Automatic identification of the anti-patterns using the rule-based approach

Ivan Polášek*, Samuel Snopko** and Ivan Kapustík***
Slovak University of Technology
Faculty of Informatics and Information Technologies
Bratislava, Slovak Republic
*polasek@fiit.stuba.sk, **ssnopko@gratex.com, ***kapustik@fiit.stuba.sk

***Abstract* — Adjustment and rebuilding of the source code is an integral part of the software engineering life cycle, mainly in modern Agile Modeling and eXtreme Programming. Fowler identified 22 bad smells that could be found in the refactoring process and this set is still growing for model flaws in the UML diagrams as well. This paper proposes a set of bad smells that may be found in software models and a way for creating facts and rules for finding them in the models.**

I. INTRODUCTION

A lot of technical papers describe the problem of refactoring and searching for bad smells, but the main part of the research deals with source code and ignores the design phase for the enterprise projects.

Kramer and Prechelt [1] define the way to represent good reusable *design patterns* using the rule-based metainterpreter in *Prolog*. They use this paradigm for detection patterns inside the software. Their work defines the necessary predicates for creating detection rules in Prolog. A good example of the rule for the design pattern *Composite* is in Fig. 1.

```
composite(Cpnt,Leaf,Compos):-
class(_,Cpnt),
class(concrete,Leaf),
class(concrete,Compos),
operation(_,_,Cpnt,Op,_,_,_),
operation(_,_,Leaf,Op,_,_,_),
operation(_,_,Compos,Op,_,_,_),
operation(_,_,Cpnt,Add,_,_,_),
operation(_,_,Cpnt,Remove,_,_,_),
operation(_,_,Cpnt,GetCh,_,_,_),
operation(_,_,Compos,Add,_,_,_),
operation(_,_,Compos,Remove,_,_,_),
operation(_,_,Compos,GetCh,_,_,_),
inheritance(Cpnt,Leaf),
inheritance(Cpnt,Compos),
aggregation(Compos,exactlyone,Cpnt,many)
```

Figure 1. The rule of the Composite in Prolog [1]

Štolc and Polášek [2] analyzed the possibilities of searching anti-patterns and created a bad smell editor as a plug-in application into the framework *Eclipse*. They use OCL[1] query engine for searching instead of the rule-based system, but they focus on the bad smells as a design flaw and the way to use the graphical editor for defining them. This editor has its own modeling language, which is conform to the UML metamodel.

The editor consists of two sides (see Fig. 2):

- The left-hand side contains bad smell pattern searching in the real model.
- The right-hand side shows the correct solution. Relationships between these two sides define which elements are identical.

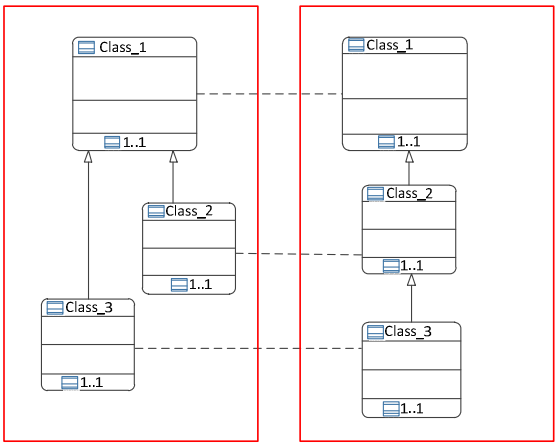

Figure 2. Representation of the two sides editor

II. OUR APPROACH

The goal was to find bad smells in system models in the project design phase, where UML diagrams represented models. Explicit knowledge about bad smells and explicit description of the models hinted to use knowledge-based systems for bad smell detection. The most straightforward way was to employ a simple rule-based system, containing only facts, rules and inference engine.

Jess was used as an actively supported rule-based language with long-time proved inference engine to create rule-based system. *Jess* is written in *Java*, so it extends its capabilities with *Java* functions.

The next step was to transform the model from UML diagrams to facts readable by *Jess*. Most UML editors can store diagrams in XMI[2] format. This file format can be efficiently handled by the *Acceleo* module from the *Eclipse* environment. Utilizing its capabilities created a

[1] Object Constraint Language

[2] XML Metadata Interchange

978-1-4673-4750-1/12/$31.00 ©2012 IEEE

fact generator, which automatically filters and transforms XMI constructs into *Jess* facts.

Rule-based system needed to recognize appropriate facts. Therefore templates were created for these facts.

TABLE I.
ORIGINAL BAD SMELLS [4]

Bad smell	Can be found in class diagram?
Duplicated Code	YES
Long Method	YES
Large Class	YES
Long Parameter List	YES
Divergent Changes	NO
Shotgun Surgery	NO
Feature Envy	NO
Data Clumps	YES
Primitive Obsession	NO
Switch Statements	NO
Parallel Inheritance Hierarchies	NO
Lazy Class	YES
Speculative Generality	NO
Temporary Field	YES
Message Chains	NO
Middle Man	YES
Inappropriate Intimacy	NO
Alternative Classes with Different Interfaces	NO
Incomplete Library Class	NO
Data Class	YES
Refused Bequest	NO
Comments	NO

Templates represent the elements of the class model needed for bad smell detection (see Fig. 3).

For example, the template of the attribute from the class diagram consists only of four slots. The slot named *nameOfClass* represents the class, where the attribute lies within, so the name of this class is saved in this slot.

The slot *name* represents the name of the attribute. The slot *type* describes the type of the attribute and the slot *visibility* represents the visibility of the attribute, with the default value set to *Public*.

Finally, it is necessary to write detection rules. According to Astels [3], the original bad smells proposed by Fowler and Beck [4] can be found in the Class Diagram and Sequence Diagram. Since most of the bad smells are occurring in the UML Class Diagram (see Table I), the research focused on work with this group of smells.

The regular UML allows class relationship with multiple ends. The presented work uses relationships with two ends, but this is not a limitation, since it is possible to transform one multi-ended relationship to multiple double-ended.

Rules are defined almost straightforwardly by Fowler

```
(deftemplate Class
    (slot name)
    (slot stereotype)
    )

(deftemplate Association
    "Associtation between the Classes"
    (slot name)
    (slot class_1)
    (slot multiplicity_1)
    (slot class_2)
    (slot multiplicity_2)
    )

(deftemplate Aggregation...

(deftemplate Composition...

(deftemplate Generalization
    "Generalization between the Classes."
    (slot superClass)
    (slot subClass)
    )

(deftemplate Dependency...

(deftemplate Attribute
    "Attribute of Class"
    (slot nameOfClass)
    (slot name)
    (slot type)
    (slot visibility (default Public))
    )

(deftemplate Function
```

Figure 3. Defined templates of the class diagram elements

and Beck – condition describes a pattern for bad smells and consequence contains print actions for smell type and recommendations for refactoring.

Once these regular bad smells were implemented, wider possibilities of knowledge-based systems were discovered. It is possible to define results also for incomplete or partial patterns. Consequently, the rules for detection of partial and repeatable bad smells were declared for the design model.

The next chapter describes the rules, their features and behaviour in detail.

III. THE PROCESS OF IDENTIFICATION

A typical example of a partial bad smell could be shown with the *duplicated code*. In [4] the duplicated code is defined as a code structure that can be found in more than one place in the source code of program. Duplicated code can be easily found in the class diagram. The system is looking for attributes with the same names and types (see Fig. 4) as well as for methods with the same names; parameter lists and return values that can be found in more *subclasses* of one *superclass*. If the system identifies this occurrence, it proposes to move the attributes or methods to the superclass. Also, there could be a situation in which the classes with the duplicated code are unrelated. In this

case, it is usually possible to extract the duplicated code to a new class, which will be the superclass of these classes.

The main problem of the partial bad smells is the particular presence of their properties. For example, if the duplicated code is only in some of the subclasses of their superclass. In Fig. 5, there are *Widget2 … WidgetN* – with duplicated method *Change()*. This situation is considered as a partial duplicated code, because not all subclasses have the duplicated attribute.

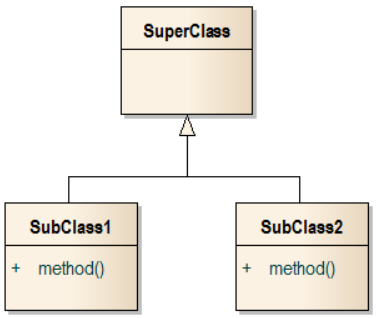

Figure 4. Regular duplicated code

In this case, it is impossible to automatically decide if this is a partial duplicated code and it is necessary to define metrics for all partial bad smells, e.g. *Duplicated Code*, *Long Method*, *Large Class*, *Long Parameter List*, *Lazy Class*, etc.

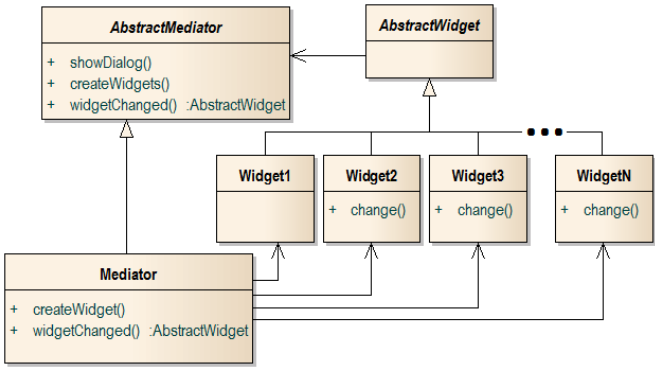

Figure 5. Partial duplicated code

In the case of partial duplicated smells there are several alternatives how to repair the model. The situation in Fig. 5 can be repaired by at least two ways (see Fig. 6 and Fig. 7), but the decision how to repair the incorrect model belongs to the model designer. The system does not repair these cases automatically, because the decision requires business view of the product.

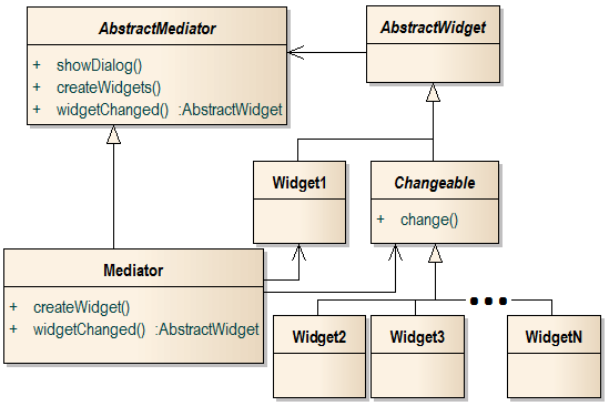

Figure 6. First solution of the partial duplicated code

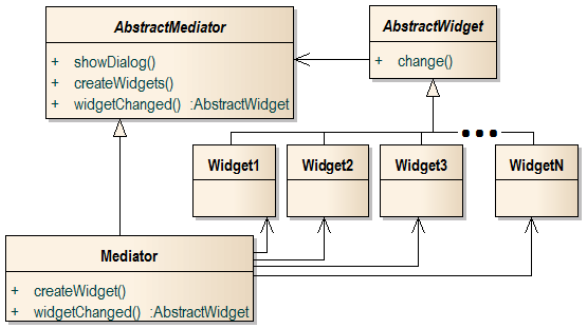

Figure 7. Second solution of the partial duplicated code

Fig. 6 presents one solution of the situation from Fig. 5. The duplicated method is extracted from the subclasses and is moved to the new class. The new class is named *Changeable* and it is the child of the *AbstractWidget*, but also the superclass for the classes *Widget2 … WidgetN*. The situation in Fig. 5 can be also solved as shown in Fig. 7 if the business conditions allow extracting the method to the superclass named *AbstractWidget*. The problem is, that in this case, the class *Widget1* also inherits the duplicated method *Change()*. Only the model designer can make this decision.

A. *Large class*

Another example of the metrics is the problem of the *Large class*. This class has too many methods, functionality and behaviour and this situation is represented in the model by a high number of attributes and methods. For this reason two metrics are created – local and overall. Local metrics accept only the number of attributes and methods from the direct neighbour classes in the special projects and the overall metric counts the number of attributes and methods of all the classes in the diagram and the external knowledge and estimations.

$$\left(\left|\{ATT_i\}\right|+\left|\{MET_i\}\right|\right) > 1.5*\left(\frac{\left(\left|\{ATT\}\right|+\left|\{MET\}\right|\right)}{\left|\{CLASS\}\right|}\right) \qquad (1)$$

Condition (1) indicates a possibility of a *Large class*. The system reports the *Large class*, if the sum of attributes and methods of the class *i* is greater than the normed average of all classes in the diagram.

B. *Rules for bad smells*

The final rules for bad smells are written in the Jess code. Fig. 8 shows the code of the partial duplicated bad smell, but only for the duplicated attributes. Similar rule is required for the duplicated method. This is as an advantage, because it makes the code clear and simple to understand.

This rule works with the base of facts. The principle of this search is easy to understand: if all the conditions from the left side are matched, then the actions from the right

side are executed. The left and the right side are separated by the symbol "=>".

The left side conditions are executed top to bottom.

```
(defrule duplicated-attribute-metric
    (Partial)
    ?superClass <- (Class)
    ?countAllSubClass <- (accumulate    (bind ?c 0)
                                (bind ?c (++ ?c))
                                ?c
                                (Generalization {superClass == superClass.name})
                                )
    ?attribute <- (Attribute)
    (Generalization {superClass == superClass.name && subClass == attribute.nameOfClass})
    ?countSmellSubClass <- (accumulate  (bind ?c 1)
                                (bind ?c (++ ?c))
                                ?c
                                (AttributeDump {superClass == superClass.name && class != attribute.nameOfClass && attribute == attribute.name})
                            )
    =>
    (if (> ?countSmellSubClass (- ?countAllSubClass ?countSmellSubClass)) then
        (printout t "PARTIAL DUPLICATED CODE : Attribute named \""?attribute.name"\" from the class \""?attribute.nameOfClass"\". This attribute
        ;(continue-looking)
        )
    )
```

Figure 8. Rule of the partial duplicated code - attribute

First, it looks if partial bad smells are allowed. Then the code "?superClass <– (Class)" requests a search for the superclass and if this condition is true for all the classes from the facts base. Then, it tries to fulfill the rest of the condition for each instance of possible superclasses. So, in the next step of the example, it counts all the subclasses of the chosen superclass. Then, it looks for the attribute, which belongs to the subclass of the chosen superclass. The system tries all the attributes. Finally, it counts the number of subclasses of the chosen superclass, which contain the chosen attribute. If the number of the subclasses with the duplicated attribute is bigger than the half of all the chosen superclass subclasses, the application reports a duplicated code.

The system can advise how to repair the identified bad smells, but it cannot refactor the code automatically. Automatic refactoring requires a complete view of the development process and product. For systems based exclusively on models, semi-automatic refactoring is recommended only. The reason is clear – the same attribute name can be, for instance, the result of a well-known copy/paste error and methods with the same name and parameters can be exactly what make sibling classes different. According to the ratio of the numbers in metrics, the application can learn what should be the right recommendation order of how to refactor bad smells. For example, if duplicated attribute occurs in more than 90 percent of all subclasses, it should be moved to the superclass. But if such attribute occurs from 50 to 60 percent of all subclasses, creation of the *SuperSubClass* (see the *Widget1* in Fig. 6) and move of the shared attribute to this new class is more suitable.

IV. Future Work

The presented work defines and creates a set of rules for bad smell searching, but if the users want to find other bad smells, they have to write new rules. The main requirements were described for the bad smell editor, templates for the rules and notation for the rule conditions in Backus-Naur form were created (see Fig. 9).

The next step subsequent to finding bad smells could be the semi-automatic refactoring of the diagrams. Jess or Java programing language can be used and the most precise way is to refactor bad smells in the base of facts and then recreate the XMI file according to these new facts. Second alternative is to use the interconnection between the Jess and Java, and try to edit diagrams directly by the right side of the rule using Java functions and libraries.

```
<constraint>  ::=  <condition> | <condition> <relation>
                   <constraint>

 <condition>  ::=  <slot-name-from-fact> <comparison>
                   <variable>"."<slot-name-from-variable> |
                   <slot-name-from-fact> <comparison>
                   <constatnt-alphanumerical>

<comparison>  ::=  "==" | "!=" | "" | ">" | "<" | >="" | "<="

  <relation>  ::=  "&&" | "||"

  <variable>  ::=  <text-without-spaces-started-with-lower-case>
```

Figure 9. BNF of condition notation

Acknowledgment

This work was partially supported by the Scientific Grant Agency of Slovak Republic, grant No. VG1/1221/12. Publication is also a partial result of the Research & Development Operational Programme for the project Research of methods for acquisition, analysis and personalized conveying of information and knowledge, ITMS 26240220039, co-funded by the ERDF.

References

[1] Ch. Kramer and L. Prechelt, "Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software", *Proceedings of the 3rd Working Conference on Reverse Engineering*, 1996

[2] M. Štolc and I. Polášek, "A Visual Based Framework for the Model Refactoring Techniques", *8th International Symposium on Applied Machine Intelligence and Informatics*, IEEE SAMI, 2010

[3] D. Astels, *Refactoring with UML*. Proc. 3rd Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering, pp. 67-70, 2002

[4] M. Fowler and K. Beck, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 2011