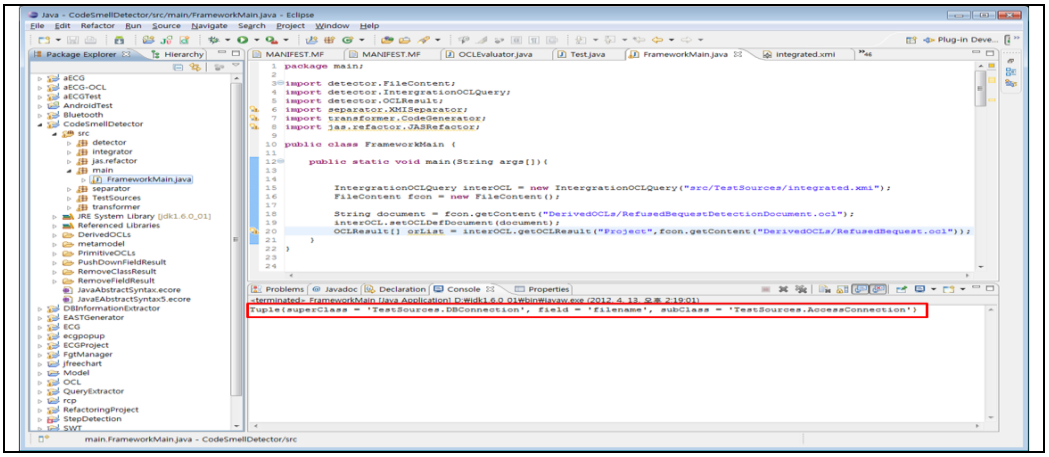


Figure 4. It Makes Output Tuple as Result of Detecting ‘Refused Bequest’ Code Smells

We apply the value obtained from execution result of Figure 4 to PushDownField method which was built in our research for real refactoring. The PushDownField method is associated with CreateField and RemovedField methods which are create and remove fields. It also includes OCL to check existing class and fields.

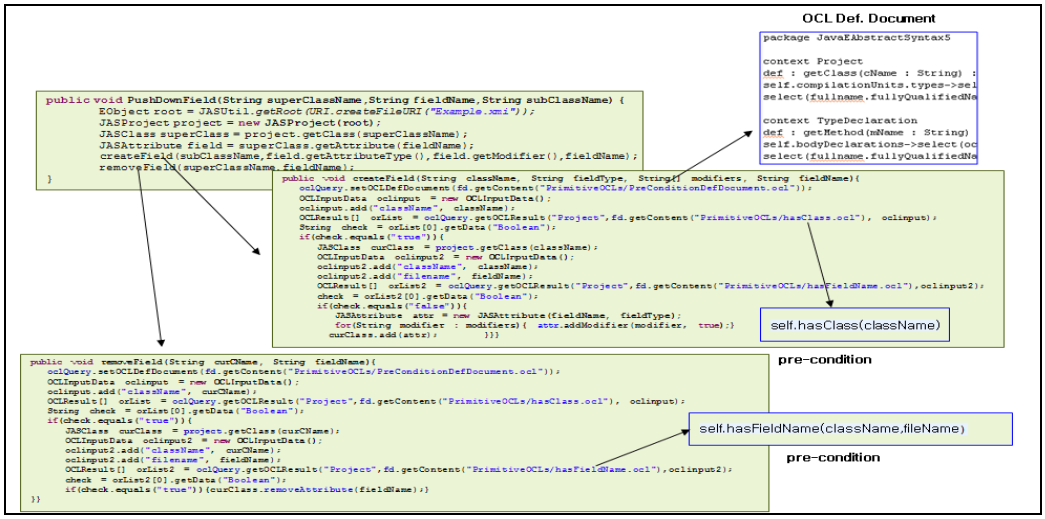


Figure 5. PushDown Field Refactoring

The CreateField and RemoveField methods are applied on XML documents which is created by JavaEAST proposed in our research. If we execute the method shown in Figure 5, it produces changed XML document based on JavaEAST. And then we transform it to another XML document based on JavaAST. Finally, we can obtain modified source code by using Code generator.

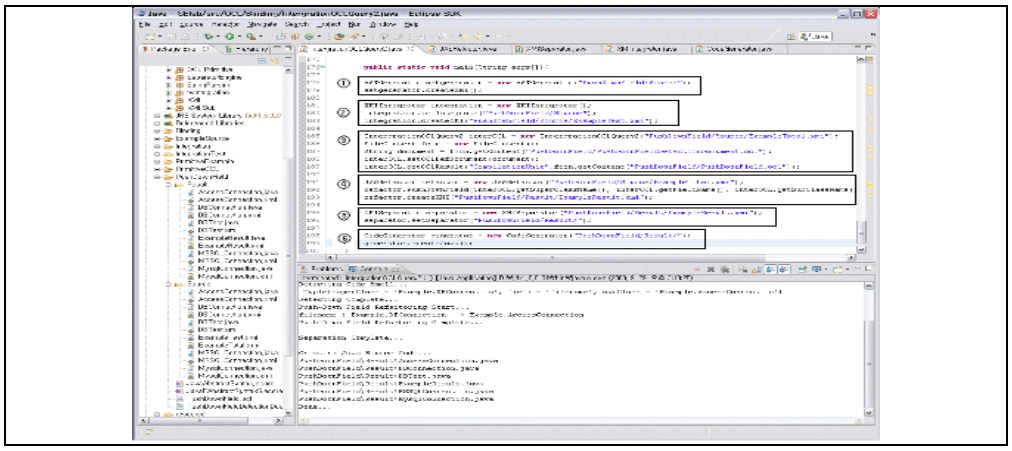
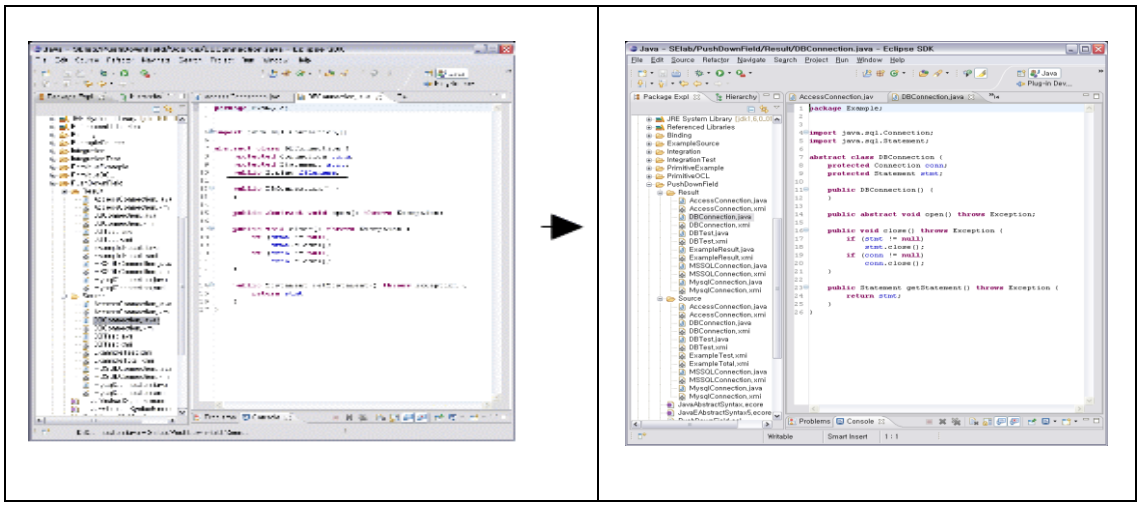
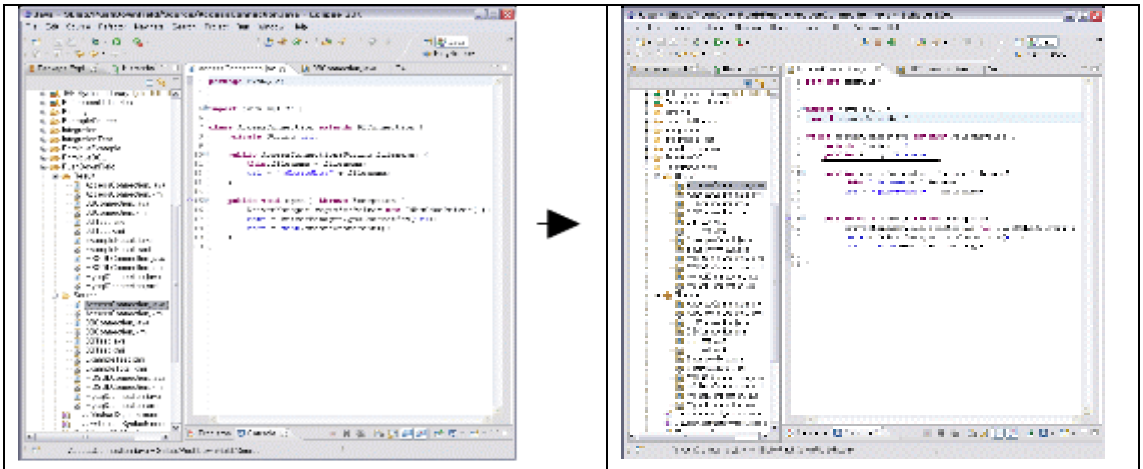


Figure 6. PushDown Field Refactoring

The Figure 6 shows detecting code smell using OCL and the source code representing the procedure to do refactoring. Steps of the procedures are following order.

1) Transform java source code into JavaAST based XML document
2) Transform JavaAST based document into JavaEAST based XML
3) Detecting code smell by using specified OCL
4) Apply refactoring on by using detected information
5) Transform JavaEAST based document into JavaAST based document
6) Transform JavaAST based document into java source code





Source code before refactoring applied	Source code after refactoring applied

Figure 7. PushDown Field Refactoring

The Figure 7 shows result of executing code in Figure 6. It also represents the 'filename' field of DBConnection class which was in Figure 3 example program class diagram, is moved into AccessConnection class.

In this chapter, we tried to detect code smells automatically and refactoring based on proposed framework in this paper. In our database connector example, pushdown field code smell is detected automatically. And we also apply refactoring process to it through six phases. Automated codesmell detection through OCL translator and automated refactoring process will produce modified source code. To detect code smell automatically, we apply specification of code smell made up in OCL to it shown in Figure 4 by using the code represented in Chapter 4. There is strong advantage to detect code smell without any program development. It is possible to detect them by just using specified OCL.

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.

We organized it not only just detecting code smell but also proposed a way to solve detected code smell. The source code will be changed by applying refactoring. As applied example, we verified detecting code smell and refactoring method proposed in our research, as result of moving detected trouble field into appropriate class by using Push down field refactoring.

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] M. Fowler, "Refactoring: Improving the Design of Existing Programs", Addison-Wesley, **(1999)**.
[2] M. Fowler, "Refactoring: Improving the Design Existing Code", Addison Wesley, **(1999)**.
[3] S. Slinger, "Code Smell Detection in Eclipse", Delft University of Technology, **(2005)**.
[4] Y. Kataoka, D. E. Michael, W. G. Griswold and D. Notkin, "Automated Support for Program Refactoring using Invariants", Proc. Int. Conf. on Software Maintenance, IEEE Computer Society Press, **(2001)**, pp. 736-743.
[5] S. Ducasse, M. Rieger and S. Demeyer, "A language independent approach for detecting duplicated code", H. Yang, L. White, editors, Proc. Int'l Conf. on Software Maintenance, IEEE Computer Society Press, **(1999)**, pp. 109-118.
[6] F. Simon, F. SteinbruUckner and C. Lewerent, "Metrics Based Refactoring", Proc. 5th European Conference on Software Maintenance and Reengineering, IEEE Computer Society Press, **(2001)**, pp. 30-38.
[7] H. E. Murphy and A. P. Black, "An Interactive Abbient Visualization for Code Smells", ACM SOFTVIS '10 Proceedings of the 5th international symposium, Software visualization, **(2010)**, pp. 5-14.
[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)**.
[12] E. Emden and L. Moonen, "Java Quality Assurance by Detecting Code Smells", Appears in Proceedings of the 9th Working Conference on Reverse Engineering, IEEE Computer Society Press, **(2002)** pp. 97-108.
[13] R. C. Holt, "Structural Manipulations of Software Architecture using Tarski Relational Algebra", Proc. 5th Working Conference on Reverse Engineering (WCRE'98), **(1998)**, pp. 210-219.