A Visual Based Framework for the Model Refactoring Techniques

M. Štolc[*], I. Polášek[*(**)]
[*] Faculty of Informatics and Information Technologies, STU Bratislava, Slovakia
[**] Gratex International, a.s., Bratislava, Slovakia
miroslav.stolc@gmail.com, ivan.polasek@fiit.stuba.sk

Abstract—**Refactoring is one of the most important rules and practices of Extreme Programming from the family of the Agile Methodologies. We propose the tool to refactor the UML model (Class Diagrams for now). In the first step we need to find the flaws (bad smells) in the model with the OCL query and then in the second step we transform the flaw to the correct fragment with the transformation script. The paper presents the set of methods and tools for the model adjustment, cooperating with the CASE systems. We analyze the concept and algorithms for the refactoring, OCL queries and transformation scripts generating. We have prepared functional prototype of the editor for the refactoring rules definition, OCL query generator and the transformation script generator. In the future, we plan to extend the framework with alternative notations (e.g., QVT graph transformation rules, PICS, Viatra2) and the other techniques to find the flaws (e.g., rule-based system with predicates of the bad smells, XMI transformations and Abstract Syntax Tree algebra, Bit-Vector and Similarity Scoring Algorithms).**

I. Introduction

Refactoring, an implicit part of the agile software development life cycle, is mainly performed as a manual change of the source code and/or the software model. This process of changing a software system does not alter the external behaviour [1], but the result of this transformation is a more readable and more effective design of the system. Automation, formalism and validation of refactoring processes are not actually closed. In this paper, we focus on the model refactoring in the UML Class Diagrams.

Our approach is inspired by transformation languages based on QVT [11]. The most popular are ATL and Operational QVT integrated into the Eclipse platform. These languages do not allow a graphical definition of the transformation. There are some frameworks which have a graphical input, but they are using their own languages (graphically oriented) and that fact makes them a little bit unusable.

Operational QVT is the Eclipse implementation of OMG QVT Specification [11]. It was designed for transformations that have to build target models of a complex structure[1].

[1] http://www.eclipse.org/m2m/qvto/doc/M2M-QVTO.pdf

II. Motivation

The problem of a bad structure or solution could be, of course, sometimes subjective, but there are some authorities and resources which are trustworthy worldwide [1, 3, 7]. As an example of a bad solution is the substructure in the upper left part of the UML model depicted in Fig. 1. We can define this situation as follows: *two subclasses of one parent class have the same attribute (method)*. This situation is referred [15] to smell No.5: *Divergent Change*.

The correction of a mentioned structural problem is moving the common attribute to the parent class. It does not change the functional meaning of the model but it simplifies its design.

In general, we can split up the refactoring into the two steps: *identifying* the flaw of the model (bad smell in the source code) and *transformation* of the fragment from the bad state to the better/correct one. To execute this process we need to define the set of rules using the appropriate notation and editor. Also, we need a mechanism for matching the elements in the model and the engine which transforms the acquired elements to the required state or structure.

The solution of our example is in the upper right part of the Fig. 1. The definition of such a *refactoring rule* could be graphically defined as a diagram with two parts: left and right side of the rule. The left side illustrates the bad smell in the model - henceforth referred to as a *model flaw*. The final solution for reorganizing the structure is in the right part of the diagram. The context of these two parts is close to the meaning of the *if* – *then* rule in knowledge systems. The general rule from the concrete structures in the upper left/right corners is presented in the bottom part of the figure. For the cardinality of the instances we use the new, fourth part of the class, for one and only one occurrence of the instance we write 1..1, for the more occurrences of the class in the concrete role we use countenance 1..n (one or more), or 2..n (more than one).

In this context the process of refactoring will iterate for each left side of the rule and find the occurrences in the model and then change these substructures to the correct one. The transformation engine executes that change as a model-to-model transformation (usually we do not transform whole model but only a small part or subset of the model).

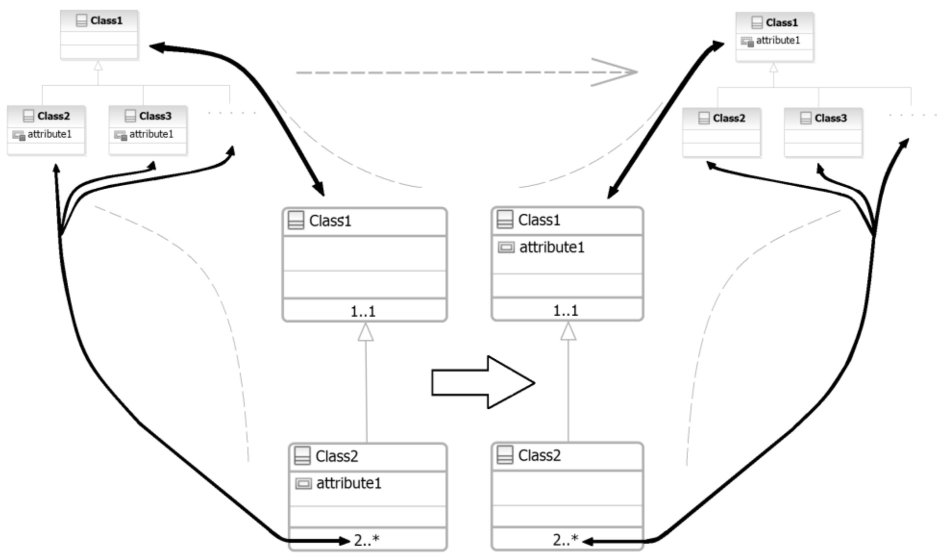


Figure 1. A sample of refactored structure and rule definition.

III. Our Approach of Model Refactoring

Our 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. 2): the user defines the model flaws and their corrections and then the framework automatically generates an OCL [12] 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.

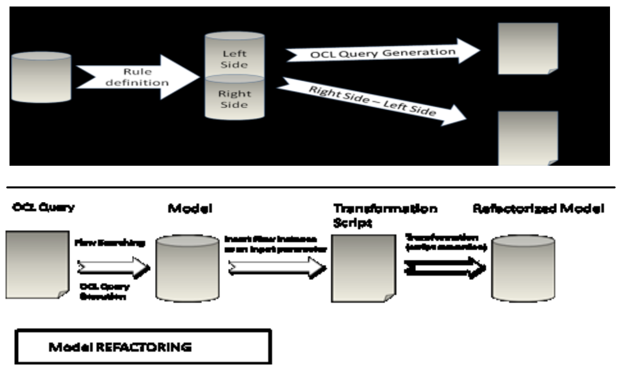


Figure 2. Conceptual scheme of our approach.

We can formalize the process as follow:

```
for each (transformation rule)
      find all flaw instances in the model following
            the left side of the rule
      if (interactive mode)
      then display all flaw instances
            for each (flaw instance)
                  if (the user accept transformation)
                  then execute the transformation
                    script for this instance
      else execute the transformation script for all
        instances
```

IV. Graphical Notation of Refactoring Rules

Graphical notation for refactoring rules is inspired by the UML and its metamodel [10]. The simplified model of the graphical notation is shown in Fig. 3. The base elements are *Classifier*, *Model*, *Relation* and *TransformationRule*. The *TransformationRule* consists of the left *ModelFragment* with the flaw and the right *ModelFragment* with the correct structure.

The main purpose of this graphical language is to prepare the left side of the rule for generating an OCL query and define the transformation to the correct, right side.

There is a new feature compared to the UML metamodel and the standard conventions: we have added the *Cardinality* to the *Classifier* to map more classes or interfaces with the same features or relationships in the diagram. Graphically, cardinality is the fourth part of the classifier element (the first three sections are the name, attributes and methods).

V. 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.

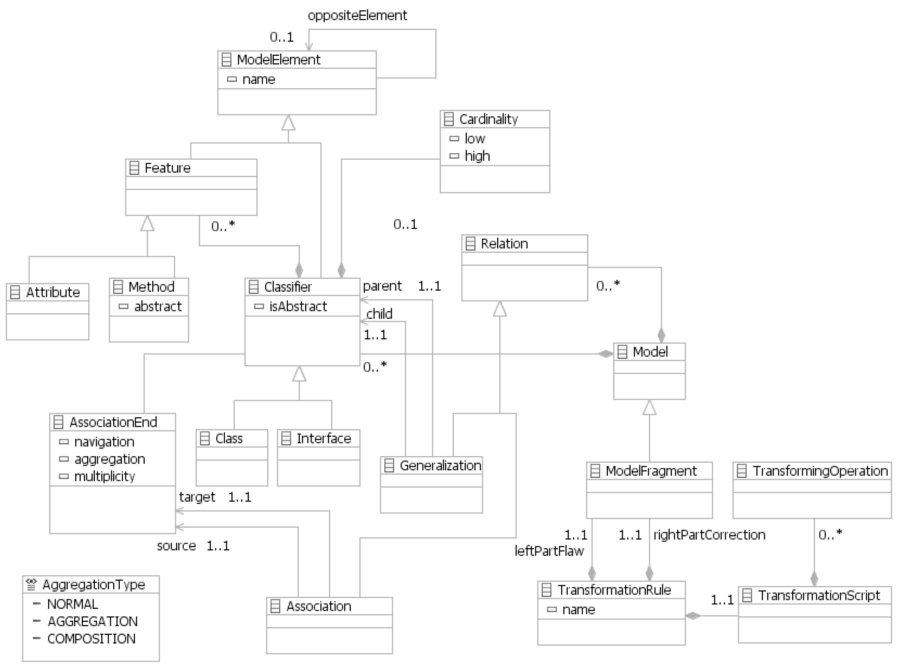


Figure 3. Simplified metamodel of graphical notation.

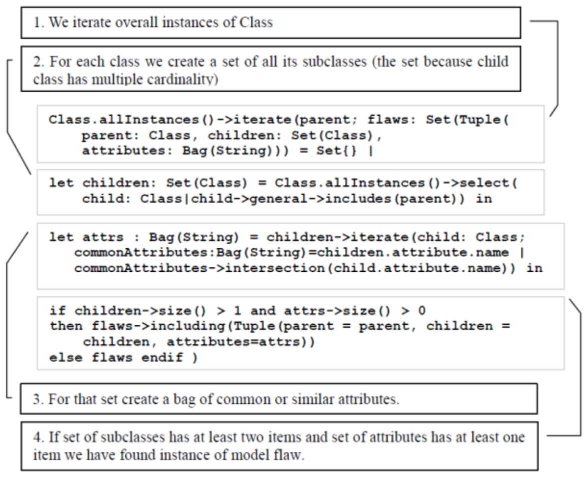
We have developed a generator of OCL queries based on traversing elements of the left side of transformation rule as the input. The output is the executable OCL query with the special structure, applicable on the UML model. The result of the query contains all appropriate elements in the searched model 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:

Tuple(parent:Class, children:Set(uml::Class), attributes:Bag(String))

There is a method to provide all the functionality for the query language with the OCL and UML [8] 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*.



1. We iterate overall instances of Class

2. For each class we create a set of all its subclasses (the set because child class has multiple cardinality)

```
Class.allInstances()->iterate(parent; flaws: Set(Tuple(
    parent: Class, children: Set(Class),
    attributes: Bag(String))) = Set{} |

let children: Set(Class) = Class.allInstances()->select(
    child: Class|child->general->includes(parent)) in

let attrs : Bag(String) = children->iterate(child: Class;
   commonAttributes:Bag(String)=children.attribute.name |
   commonAttributes->intersection(child.attribute.name)) in

if children->size() > 1 and attrs->size() > 0
then flaws->including(Tuple(parent = parent, children =
    children, attributes=attrs))
else flaws endif )
```

3. For that set create a bag of common or similar attributes.

4. If set of subclasses has at least two items and set of attributes has at least one item we have found instance of model flaw.

Figure 4. The OCL query for previous sample.

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.

The example of the query for the model flaw (Fig. 1.) is shown thereinbefore (Fig. 4.). In the introduced, flaw we mention a concept of common or similar attribute. It is difficult to determine whether the attributes are equal. In this moment, we can declare two attributes as similar if they have the same name (and the type).

VI. Model Flaw Searching

We can use the existing set of the OCL queries to find the instances of the model flaw. In Fig. 5 is the real sample of the model illustrated the loan proposal domain. The model is more complex and this picture shows only one fragment of it. The classes EstateEvaluation, ReviewerStatement and FinancialAnalysis have the same attribute *body* and this redundancy is the bad smell in this model.

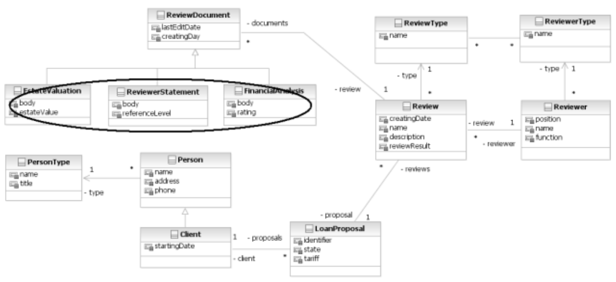


Figure 5. Sample model for a demonstration of OCL query.

Output from the executing of the query looks as follow:

```
[Tuple{
        parent = Class(name: ReviewDocument),
        children = Set{Class(name: EstateEvaluation),
                Class(name: ReviewerStatement),
                Class(name: FinancialAnalysis)},
        attributes = Bag{'body'}
}]
```

The query was executed in the Eclipse using customized EMF interpreter for OCL. The model was exported from CASE tool as instance of UML2.ecore (metamodel).

VII. 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 [1, 2, 5] 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)

The same expression in the shortened form:

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

Element as the concrete component will be specified by its name, but for some types of elements the name is not present (i.e. generalization relation); therefore, it must be specified in another way (for generalization it can be used the join of the type and the names of the source and target class).

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 [4, 9].
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. Transformation script of the example from Fig. 1 would be:

*Classifier(***SubClass***) = Classifier(***SubClass***) – Attribute(***name***)*
*Classifier(***TopClass***) = Classifier(***TopClass***) + Attribute(***name***)*

After matching the real flaw in the model from Fig. 5 we must remove attributes from *EstateEvaluation*, *ReviewerStatement* and *FinancialAnalysis* and then add the copy of removed attribute to *ReviewDocument*:
The executable, prepared script for the correction with the real flaw instance as an input parameter is

*Classifier(***EstateEvaluation***) =*
*Classifier(***EstateEvaluation***) – Attribute(***body***)*
*Classifier(***ReviewerStatement***) =*
*Classifier(***ReviewerStatement***) – Attribute(***body***)*
*Classifier(***FinancialAnalysis***) =*
*Classifier(***FinancialAnalysis***) – Attribute(***body***)*
*Classifier(***ReviewDocument***) =*
*Classifier(***ReviewDocument***) + Attribute(***body***)*

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:

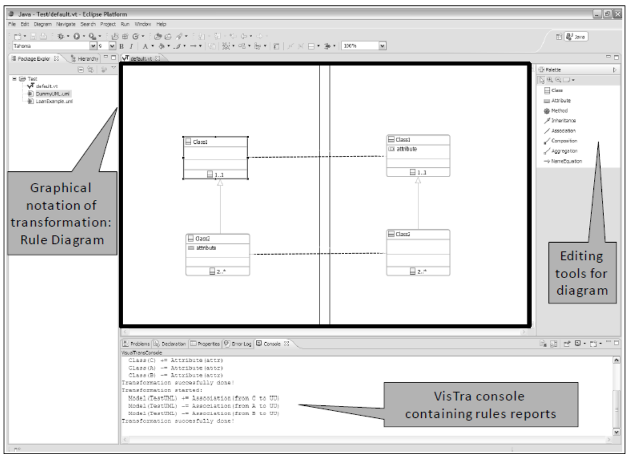
```
for each (classifier c on the right side of the rule)
       if (not exists the corresponding c in the left
         side)
       then Append add classifier c operation for model
         for each (feature f (or relation) of the
           classifier)
                 Append add feature f operation
       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
```

VIII. Framework Concept

Our framework *VisTra* is visual oriented tool for transformation and refactoring of the models. Now it contains two generators and graphical editor of the refactoring rules, based on EMF model, created with GMF framework. GMF as a graphical engine supports simple definition of graphical editor in several XML files to generate the eclipse plugin with the editor logic. The metamodel of the diagram elements used in the editor is defined in EMF ecore model in Fig. 3.

The editor is working with the left and right side diagrams of the *TransformationRule* which induces *TransformationScript* containing the set of *TransformingOperation*, specialized in the middle level to the abstract subclasses *RemoveRule* and *AddRule* which represent two types of the rules according to analysis in section 7. The concrete real subclasses (*RemovePropery, AddAssociation*, etc.) are on the bottom level of the rules hierarchy. Design pattern *Strategy* is used to implement *Execute* methods in the concrete transforming operation.

Only the graphical editor of the rule is visible for the user. The screenshot of the user interface is in Fig. 6. We can see two sides (left and right) of the rule, with their similar elements in each side and mapping lines between elements in a standard eclipse perspective with editor in the centre, toolbar for adding new elements on the right side.


Graphical notation of transformation: Rule Diagram
Editing tools for diagram
VisTra console containing rules reports
Class1
Class2
attribute
1..1
2..*
Problems
Declaration
Properties
Error Log
Console
VisualTransConsole

```
  Class(C) += Attribute(attr)
  Class(A) -= Attribute(attr)
  Class(B) -= Attribute(attr)
Transformation succesfully done!
Transformation started:
  Model(TestUML) += Association(from C to UU)
  Model(TestUML) -= Association(from A to UU)
  Model(TestUML) -= Association(from B to UU)
Transformation succesfully done!
```

Figure 6. *VisTra* editor screenshot as an eclipse plugin.

The scenario for using the editor follows:

- Create the model of the flaw on the left side
- Copy the whole model from left to right side (mapping lines are created automatically) with the simple editor function
- Update or create the right side as the correction of the flaw on the left side
- Save the rule with the simple editor function (framework will automatically generate OCL query and the transformation script)

The editor is developed with EMF and GMF therefore the integration to Eclipse is very simple. The integrator could use SWT[2] and JFace[3] to show transformation catalog – the list of available transformations. The end user can invoke transformation rules on the UML model.

IX. Related Work

Except from the ATL and Operational QVT, there are other approaches with graphical notation: Viatra and the framework Roclet. Viatra framework provides the rule and pattern-based transformation language for manipulating graph models by combining graph transformation and abstract state machines into a single specification paradigm [2]. It is based on the mentioned QVT specification and fulfills its main requirements and chose the VPM (Visual and Precise Metamodeling).

Refactoring can cause incorrect OCL constraints [5] over changed elements. Authors show the table with refactoring rules i.e. *RenameAttribute*, *PullUpAttribute* etc. containing an influence to OCL constraints. The work results in framework for refactoring called RoclET[4]. The framework includes the UML editor, the OCL editor and evaluator and refactoring facilities. Attached UML constraints are automatically updated after refactoring. If it is not possible, the user gets a warning.

The process of refactoring can be summarized in five simple operations [4]: add, remove, mode, generalization and specialization of model elements. Authors present examples especially on refactoring State diagram. Their notation is based on OCL uses pre and post condition of statement. Pre condition describes model flaw and post condition model correction.

Catalog [3] with approximately forty transformations on Class and State UML diagram is divided into the five groups: refinements, quality improvements, elaborations, abstractions and transformation in design patterns. This catalog is used in ATL project as the reference transformations[5]. On the project web pages there are some of them as an example. We also use it as the basis for the testing of the *VisTra*.

X. Conclusions and the Future Work

In this paper, we have presented our approach to refactoring using a graphical definition of the transformation rules (representing model flaw and its optimized structure) and the automatic generation of the OCL queries and the transformation scripts.

In the future, we plan to extend the rule notation with elements for aspect-oriented refactoring [20] and improve editor with new features mainly in graphical notation: naming/renaming the elements, *transitive relationships* [6] and *relation constraints* for the cardinality of the common classes collection in the diagram at the ends of their equivalent relations in composed brackets: {1}, {1..n}, ..

Next UML diagrams in *VisTra* could be the State, Sequence and Use Case Diagrams.

[2] www.eclipse.org/swt/
[3] wiki.eclipse.org/index.php/JFace
[4] http://www.roclet.org/index.php
[5] http://www.eclipse.org/m2m/atl/atlTransformations/

VisTra uses standard UML notation in transformation rules, but we plan to prepare next versions with QVT graph transformation rules grammar or PICS [17] conforming to Viatra and/or EMF Model Transformation Rule[6] [18, 19] 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* [15] or *design pattern* [16], etc.

Another possibility for finding the flaw in the model 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. 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 use *rule-based system* with the collection of predicates of the bad smells to find the fragment (we have the prototype in *Jess*) or the other approaches from the design pattern domain:

1. *Abstract Syntax Tree* (AST) of the code smell can be the basis for the seeking flaws in the model.
2. A further possibility is to create Euler Graph from the flaw, then the string as a fingerprint of the smell and then apply the *Bit-Vector Algorithm* [13].
3. Another alternative is to map the flaw to the matrix and then apply *Similarity Scoring Algorithm* [14].

ACKNOWLEDGMENT

The authors' research on the subject of this contribution is partially supported by the Scientific Grant Agency of the Slovak Republic grant No. VG1/0508/09.

REFERENCES

[1] Fowler M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, (1999). ISBN 0201485672.

[2] 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.

[3] Lano K.: Catalogue of Model Transformations. [Online; accessed April 16th, 2009].
Available at: http://www.dcs.kcl.ac.uk/staff/kcl/tcat.pdf

[4] 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.

[5] 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.

[6] Stein D., Hanenberg S., Unland R.: Query Models. In: Lecture Notes in Computer Science Volume 3273. Springer, Berlin, (2004), pp. 98-112

[7] Opdyke W.: Refactoring object-oriented frameworks. PhD. Thesis, 1992.

[8] 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.

[9] Pollet D., Vojtisek D., Jezequel J.: OCL as a Core UML Transformation Language, WITUML 2002 – Position Paper.

[10] OMG. Unified Modeling Language (OMG UML), Superstructure, V2.2, 2009.

[11] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation, V1.0, 2008.

[12] OMG. Object Constraint Language (OMG OCL), V2.0, 2006.

[13] Kaczor, O., Gueheneuc, Y. G., Hamel, S.: Efficient identification of design patterns with bit - vector algorithm. In: Proceedings of the 10th European Conference on Software Maintenance and Reengineering, March 2006, ISBN 0-7695-2536-9, pp. 175 – 184.

[14] Tsantsalis, N., Chatzigeorgiou, A., Stephanides, G., Halkidis, S., T.: Design Pattern Detection Using Similarity Scoring. In: IEEE Transactions on Software Engineering, volume 32, November 2006, pp. 896 – 909.

[15] Fowler, M.: Analysis Patterns: Reusable Object Models, Addison-Wesley, 2000. ISBN 0201895420.

[16] Gamma, E. et al.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995. ISBN 0201633612.

[17] 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.

[18] 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.

[19] 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.

[20] Vranić, V. et al.: Aspect-Oriented Change Realization and Their Interaction. In: e-Informatica Software Engineering Journal, Volume 3, Issue 1, Wroclaw, 2009. ISSN 1897 7979, pp. 43–58.

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