Identification of the bad smells in Java code using XPath

Michal Námešný
Faculty of Informatics and Information Technologies
Slovak University of Technology in Bratislava
Bratislava, Slovakia
michal.namesny@gmx.com

Ivan Polášek
Faculty of Informatics and Information Technologies
Slovak University of Technology in Bratislava
Gratex International, a.s., Bratislava, Slovakia
polasek@fiit.stuba.sk

***Abstract*—Refactoring is a technique in software development which we use to try to keep the code clear and easier for efficient adding of a new functionality. In the process of refactoring we try to find unsuitable structures in the source code and fix them without changing its external behavior. This requires a certain amount of time which may expand due to the size of the software that is being developed. An opportunity of acceleration of this technique is in its automation. We propose and implement code smell's editor with generator of XPath expressions. Expressions are evaluated over serialized Abstract Syntax Tree of Java code to XML. After the evaluation of XPath expression, the returned nodes represent an instances of code smell. Refactoring tool from the smell structure determine appropriate refactoring from predefined catalog.**

***Keywords*-Refactoring; Bad smells; XPath; AST**

I. Introduction

Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure[1]. Refactoring is a necessary part of program coding. Keep code clean and well structured is important, especially when we are developing a large and very complex software systems. Automation of this process is a good way to find an inappropriate code structures called smells early and eliminate routine in their correction. In this paper, we focus on the refactoring of program code written in the Java language.

II. Problem

The whole refactoring we can divide into two steps, the identification and the correction of the bad smell. Even if there are resources [1][2] where are individually described each smell, execution of the both steps are very subjective and depends on the type of project and programmer experiences. Introduction of automation can reduce influence of programmers on these processes and improve quality of code.

A lot of approches try identify bad smells by reusing algorithms for identification of design patters. Examples of such algorithms are Bit-vector Algorithm, Pattern Matching or Similarity Scoring Algorithm. These solutions are appropriate only for certain types of bad smells, mostly design flaws in the model. In the figure 1 is Observer design pattern. His structure is very different from structure of code smells. He consist of several classes and each class plays specific role. Basic differences between design patterns and bad smells are shown in table I.

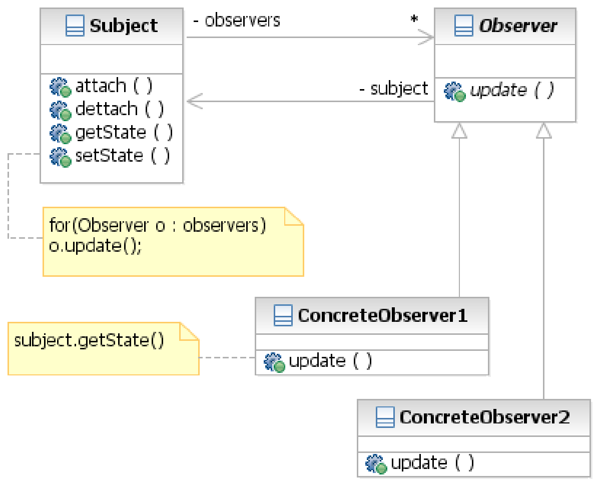


Figure 1. Observer design pattern [3]

Table I
Difference between design patterns and bad smells

Design patterns	Bad smells in code
General solution, which may not be code complete	Concreate implementation
Consist of three and more classes with known cardinality of associations	Bad smell usually occur within class or between two classes
Know predefined structure	Various structure, which depends on concreate smell. Long Method has unknown body. Lazy Class is smell due to missing functionality in class
Design patterns are well described with their structure and containing components	Subjective opinion of programmer to identify bad smell. Long method may not be always Long method smell

III. Related Work

There are many approaches to identify bad smells in code, but few of them are able apply appropriate refactoring on the found incorrect structure. Moha proposed a method for the specification and detection of code and design smells[4] and instantiate it as detection technique DETEX in tool Ptidej[1]. With this technique is possible identify bad smells and also code antipatterns such as Blob or Spaghetti Code, which indicate larger design problems.

Lanza and Marinescu presented object-oriented metrics[5] and their usage in practice. Analyse of the code with metrics is alternative approach how to identify bad smells. We aren't identifing concreate instances of smell, but we are looking for deviations from good object-oriented design. This approach was implemented in iPlasma[2] tool.

JDeodorant[3] is an Eclipse plugin that identifies predefined bad smells - Feature envy, Special case, God class and Long method without the possibility of extending. An API for searching of smells is based on the Abstract Syntax Tree. Advance of this tool is ability resolved these few smells with exact methodology. For example Long method is resolved by Extract Method refactoring[6] and God class is resolved with Extract Class[7].

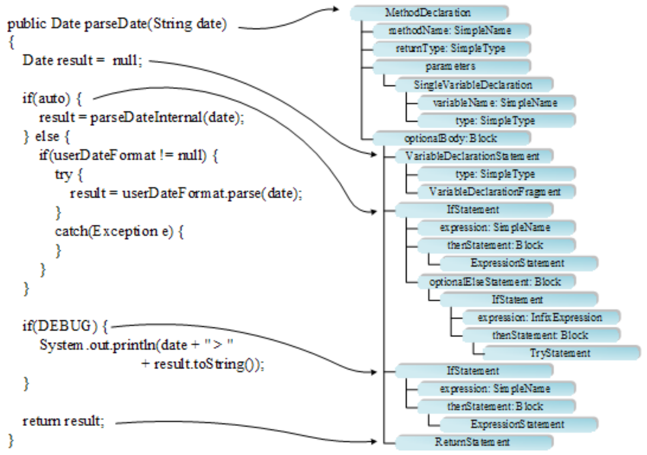
IV. Approach

Our approach is based on development enviroment Eclipse[4] and its base framework The Abstract Syntax Tree (AST). AST maps Java source code in a tree form, which is more convenient to static analyse and program modifications. Each element in the Java is represented as a specific subclass of ASTNode. In the figure 2 is a sample of mapping method to the AST. For example method is represented as MethodDeclaration node, variable declaration is represented as VariableDeclarationFragment node and strings which aren't a Java keywords are represented as SimpleName nodes. The good is as we view on the code. When we are looking for smell Long method, we don't count number of lines which depends on custom user code formatter, but we count used statements in examine method.

In the figure 3 is a typical workflow of an application using AST. After parsing source code into AST with ASTParser, we can directly modify it, or note these modifications in a separate protocol handled by an instance of ASTRewrite. At the end, we apply performed changes back to the source. This is a effective way how to programatically manipulate with Java code.

The follwing sections describe main parts of our approach.

[1]http://www.ptidej.net/downloads/ptidejdemo/
[2]http://loose.upt.ro/iplasma/index.html
[3]http://www.jdeodorant.com/
[4]http://www.eclipse.org/



```
public Date parseDate(String date)
{
  Date result =  null;

  if(auto) {
    result = parseDateInternal(date);
  } else {
    if(userDateFormat != null) {
      try {
        result = userDateFormat.parse(date);
      }
      catch(Exception e) {
      }
    }
  }

  if(DEBUG) {
    System.out.println(date + "> "
                       + result.toString());
  }

  return result;
}
```

Figure 2. Java method mapping on AST

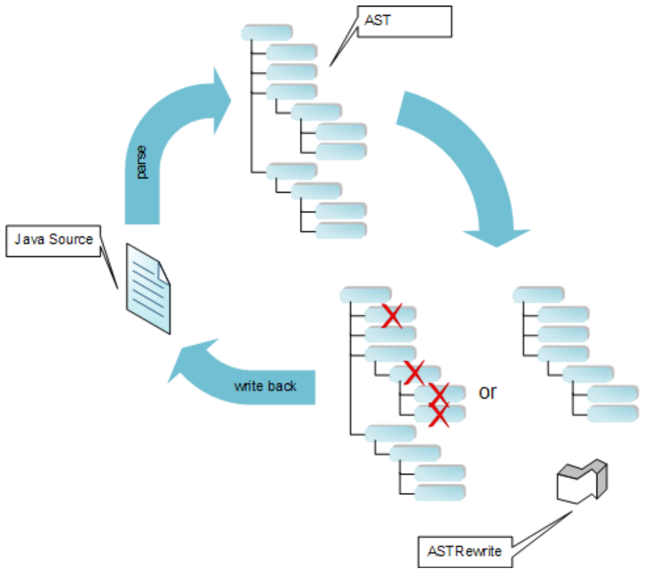


Figure 3. AST workflow [8]

A. AST and search engine

The easiest way to traverse and search patterns in AST is using Visitor design pattern. For every smell we can create own visitor and evaluate it over tree. This approach is powerful according to speed of search and his accuracy, but take a long time to program it.

To reduce time needed to define a smell catalog, we have to choose another format for storing smell definitions. If we consider that XML has tree structure too, we can descibe bad smell with XPath. XPath is used to query XML documents, exactly Document Object Model(DOM). We are able to make transformation between AST and DOM. In our solution we serialize AST to XML document and load it with DOM parser.

Search engine just evaluate smell's XPath expression over

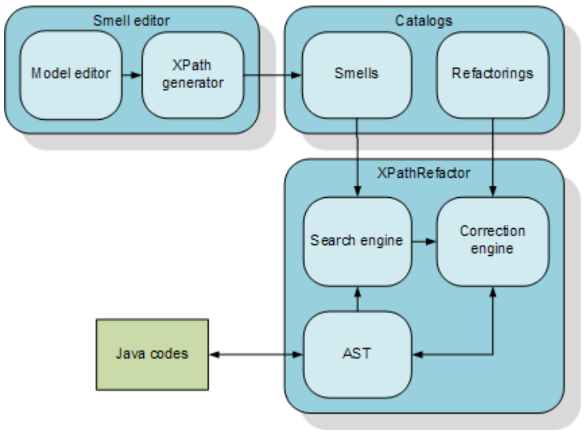


Figure 4. Refactoring tool architecture

transformed AST and return selected nodes as founded smell's instances. XPath expressions are saved in common editable catalog. However many programmers are familiar with Xpath, there was an ambition to afford opportunity edit catalog by a wider audience. In the figure 4 is a rough architecture of the refactoring framework. AST of parsed Java codes and smell catalog are inputs to the seach engine. Correction engine can repair founded bad smells, if exist appropriate refactorings in predefined catalog. To edit smell catalog we propose graphical Smell editor with XPath generator.

B. *Our contribution: Smell editor with XPath generator*

We model the smell with the basic building block of code. First we identified main common parts such as Class, Method, SwichStatement, IfStatement and Parameters needed to modeling smell's structure. Than we identified elements Too Large, Too Small and Constant Number called smell symptom's, which will indicate something wrong in the code. Meaning of the smell symptom depends on it's parent element. For example symptom Too Large in the Class element will be indicate Large Class bad smell, but in the Parameters will be indicate Long parameter list. The model of the smell is in the figure 5.

Entry to the model is class Smell, which contains method getXPath(). The method is recursively called over defined structure and compose XPath expression. Essentially XPath generator is implemented in getXPath method's bodies in all classes. Generator is presented by the following pseudocode.

```
public String getXPath() {
    XPathBuilder xpath = new XPathBuilder();
    xpath.append(NAVIGATE_EXPRESSION);
    xpath.append(PREDICATE);
    foreach(ISmellPart part)
        xpath.append(part.getXPath(this));
    if(smellSymptom != null)
        xpath.append(smellSymptom.getXPath(this));
    xpath.append(SELECT_EXPRESSION);
}
```

XPathBuilder is helper class to compose XPath expression, which inherit from StringBuilder. Expression is build in five steps, which can be omitted in dependence on element type. First is navigate to particular nodes from previous node. Nodes can be constrained by predicates such as methods of type constructor. Next we loop over child nodes of smell and generate subexpression. Actual model allow only one smell symptom in smell part. Finally append select node expression, which can be defective method name, or whole node. Mapping between designed smell (Long parameter list) and generated XPath query is ilustrated in the figure 6.

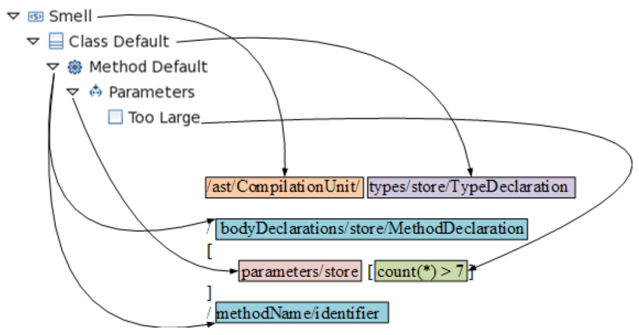


Figure 6. Long parameter list smell

On the next Long method smell in the figure 7 and corresponding XPath expression we can see difference in using Too Large element with another parent item.

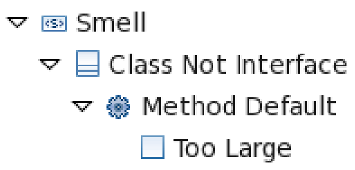


Figure 7. Long method smell

```
/ast/CompilationUnit/
types/store/TypeDeclaration[isInterface='false']
/bodyDeclarations/store/MethodDeclaration
[
    count(optionalBody/statements/store/*)>30
]
/methodName/identifier
```

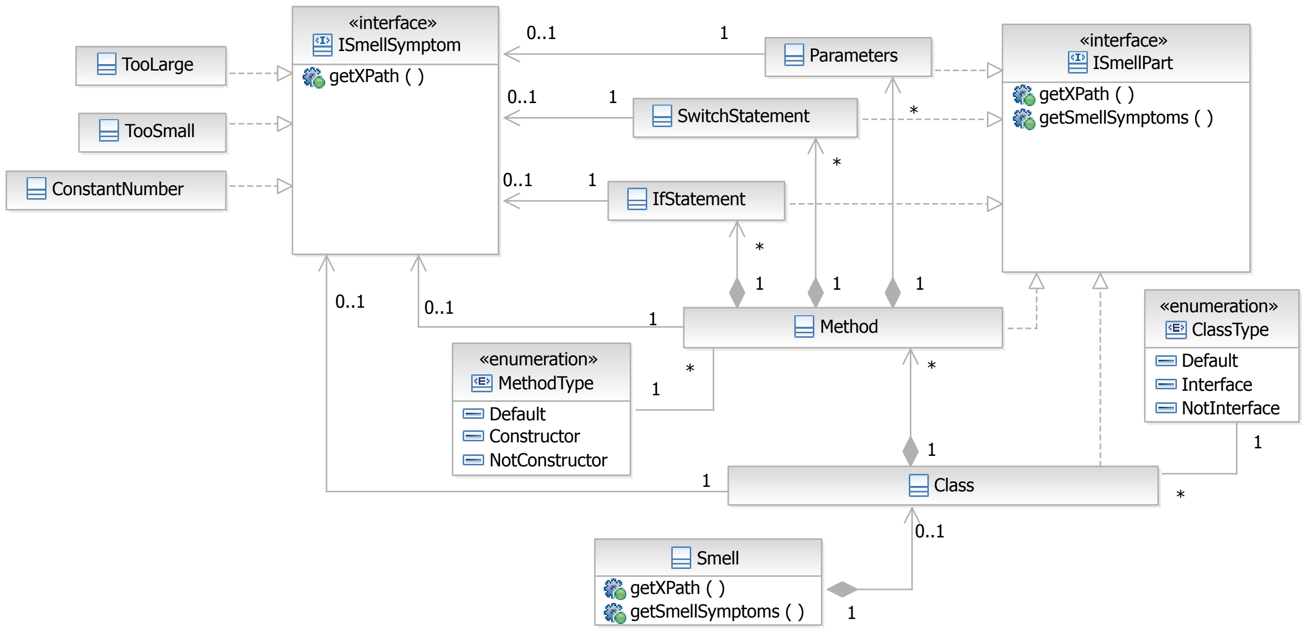


Figure 5. Smell model

The whole smell editor was build from designed model with technology Eclipse modeling framework[5] as Eclipse plug-in. With this technology is possible to easily extend the smell model with new elements. After adding new element, the smell editor is regenerated and we just extend XPath generator with valid behaviour. In the figure 8 is screenshot of our smell editor.

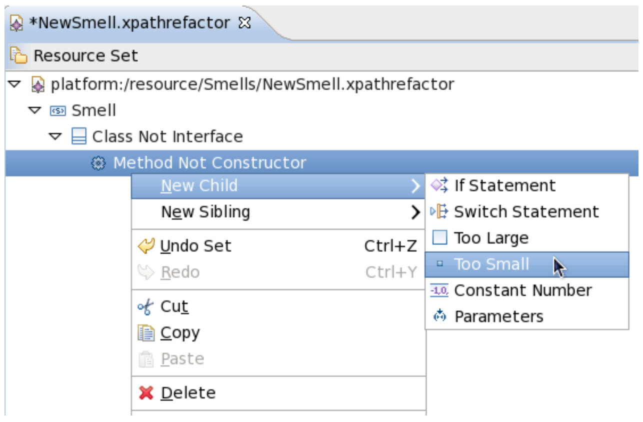


Figure 8. Smell editor

The list of bad smells, which are able design in our framework:

- Long Method
- Large Class
- Long Parameter List
- Switch Statements
- Lazy Class
- Magic number
- Special case

These are only basic bad smells. In our framework is possible combine more bad smells into the one, such as long method with many parameters. Additional identified components with which we can extend the Smell model are CatchStatement, or Name element to enable model Type embedded in name or Uncommunicative name bad smells.

C. Correction

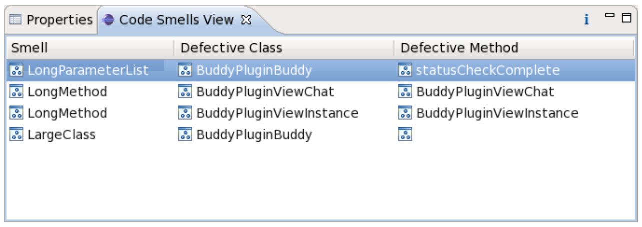
The correction engine is based on the simple rule system. We have predefined set of refactorings with preconditions in refactoring catalog. Precondition consist from smell sypmtoms and their parent's elements. The right refactoring is choose by knowledge, that we know which refactoring is the best for a symptoms defined in the smell. Every refactoring in catalog match to the one program method in system.

There are many chalenges in automation correction domain. We need more than 40 refactoring to correct all 22 Fowlers bad smells. Many smells is possible repair in two and more ways. Any way is incorrect, but one of them is

[5]http://www.eclipse.org/modeling/emf/

better than others in concreate situation. So simle refactoring as Extract Method (for programmer) takes a long time to program it. We have to identify which fragments of code extract from defective method [6] [9] to create new valuable methods.

To provide semi-automation correction of some smells we reuse refactoring wizards from Eclipse Language Toolkit. After selecting found smell in fig. 9, user can execute refactoring action.



Smell	Defective Class	Defective Method
LongParameterList	BuddyPluginBuddy	statusCheckComplete
LongMethod	BuddyPluginViewChat	BuddyPluginViewChat
LongMethod	BuddyPluginViewInstance	BuddyPluginViewInstance
LargeClass	BuddyPluginBuddy	

Figure 9. Smell View

V. Related work

Our solution is inspired by tool PMD[6]. PMD check coding styles rather than search bad smells. It looks for potential problems like overcomplicated expressions, or empty catch/finally statements. PMD enable extends detection rule catalog, but user has to be familiar with Java or XPath.

The most work in automated correction made Jdeodorant team. In their tool is implemented following five fully automated refactorings.

- Move Method
- Replace Conditional with Polymorphism
- Replace Type code with State/Strategy
- Extract Method
- Extract Class

There some preddefined refactoring in Eclipse framework too, but they need user interaction with enviroment. For example Jdeodorant automatically determines method blocks for applying Extract Method refactoring. In Eclipse user have to manually select chosen lines of code, before running refactoring action. Some preddefined useable refactorings in Eclipse are Extract Constant, Extract Superclass, Extract Class, Push down, Push up, Introduce Parameter Object or Encapsulate Field.

Isn't so simple automatize these Eclipse refactorings. Even if we identify the right refactoring for current bad smell, we don't know to determine the most convenient scope of his application. Long parameter list smell we can refactor with Introduce Parameter Object. It is not correct to replace all parameters with one object, but choose only those which are in context, respectively are found together in other methods as data clumps. Before applying refactorings on defective code is important to check refactoring preconditions. We can't refactor method declaration, if it is implementation of public interface. Every refactoring needs precise methodology, how to be performed.

[6]http://pmd.sourceforge.net/

Table II
Bad smells

Bad Smell	Automated identification	Automated correction	XPath identification
Duplicated Code	x	x	
Long Method	x	x	x
Large Class	x	x	x
Long Parameter List	x	x	x
Divergent Change			
Shotgun Surgery			
Feature Envy	x	x	
Data Clumps	x		
Primitive Obsession	x		
Switch Statements	x	x	x
Parallel Inheritance Hierarchies	x		
Lazy Class	x		x
Speculative Generality	x		
Temporary Field	x	x	
Message Chains	x		
Middle Man	x		
Inappropriate Intimacy	x	x	x
Alternative Classes with Different Interfaces			
Incomplete Library Class			
Data Class	x		x
Refused Bequest	x		
Comments	x		
Type embedded in name	x		partly
Uncommunicative name	partly		partly
Inconsistent Names			
Dead code	x		
Magic number	x	x	x
Null check	x		
Special case	x	x	x
Complicated boolean expression	x	x	x

VI. Conclusion and future work

Refactoring is the important part of developing software systems. Automation detection of bad smells can improve quality of the code and automation correction reduce routine work during resolving a founded smells. We proposed and implement the XPath based method for detection bad smells in the Java code. In the table II is overview of the bad smells and opportunities in their automated identification and correction. In the third column are marked bad smells which are possible identify with our approach.

Currently the thresholds(number of parameters in Long parameter list, or number of statements in Long method smell) in the generated XPath expression are defined in the code, but there is a option integrate it into Eclipse preferences. The user will be able to define custom thresholds, which substitute appropriate mark in XPath before running search action. The next option is integrate with some code analysis tool, which set thresholds on optimal value depending on the project.

Search engine performance can be improved by evaluating XPath expression over AST, or direct transformation from AST to DOM object. Transformation can be performed using Visitor design pattern. Using polymorphic methods visit() and endVisit() we can effectively constrain tree nodes to transformation.

```
class Visitor extends ASTVisitor {
    Document xmlDocument;
    boolean visit(MethodDeclaration node) {
        lastClass.appendChild(transform(node));
        return true;
    }
    boolean visit(Javadoc node) {
        return false;
    }
    ...
}
```

Extending the smell model with relations between classes, or another elements such as CatchStatement enable describe a larger amount of smells. But our future work shoud be focus on creating large catalog of code refactorings, or propose more effective way to refactor program code.

References

[1] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[2] W. Wake, *Refactoring Workbook*. Addison Wesley, 2003.

[3] E. Gamma, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[4] N. Moha, *DECOR: A Method for the Specification and Detection of Code and Design Smells*. IEEE Transactions on Software Engineering, vol. 36, no. 1, pp. 20-36, 2010.

[5] M. Lanza, R. Marinescu, *Object-Oriented Metrics in Practice*. Springer-Verlag, New York, 2005.

[6] N. Tsantalis, A. Chatzigeorgiou, *Identification of Extract Method Refactoring Opportunities*. pp. 119-128, 13th European Conference on Software Maintenance and Reengineering (CSMR'2009), Kaiserslautern, Germany, March 24-27, 2009.

[7] N. Tsantalis, A. Chatzigeorgiou, *Identification of Move Method Refactoring Opportunities*. IEEE Transactions on Software Engineering, vol. 35, no. 3, pp. 347-367, May/June 2009.

[8] T. Kuhn, O. Thomann, *Abstract Syntax Tree*. 2006.

[9] L. Yang, H. Liu, Z. Niu, *Identifying Fragments to Be Extracted from Long Methods*. apsec, pp. 43-49, 2009 16th Asia-Pacific Software Engineering Conference, 2009.