On Extended Similarity Scoring and Bit-vector Algorithms for Design Smell Detection

I. Polášek*, P. Líška*, J. Kelemen** and J. Lang*

*Faculty of Informatics and Information Technology Slovak University of Technology in Bratislava, Ilkovičova 3, 842 16 Bratislava, Slovakia
** Institute of Informatics and IT4I Center of Excellence, Silesian University, Bezruc Sq. 13, 746 01 Opava, Czech Republic and School of Management, Panonska str. 17, 851 04 Bratislava, Slovakia

polasek@fiit.stuba.sk, peterfoxik@gmail.com, kel10um@axpsu.fpf.slu.cz, lang@fiit.stuba.sk

***Abstract*- The occurrence of design smells or anti-patterns in software models complicate development process and reduce the software quality. The contribution proposes an extension to Similarity Scoring Algorithm and Bit-vector Algorithm, originally used for design patterns detection. This paper summarizes both original approaches, important differences between design patterns and anti-patterns structures, modifications and extensions of algorithms and their application to detect selected design smells.**

***Keywords*- design smell; anti-pattern; refactoring; bit-vector algorithm; similarity scoring**

I. 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.

II. 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.

Authors in [8] proposed an experimental study 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.

Dong et al. in [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.

Our approach is inspired by [5, 6] and our intention is to verify efficiency of these methods for design smells detection, despite differences between design patterns and anti-patterns. However we have to modify and extend the algorithms for this slightly different purpose.

A. 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.

B. 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.

C. 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.

III. 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.

IV. Our Approach: Adapted and Extended Methods for Design Smell Detection

Our main concern is the adaptation of selected methods by extending their searching capabilities for design 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).

A. *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 numbered to differ from other watched methods.

In Figure 1 (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). Figure 1 (b) shows Eulerian Graph for this smell.

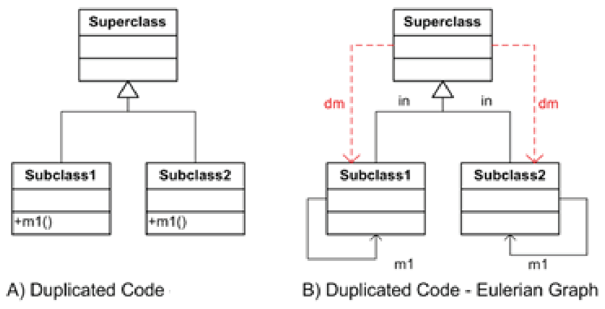


Figure 1 Design smell Duplicated Code

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.
```

B. *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 Figure 2 is simple model with two relationships (one with and one without association class) and one *m:n* relationship.

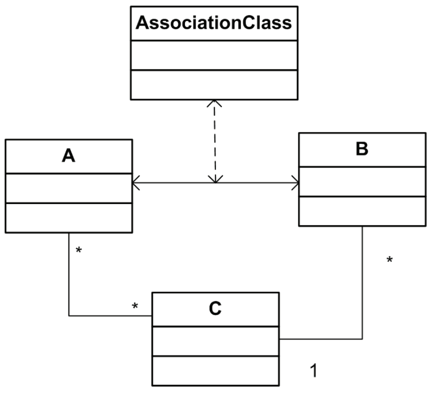


Figure 2 Simple model with various types of associations

Matrices for this model are

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

which looks as follow:

$$\begin{array}{c|ccc} & A & B & C \\ \hline A & 0 & 0 & 1 \\ B & 0 & 0 & 0 \\ C & 1 & 0 & 0 \end{array} \qquad \begin{array}{c|ccc} & A & B & C \\ \hline A & 0 & 0 & 0 \\ B & 0 & 0 & 1 \\ C & 0 & 0 & 0 \end{array} \qquad \begin{array}{c|ccc} & A & B & C \\ \hline A & 0 & 1 & 0 \\ B & 1 & 0 & 0 \\ C & 0 & 0 & 0 \end{array}$$

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).

4) 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.

V. Our First Results (and a Short Evaluation)

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

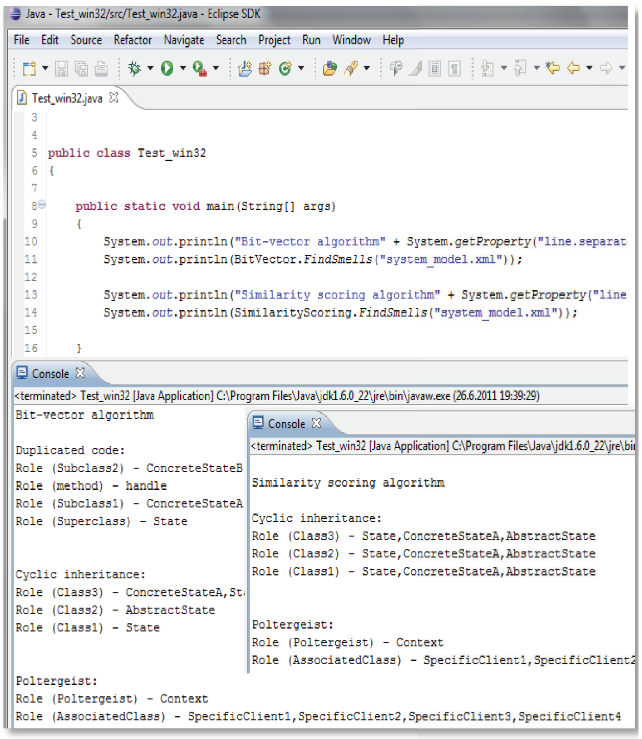
Success rate for both algorithms was relatively high, but Similarity Scoring algorithm was dependant on threshold value. When threshold was too high, success rate decreased and low threshold caused increased number of false positives.

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*. Table 1 summarizes applicability of both algorithms for selected design smells.

TABLE 1.
SELECTED SMELLS AND USABILITY OF EXTENDED ALGORITHMS

Design smell	Description	Bit-vector	Similarity scoring
Duplicated Code	Methods and/or attributes with the same name in different classes.	Usable	-
Large Class	Too many methods and/or attributes in the class.	Usable	Usable
Cyclic inheritance	Classes inherit from other class in a circle.	Usable	Usable
Missing the association class	Association (m:n) without association class.	Usable	Usable
Poltergeist	Unnecessary, inefficient and complex class caused transient and needless invocations. Too many associations with the other classes.	Usable	Usable

In Figure 3 is the frame of our prepared system with the sample of results from both algorithms. Brackets contain the name of class role in the smell and after dash is the list of candidate classes for this role.


Java - Test_win32/src/Test_win32.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Test_win32.java
Console

```java


public class Test_win32
{

    public static void main(String[] args)
    {
        System.out.println("Bit-vector algorithm" + System.getProperty("line.separat
        System.out.println(BitVector.FindSmells("system_model.xml"));

        System.out.println("Similarity scoring algorithm" + System.getProperty("line
        System.out.println(SimilarityScoring.FindSmells("system_model.xml"));

    }
```

```
<terminated> Test_win32 [Java Application] C:\Program Files\Java\jdk1.6.0_22\jre\bin\javaw.exe (26.6.2011 19:39:29)
Bit-vector algorithm

Duplicated code:
Role (Subclass2) - ConcreteStateB
Role (method) - handle
Role (Subclass1) - ConcreteStateA
Role (Superclass) - State


Cyclic inheritance:
Role (Class3) - ConcreteStateA,St
Role (Class2) - AbstractState
Role (Class1) - State


Poltergeist:
Role (Poltergeist) - Context
Role (AssociatedClass) - SpecificClient1,SpecificClient2,SpecificClient3,SpecificClient4
```

```
<terminated> Test_win32 [Java Application] C:\Program Files\Java\jdk1.6.0_22\jre\bi

Similarity scoring algorithm

Cyclic inheritance:
Role (Class3) - State,ConcreteStateA,AbstractState
Role (Class2) - State,ConcreteStateA,AbstractState
Role (Class1) - State,ConcreteStateA,AbstractState


Poltergeist:
Role (Poltergeist) - Context
Role (AssociatedClass) - SpecificClient1,SpecificClient2
```

Figure. 3 The frame of the system and the result samples

VI. Future Work

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 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] 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