Anti-pattern Detection as a Knowledge Utilization

IVAN POLÁŠEK
FIIT STU, Bratislava, Slovakia

Abstract. The occurrence of anti-patterns in software complicate development process and reduce the software quality. The contribution proposes selected methods as an OCL Query, extension to Similarity Scoring Algorithm, Bit-vector Algorithm and rule based approach originally used for design patterns detection. This paper summarizes approaches, important differences between design patterns and anti-patterns structures, modifications and extensions of algorithms and their application to detect selected anti-pattterns.

1 Introduction – Basic Notions and Previous Works

Our intention in this contribution is to verify efficiency of the methods mentioned in this article for design smells detection, despite differences between design patterns and anti-patterns. We have to modify and extend the mentioned algorithms for this slightly different purpose.

Design patterns were introduced as a reusable solutions to a commonly occurring problems in software design [1]. A term *anti-pattern* was defined one year after the introduction of design patterns which represents undesirable design structure [2]. As a reference to the code smell [3], a *design smell* is a synonym to the anti-pattern contained in the software model and we focus on the smells appearing in UML (Unified Modeling Language) Class Diagram.

Software modeling is a part of almost every industrial and massive software development method. Software analysts and designers are fallible and unsure in new, unclear and vague domain which leads to design smells very often. It could complicate software model and software based on this model is harder to extend, maintain or implement correctly at all.

Refactoring is used to fix these undesirable structures and to restructure software by applying a series of refactoring rules without changing its observable behavior [3]. We can split refactoring of software models into two interlinked steps: *detection* of design smell in a model and *restructuring* bad structure to the better one. Our concern is the automatic identification of bad structures.

2 Related Works

There are several methods for identification of structures in software models. OCL (Object constraint language) is a declarative language for describing rules that apply to UML models [4]. OCL provides constraints and query expressions for identifying structures in models. It is used in refactoring methods and programs, for instance OCL is a component of the QVT (Queries / Views / Transformation), the OMG (Object Management Group) standard for transforming models. ATL (ATLAS Transformation Language) is a model transformation language which is part of Eclipse Modeling Project.

Some tools and frameworks provide software quality assessment, software diagnose and anti-patterns or smells detection. Borland Together offers a set of Eclipse plug-ins which supports model validation and transformation by using OCL and QVT. VIATRA2 is a framework for automated precise model transformation with utilization of graph transformations and ASM (Abstract State Machines).

Ramaswamy et al. in [7] introduced method for pattern matching in the string by using approximate fingerprinting. Main contribution is "approximate" searching by sliding fingerprinting window by more than one byte in order to faster processing and reducing memory accesses, while keeping low number of false positives. Approximate matching method was designed for analysis over network data streams where speed and number of memory accesses is important. This method is not particularly useful for design smells identification, because increasing number of false positives is disadvantage and memory access is not a concern.

Gueheneuc et al. proposed an experimental study [8] of classes playing roles in design patterns. For identification they compare external attributes of classes like size (number of methods and fields), filiations (number of parents and children), cohesion (degree to which methods and attributes of a class belong together) and coupling (strength of the association between classes). Machine learning algorithm is used to find commonalities among classes playing a role in design pattern.

Jakubík in [11] extended similarity scoring algorithm from [6] to optimize design pattern identification. They apply weighting to the structural parts of the design pattern, thus more important features (i.e., attributes) of pattern have higher weight in process of calculating similarity. This extension reduces number of the false positives even when used with high threshold. When similarity scores are calculated, results are filtered by instance filtering extension which checks if identified patterns have all important features.

Kramer and Prechelt [16] 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)
```

Fig. 1. The rule of the Composite in Prolog [16]

Dong et al. [9] presented an approach with three phases: structural, behavioral, and semantic analyses. In structural analysis phase they propose matrix with weights of structural information of the system and design pattern. Each important structural element is associated with prime number (e.g., 2 = Attribute, 3 = Method, 5 = Association etc.) and weight for each class is calculated as multiplication of these prime numbers. Weights are calculated in $n \times n$ matrix (each cell initially has value 1) where n is number of classes. Afterwards cells are multiplied by appropriate prime number for each structural element they fulfill. Behavioral analysis helps eliminate false positives by identifying behavioral characteristics of pattern candidates in source code. Lastly semantic analysis verifies naming conventions of classes.

Majtás [15] proposed to add metric (count of instances, operation invokes, call invokes, attribute changes) for behavioural analysis to increase the precision of the detection.

2.1 Bit-vector Algorithm for Design Patterns

Kaczor et al. in [5] described a technique to identify design patterns in models by using bit-vector algorithm on string representation. Firstly, model and design pattern are transformed into digraphs (directed graphs). Software model is actually graph where entities are represented by vertices and relationships between entities are graph's edges. For sake of simplicity authors consider only binary relationships as edges: creation, specialization, implementation, use, association, aggregation and composition. Binary relationships are directed, thus created graphs will be digraphs.

For transforming these graphs into string representation by Eulerian path, graphs need to have Eulerian circuit (trail in a graph which visits each edge exactly once and ends in a starting vertex). A digraph is typically not an Eulerian graph (i.e., graph with an Eulerian circuit). Transformation of a digraph into an Eulerian graph consists of adding *dummy edges* between vertices with unequal in-degree and out-degree. Authors use transportation simplex to obtain number of needed dummy edges. Then transportation simplex computes minimum cost (minimum number of dummy edges).

Afterwards string representation is created with Eulerian path by traversing each edge of graph exactly once. String representation consists of connected triples *Vertex Edge Vertex* (or *Class Relationship Class* in model terms). These triplets from design pattern string representation are sequentially parsed and searched in model string representation. When all triplets are parsed, candidate classes for design pattern are identified.

2.2 Similarity Scoring Algorithm for Design Patterns

Tsantalis et al. in [6] introduced an approach to design pattern identification based on algorithm for calculating similarity between vertices in two graphs.

System model and patterns are represented as the matrices reflecting model attributes like generalizations, associations, abstract classes, abstract method invocations, object creations etc. Similarity algorithm is not matrix type dependant, thus other matrices could be added as needed. Mentioned advantages of matrix representation are 1) easy manipulation with the data and 2) higher readability by computer researchers.

Every matrix type is created for model and pattern and similarity of this pair of matrices is calculated. This process repeats for every matrix type and all similarity scores are summed and normalized. For calculating similarity between matrices authors used equation proposed in [8].

Authors minimized the number of the matrix types because some attributes are quite common in system models, which leads to increased number of false positives.

2.3 Other works

Moha et al. [12] proposed a method called DECOR that defines steps for the specification and detection of code and design smells. Authors also contribute with the detection technique DETEX, which instantiates this method.

The detection technique consists of

1. *domain analysis* (an unified vocabulary of reusable concepts is created and a classification of smells is defined using the key concepts),
2. *specification* (performed using a domain-specific language (DSL) in the form of rule cards using the previous vocabulary and taxonomy and rules describing the properties that a class must have to be considered as a smell),
3. *algorithm generation* (detection algorithms are automatically generated from models of the rule cards)
4. and *detection* (algorithms are applied automatically on models of the systems).

DETEX was tested by detecting anti-patterns Blob, Functional Decomposition, Spaghetti Code and Swiss Army Knife in several projects with a precision from 41.1% to 88.6%.

Java based software PMD contains static ruleset source code analyzer and identifies possible bugs, dead code, duplicate code and other potential code problems. PMD includes a set of built-in rules and supports the ability to write custom rules (using XPath or Java classes).

Integrated environment for quality analysis of object-oriented software systems iPlasma [13] includes support for all the necessary phases of analysis.

Eclipse plug-in JDeodorant [14] identifies Feature Envy bad smell in Java projects and resolves them by applying the appropriate Move Method refactorings.

Works focused on *pattern* identification in software model structures inspired us to use their algorithms for identifying design smells and test their advances and potence.

There are several possibilities for searching pattern in software model. We are focusing on *bit-vector* and *similarity scoring algorithms*, because they operate with strings and matrices respectively. These representations of model are easily created from XMI (XML Metadata Interchange, most common interchangeable format for software models), which allows transformation into the other structures and supports our technique to integrate with other tools following this OMG standard. Another reason is application of these algorithms for design pattern identification in some works.

3 Differences Between Design Patterns and Design Smells

The main difference between design patterns and design smells is in the structure complexity. Design patterns usually consist of several classes and each class plays specific role. Complexity and roles of classes are often used in automated design pattern identification because their important features can be easily defined and recognized, which is not possible in case of some anti-patterns.

Presence of design pattern in a model is often clear, but a design smell can be recognized as a smell by one designer and as a good design by the other at the same time. Moreover, some anti-patterns are the exact opposite of the others. Therefore it is designer's decision if an instance is actually an anti-pattern.

Some design smells are loosely defined (e.g., Large Class has *many* methods and attributes, Lazy Class does *too little*), they have no exact structure which may cause complications for automated detection.

All design patterns are defined by several structure features, but anti-patterns have very large variety of characteristics (e.g., number of methods, naming of methods/classes, method parameters, class functionality etc.), therefore it is harder to apply general detection rules to all of them.

4 Our Approaches: Adapted and Extended Methods for Anti-pattern Detection

Our main concern is the adaptation of selected methods by extending their searching capabilities for smell detection. Most anti-patterns have additional structural features, thus more model attributes need to be compared. We have chosen several smells attributes different from design patterns features which cannot be detected by original methods.

Smell characteristics (e.g., what is *many* methods and attributes) need to be defined. On the other hand, some design patterns characteristics are also usable for flaw detection. Structural features included in both extended methods are:

- associations (with cardinality)
- generalizations
- class abstraction (whether a class is concrete, abstract or interface).

4.1 Bit-vector Algorithm Extensions

1) Names of methods/attributes

First extension appends the method/attribute name comparision, because some smells may need to identify occurrence of a concrete method or attribute in several classes. An edge is added in digraphs for each method. Starting and ending point of this edge is the same class where the method is occurring. Name of this edge could be:

- *mX* – for general method
- *mNUMBER* (e.g., *m1*) – for specific method which is considered as an important and umbered to differ from other watched methods.

In Fig. 2. (a) is a diagram for design smell *Duplicated Code* where two subclasses have concrete method which is not in their superclass (i.e., method is not overridden). Fig. 2. (b) shows Eulerian Graph for this smell.

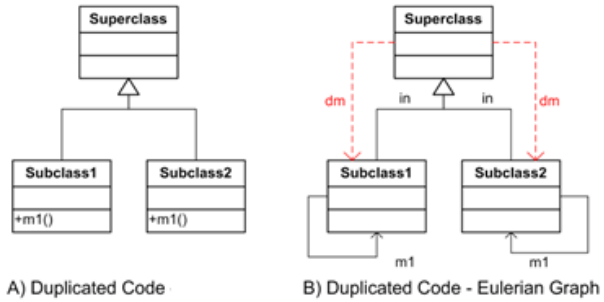

Fig. 2. Design smell Duplicated Code

String representation of this Eulerian graph is:
Superclass dm Subclass1 m1 Subclass1 in Superclass dm Subclass2 m1 Subclass2 in Superclass

2) Association types

The way to resolve *m:n* relationships is to separate these two classes and create two one-to-many (*1:n*) relationships with third intersect class (i.e., association class). We represent *m:n* and *1:n* relationships differently in string representation of a model:

- *as (association)* – for the correct *1:1* or *1:n* relationships (e.g., *Class1 as Class2*) where association class is not needed
- *asMN* (*association m:n*) – for *m:n* relationships (e.g., *Class1 asMN Class2*) *without* association class. These relationships are considered as a smell
- *ac (association class)* – for relationships (e.g., *Class1 ac Class2*) *with* association class.

3) Many methods/attributes/associations

Some design smells are identified by *many* methods, attributes or associations. We add specific edges to the digraph representation for these characteristics:

- *a** – class has *many* attributes (e.g., string representation is *Class1 a* Class1*)
- *m** – class has *many* methods (e.g., *Class1 m* Class1*)

- *as* / asMN* / ac** (without or with association class) – class in role1 has *many* association with classes in role2 (e.g., *ClassRole1 as* ClassRole2*)

Before the runtime of bit-vector algorithm, the value of variable *many* is determined according to an actual system model.

4) The Extended algorithm.

1. Create dictionary for mapping method names from system model and placeholders in string representation (e.g., method *GetColor = m1*).
2. Calculate value of *many* attributes, methods and associations for actual system model.
3. Create digraphs, Eulerian graphs and string representations for design smells and system model.
4. *For* each design smell
   Iterate through *triplets* of *Vertex Edge Vertex* in string of design smell and identify them in system model string representation.
      If Edge is *m**, *a**, *as** or *ac** *Then*
         search for at least *many* (i.e., calculated number) of these edges in a system model.
         List classes matching this condition
         as candidate classes.
         If class does not have *many* edges
            Then remove it from the list.
      If Edge is *mNUMBER Then*
         save this triplet to the *external stack*
         and skip to the next triplet.
      Else search for an *Edge* in the system model.
         If Edge exists,
            Then list classes as candidate classes.
         Else remove them from the candidate list.
5. *Iterate* through external stack with *mNUMBER* edges and check if selected candidate classes meet these conditions by searching them in system model. Remove classes not matching condition from the candidate list.
6. Prepare remaining classes in the list as the candidate classes for the role in design smell.

4.2 The Similarity Scoring Algorithm

1) Names of methods

This algorithm is not suitable for the method name comparison, because they cannot be properly represented in matrices. One of possible solutions is to create $n \times n$ matrix (n is number of classes) for each method contained in model more than once. Then 1 on diagonal in these matrices represents occurrence of method in class (row = column = class). But this method will be very compute-intensive for large models.

2) Association class

We add matrix for *n:m* relationships *without* association class (i.e., design smell), matrix for *1:n* relationships (i.e., associations where association class is not needed) and original association matrix is used for *n:m* relationships *with* association class.

In Fig. 3 is simple model with two relationships (one with and one without association class) and one *m:n* relationship and the matrices for this model:

- Many-to-many relationships without association class
- One-to-many relationships
- Many-to-many relationships with association class

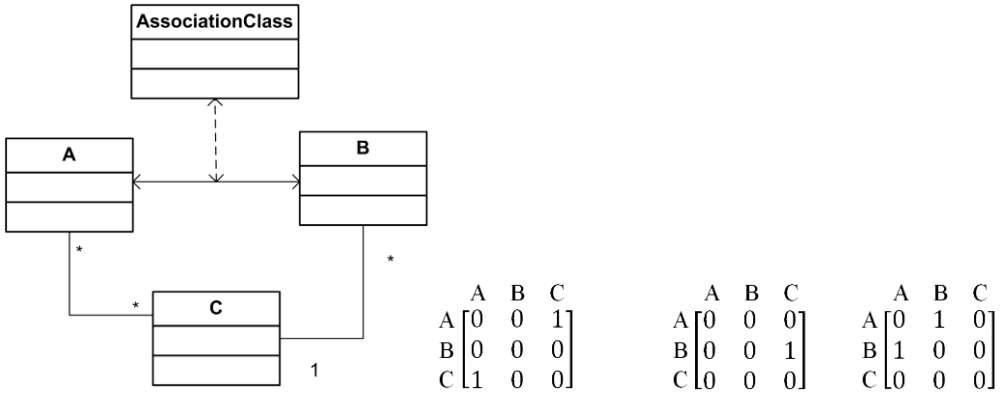

Fig. 3. Simple model with various types of associations and matrices for this model

3) Many methods/attributes/associations

For detection of *many* methods and/or attributes in classes we add matrices for both characteristics. These matrices have number 1 on diagonal where class has at least *many* methods and/or attributes.

It is not necessary to modify original algorithm to detect *many* associations, because associations are in separate matrix, which can be used for this characteristic of a smell (i.e., generate matrix with *many* associations relative to actual system model).

3) The Extended algorithm

1. Calculate value of *many* attributes, methods and associations for actual system model.
2. Create matrices for design smells and system model.
3. *For* each design smell
   For each nonempty matrix of design smell iterate even number of times and stop upon convergence

$$Z_{k+1} = \frac{B \times z_k \times A^T + B^T \times Z_k \times A}{||B \times z_k \times A^T + B^T \times Z_k \times A||_1} \quad (1)$$

   - A and B are matrices for system model and design smell respectively,
   - k is an iteration number,
   - $||...||_1$ is the 1-norm of a matrix,
   - Z_0 is an $n_B \times n_A$ matrix filled with ones and the last value of Z_k is the final similarity matrix.
   - Equation (1) is proposed in [8].
4. Sum all similarity matrices and normalize.
5. Get candidate classes for design smell by comparing values in the final similarity matrix with a threshold.

4.3 Rule-based 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[1] format. This file format can be efficiently handled by the *Acceleo* module from the *Eclipse* environment. Utilizing its capabilities created a *fact generator*, which automatically filters and transforms XMI constructs into *Jess* facts.

Finally, it is necessary to write detection rules. According to Astels [18], the original bad smells proposed by Fowler and Beck [1] can be found in the Class Diagram and Sequence Diagram. Since most of the bad smells are occurring in the UML Class Diagram (*Duplicated Code*, *Large Class*, *Lazy Class*, *Middle Man*, *Data Class*, …), the research focused on work with this group of smells.

1) *The process of identification*

A typical example of a partial bad smell could be shown with the *duplicated code*. In [1] the duplicated code is defined as a code structure that can be found in more than one place in the source code of program. The system is looking for attributes with the same names and types 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.

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.

2) *Rules for bad smells*

The final rules for bad smells are written in the Jess code. Fig. 4 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. 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.

```
(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.na
                           )
    =>
    (if (> ?countSmellSubClass (- ?countAllSubClass ?countSmellSubClass)) then
        (printout t "PARTIAL DUPLICATED CODE : Attribute named \""?attribute.name"\" from the class \""?attribute.nameOfClass"\". This attri
        ;(continue-looking)
        )
    )
```

Fig. 4. Rule of the partial duplicated code - attribute

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* and move of the shared attribute to this new class is more suitable.

4.4 Identification using Syntax Tree

We propose and implement code smell 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 instances of the code smell. Refactoring tool from the smell structure determine appropriate refactoring from predefined catalog.

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 XPath expression over transformed AST and return selected nodes as found smell 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. AST of parsed Java codes and smell catalog are inputs to the search engine. Correction engine can repair found bad smells, if exist appropriate refactorings in predefined catalog. To edit smell catalog we propose graphical Smell editor with XPath generator.

We model the smell with the basic building block of the source code. First we identified main common parts such as *Class*, *Method*, *SwichStatement*, *IfStatement* and *Parameters* needed to model the smell structure. Then we specified attributes *Too Large*, *Too Small* and *Constant Number* called smell symptoms, which will indicate something wrong in the code. Meaning of the smell symptoms depends on its 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*.

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

```
1.  public String getXPath() {
2.          XPathBuilder xpath = new XPathBuilder();
3.          xpath.append(NAVIGATE EXPRESSION);
4.          xpath.append(PREDICATE);
5.          foreach(ISmellPart part)
6.                  xpath.append(part.getXPath(this));
```

```
7.          if(smellSymptom != null)
8.                  xpath.append(smellSymptom.getXPath(this));
9.          xpath.append(SELECT EXPRESSION);
10.  }
```

XPathBuilder composes XPath expression, created in five steps, which can be omitted in dependence on element type. First navigates to particular nodes from the previous node. Nodes can be constrained by predicates such as the methods of type constructor. Next we loop over child nodes of smell and generate subexpression. Actual model allows only one smell symptom in smell part. Finally we 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 illustrated in Fig. 5.

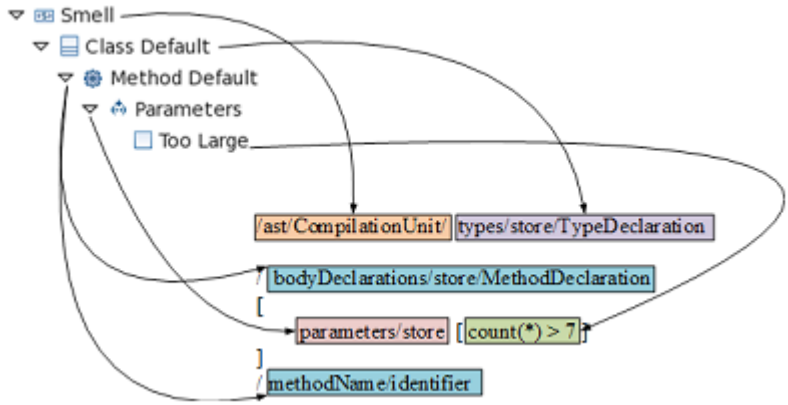

Fig. 5. Long parameter list smell

4.5 Identification using OCL Query

This approach has two phases: the graphical definition of the refactoring rules and their usage (finding the bad smell, defined on the left side of the rule, and applying the transformation script to accomplish/reach the right side of the rule). This concept is on the scheme below (Fig. 6): the user defines the model flaws and their corrections and then the framework automatically generates an OCL [4] query from the model flaw (left side of the refactoring rule) and generates transformation script as the difference between the left and the right side.

The usage or the execution is the second phase of the refactoring. It consists of two processes: the flaw searching and the transformation execution. The system executes the OCL query to find concrete flaws. Result of the query is the collection of the OCL data structure containing all parts and elements of the model flaw. This is the input to the transformation process. Transformation (generated immediately after definition of each rule) is executed with appropriate input components taken from the model flaw instance.

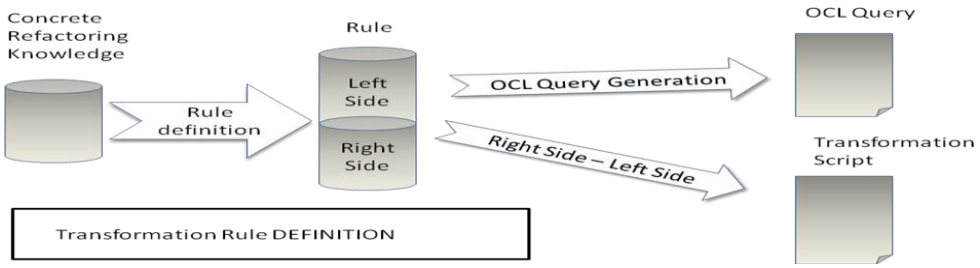

Fig. 6. Conceptual scheme of OCL Query approach.

1) Construction of OCL Query

We decided to use OCL as the query language because of its power, reusability and portability to run on diverse models, diagrams and CASE systems. It is used in many transformation languages. We have developed a generator of OCL queries based on traversing elements of the left side of transformation rule as the input. The output is the executable OCL query with the special structure, applicable on the UML model.

The result of the query contains all appropriate elements in the searched model used as the input to transformation. An instance of the model flaw will be a *tuple* which is one of the complex types in OCL. The tuple has concrete items; each one has a name and the type. A sample definition of the tuple with three items of different types could be as follow:

There is a method to provide all the functionality for the query language with the OCL and UML [19] in conjunction. The first step in our algorithm is the specification of the query result. We iterate through all elements from the left side and according to the cardinality of each element (described in section 4) we define the type of tuple item (single element or collection). Then we generate a body of the query. In the beginning we collect the elements with the single cardinality (1..1) and we use the OCL operation *Element.allInstances()* for the iteration.

Pseudo code of the OCL query generation, written as a procedure with one input parameter – left side of the model, is shown below:

```
String generateQuery(model.leftSide)
   Create result type (ordering elements also)
   Initialize query with first element iteration:
        at least one class must be present in
        model generating"Class.allInstances.."
   while (exists next element) {
        if (next element has single cardinality)
          Add to query "Class.allInstances..."
        else (multi cardinality)
          Add to query
           "let..Set(element-> select.."
   }
   Over sets and elements generated before
        add to query the main condition
   Set temporary variables to result type instance
        add to query final accumulation
   return query
}
```

We continue in the body of the iteration; for another element with the single cardinality we use another iterate operation again; for the element with cardinality 1..n (many) we use the *let* expression to create a temporary variable of collection type containing appropriate elements. After using all nonrelational elements (classes, attributes, methods) we can construct the main

query condition. The relational elements (association and inheritance) are used to complete the select on elements creating the temporary variables or the condition in the *iterate* operation. The main query condition contains collection size restrictions (from element cardinalities). The last part of the query is the end of the first *iterate* expression called *accumulation*.

The query is generated and completed from the string parts. Each part is appended to the actually defined string with the care about bracketing and naming the elements to prevent the collisions and irregular phrases.

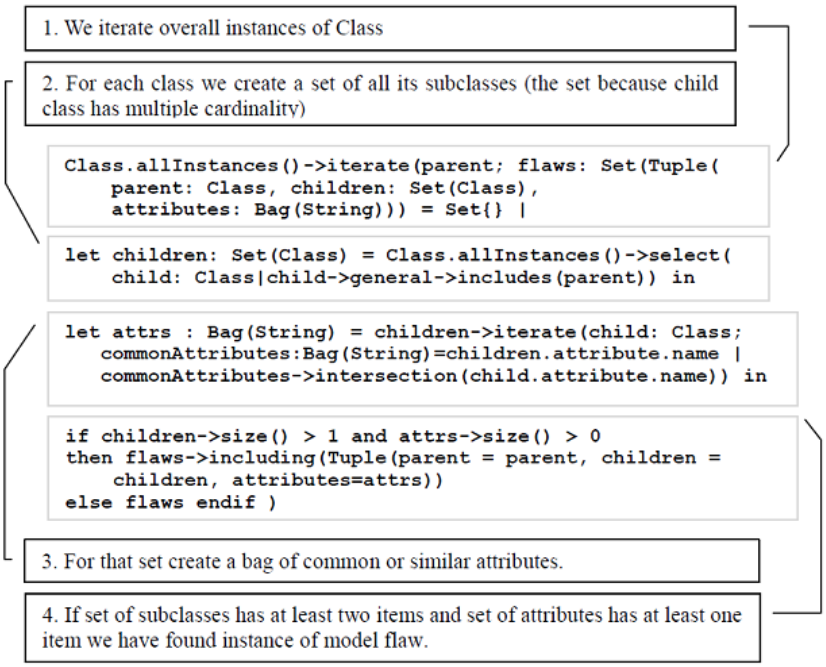

Fig. 7. Example of generated OCL Query

2) *Flaw Correction and Transformation Language Notation*

The transformation of the model flaw to correct structure is the last step of the process of model refactoring. Our concept and language for transformation is inspired by refactoring rules [3, 20, 21] and the basic operations will be *add* or *remove* the element *to* or *from* the specified container. The operation in general can be as follow:

Container(Name) = Container(Name) [operator] Element(Specification)

or in the shortened form: **Container(Name) [operator]= Element(Specification)**

There are some approaches using the OCL as a whole transformation engine. In most cases there are pre and post statements to define flaw and correction [22, 23].

The item *Container* can be *Model* or *Classifier* (covering Class and Interface), and the item *Element* can be *Class*, *Interface*, *Attribute*, *Method*, *Generalization* and *Association*. Operator – or –= remove and operator + or += add the Element to the Container.

The transformation script for refactoring is generated as the difference between the left and the right side (diagram) of the rule: Transformation Script = *Model*(**right** side of the rule) – *Model*(**left** side of the rule)

We use the mapping of the similar elements (models and classifiers) in both diagrams to create the pairs and their differences in their sets of properties, operations, associations, etc. The algorithm of the transformation script generation is as follow:

for each (classifier ***c*** on the right side of the rule)

```
if (not exists the corresponding c in the left
 side)
then Append add classifier c operation for model
 for each (feature f (or relation) of the
   classifier)
       Append add feature f operation
else
 for each (feature f (or relation) of the
   classifier)
 if (not exists the corresponding f
   in the left side)
       Append add feature f operation
for each (feature f in the corresponding class
       on the left side)
 if (not exists the corresponding f in the
       right side)
       Append remove feature f operation
for each (classifier c on the left side
 without corresponding classifier)
 Append remove classifier c operation for model
```

5 Conclusions and the Future Work

Our refactoring framework *VisTra* uses standard UML notation in transformation rules, but we plan to prepare next versions with QVT graph transformation rules grammar or PICS [24] conforming to Viatra and/or EMF Model Transformation Rule[2] [25, 26] generating scripts in readable Viatra2 language.

VisTra concept is open to serve as the *query tool for the seeking* the concrete fragment in the whole project. We can draw very quickly the structure in the left side of the rule and run the OCL query to find requested fragment (for example, all n:m associations without association class), *analysis* [27] or *design pattern* [3], etc.

Another possibility for finding the anti-pattern is the *usage of the XMI protocol*. We can generate query, containing the XPath commands to find the structure in the XMI file exported from the model or the source code. Then we can substitute bad smell elements in the XMI files and use this file as an import to create updated/optimized model.

We can combine *Vistra* with our rule-based system with the collection of predicates of the bad smells and the other prepared approaches from the design pattern domain: *xPath engine*, *Bit-Vector* and *Similarity Scoring Algorithm*.

For the last two algorithms we have tested identification of more design smells simultaneously: *Duplicated Code*, *Large Class*, *Cyclic inheritance*, *Missing Association Class* and *Poltergeist* in the models.

Success rate for both algorithms was relatively high, but Similarity Scoring algorithm was dependant on threshold value. When threshold was too high, success rate decreased and low threshold caused increased number of false positives.

In case of some simple smells these algorithm are *too robust* and time-consuming to prepare data for indentifying them (e.g., creating matrices with values if class has *many* methods, after creating this matrix we already know results and therefore no algorithm is needed). On the other hand some of these simple characteristics could be useful for more complex smells, which have more characteristics and one of them is *many* methods. Algorithms are useful for more complex design smells like *Duplicated Code* and *Poltergeist*.

[2] http://user.cs.tu-berlin.de/_emftrans/

In the future, we can optimize both algorithms with parallel computing methods simultaneously creating the model of the system and the strings/matrices for detecting the model flaws.

We plan to compare effectiveness of these two methods with other approaches and extend their identification ability for more design smells. We need to combine them with another methods we are now implementing (OCLQuery in our VisualTrasform, Abstract Syntax Tree manipulations for source codes, rule based knowledge system in Jess) and integrate them all to our smell detection and refactoring framework, to increase the precission of the detection.

Acknowledgment

This work was supported by 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.

Literature

1. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995. ISBN 0201633612.
2. Koenig, A.: Patterns and Antipatterns, Journal of Object-Oriented Programming 8, pp. 46–48, 1995.
3. Fowler, M.: Refactoring: Improving the Design of Existing Code, Addison-Wesley, (1999). ISBN 0201485672.
4. OMG. Object Constraint Language (OMG OCL), V2.0, 2006.
5. Kaczor, O., Gueheneuc, Y.G., Hamel, S.: Efficient identification of design patterns with bit - vector algorithm, Proceedings of the 10th European Conference on Software Maintenance and Reengineering, pp. 175–184, March 2006. ISBN 0-7695-2536-9.
6. Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., Halkidis, S.T.: Design pattern detection using similarity scoring, in IEEE Transactions on Software Engineering**,** vol. 32, no. 11, pp. 896–909, 2006.
7. Ramaswamy, R., Kencl, L., Iannaccone, G.: Approximate fingerprinting to accelerate pattern matching, Proceedings of the 6th ACM SIGCOMM conference on Internet measurement, Rio de Janeriro, Brazil, October 25-27, 2006.
8. Gueheneuc, Y.G., Sahraoui, H., Zaidi, F.: Fingerprinting Design Patterns, Proceedings of the 11th Working Conference on Reverse Engineering, pp. 172–181, November 08-12, 2004.
9. Dong, J., Lad, D., Zhao, Y.: DP-Miner: Design Pattern Discovery Using Matrix, Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, pp. 371–380, March 26-29, 2007.
10. Blondel, V., Gajardo, A., Heymans, M., Senellart, P., VanDooren, P.:A Measure of Similarity between Graph Vertices: Applications to Synonym Extraction and Web Searching, SIAM Rev., vol. 46, no. 4, pp. 647–666, 2004.
11. Jakubík, J.: Extension for Design Pattern Identification Using Similarity Scoring Algorithm. Information Sciences and Technologies Bulletin of the ACM Slovakia, Vol. 1, No. 1 (2009) 25-32
12. Moha, N., Gueheneuc, Y., Duchien, L., Le Meur, A.: DECOR: A Method for the Specification and Detection of Code and Design Smells, IEEE Transactions on Software Engineering, pp. 20-36, January/February, 2010.

13. Marinescu, C., Marinescu, R., Mihancea, P., Ratiu, D., Wettel, R.: iPlasma: An Integrated Platform for Quality Assessment of Object-Oriented Design," Proceedings of the 21st IEEE International Conference on Software Maintenance - Industrial and Tool volume, ICSM 2005, pp. 77-80, 25-30 September 2005.
14. Fokaefs, M., Tsantalis, N. Chatzigeorgiou, A.: JDeodorant: Identification and Removal of Feature Envy Bad Smells, Proceedings ICSM, pp. 519-520, 2007.
15. Majtás, Ľ.:Contribution to the Creation and Recognition of the Design Patterns Instances. Information Sciences and Technologies Bulletin of the ACM Slovakia, Vol. 3, No. 1 (2011) 84-92
16. 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
17. D. Astels, *Refactoring with UML*. Proc. 3rd Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering, pp. 67-70, 2002
18. Akehurst D., Bordbar B.: On Querying UML data models with OCL. In: Lecture Notes In Computer Science; Vol. 2185, Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools, (2001), pp. 91-103
19. Balogh A., Varro D.: Advanced Model Transformation Language Constructs in the VIATRA2 Framework. In: Proceedings of the 2006 ACM symposium on Applied computing, (2006), pp. 1280 – 1287.
20. Markovic S., Baar T.: Refactoring OCL Annotated UML Class Diagrams. In: Model Driven Engineering Languages and Systems, Lecture Notes in Computer Science 209, Volume 3713. Springer, Berlin, (2005), pp. 280-294.
21. Sunye G., Pollet D., Le Traon Y., Jezequel J.M. Refactoring UML Models. In: Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools, (2001), pp. 134 – 148.
22. Pollet D., Vojtisek D., Jezequel J.: OCL as a Core UML Transformation Language, WITUML 2002 – Position Paper.
23. Baar, T., Whittle, J.: On the Usage of Concrete Syntax in Model Transformation Rules. In: Ershov Memorial Conf. Volume 4378 of LNCS., Springer, Berlin, (2007), pp. 84–97.
24. Biermann, E., Ehrig, K., Köhler, C., Kuhns, G., Taentzer, G., Weiss, E.: Graphical Definition of In-Place Transformations in the Eclipse Modeling Framework. In: 9th Int. Conf. on Model Driven Engineering Languages and Systems, MoDELS'06. Volume 4199 of LNCS., Springer, Berlin, (2006), pp. 425–439.
25. Mens, T.: On the Use of Graph Transformations for Model Refactoring. In: Int. Summer School on Generative and Transformational Techniques in Software Engineering, GTTSE'05. Volume 4143 of LNCS., Springer, Berlin, (2006), pp. 219–257.
26. Fowler, M.: Analysis Patterns: Reusable Object Models, Addison-Wesley, 2000. ISBN 0201895420.

Contact data:
Ivan Polášek, Ing., PhD.
Faculty of Informatics and Information Technologies, Slovak University of Technology in Bratislava, Ilkovičova 3, 842 16, Bratislava 4, Slovakia
polasek@fiit.stuba.sk