DESIGN SMELL DETECTION WITH SIMILARITY SCORING AND FINGERPRINTING: PRELIMINARY STUDY

Peter Líška
Faculty of Informatics and Information Technology
Slovak University of Technology in Bratislava
Ilkovičova 3, 842 16 Bratislava, Slovakia
xliskap3@stuba.sk

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

***Abstract*—Design smells in software models reduce the software quality. Smells identification supports the refactoring, which is a way to improve the quality of models and subsequently increasing software readability, maintainability and extensibility. We propose a preliminary study of using Similarity Scoring Algorithm and Fingerprinting Algorithm for design smells detection. In the future, we plan to do extensive verification on several large projects, integrate these methods to the smells detection framework and compare effectiveness with other approaches.**

Keywords – design smells; anti-patterns; refactoring; algorithms; fingerprinting; similarity scoring

I. INTRODUCTION

Software modeling is a part of almost every software development method and software designers are fallible, therefore designer's carelessness or inexperience may lead to design smells. It could unnecessarily complicate software model and software based on this model may be harder to extend, maintain or it may not be possible to implement correctly at all.

To fix these undesirable structures a refactoring is used. We restructure software by applying a series of refactoring rules without changing its observable behavior [1]. 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. There exist several methods for both steps, but our concern is identification of bad structures, therefore selected methods aim only this part of refactoring.

II. RELATED WORKS

Kaczor et al. in [2] described a technique to identify design patterns in models by using fingerprinting algorithm on string representation of both. Firstly, model and design pattern are transformed into digraphs (directed graphs). Software model is actually graph where vertices are represented as entities 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. The technique was tested with design patterns Abstract Factory and Composite on 3 projects.

Tsantalis et al. in [3] 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 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) readability by computer scientists.

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 [4].

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

III. PROPOSED EXTENSIONS

Our main concern is adaptation of original methods by extending their searching capabilities for design smells detection. Most anti-patterns are simpler and have different structure features, thus other model attributes needs to be compared. We have chosen several smells attributes different from design patterns features, i.e., they cannot be detected by original methods.

Smells have many different characteristics and some of them do not have exact definition, thus when we want to detect them automatically, it is needed to define their characteristics (e.g., what is *many* methods and attributes). On the other hand, some design patterns characteristics are also usable for smells detection.

Structural features included in both extended methods are:

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

978-0-7695-4418-2/11 $26.00 © 2011 IEEE
DOI 10.1109/ECBS-EERC.2011.39

A. Fingerprinting Algorithm

1) Names of methods

First extension adds method names comparison, because some smells may need to identify occurrence of a concrete method in several classes. An edge is added in digraphs for each method. Starting and ending point of this edge is a class where the method is occurring. Name of this edge could be:

- *mX* – for general method.
- *mNUMBER (e.g., m1)* – for specific method which name is considered as important (it is numbered because there can be defined different specific methods).

2) Association class

Many-to-many (m:n) relationships between classes increase model complexity. The way to resolve many-to-many 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 many-to-many and one-to-many relationships differently in string representation of a model:

- *as* (association) – for many-to-many relationships (e.g., *Class1 as Class2*). These relationships are considered as a smell.
- *ac (association class)* – for one-to-many relationships (e.g., *Class1 ac Class2*). These relationships have association class or association class is not needed.

3) Many methods/attributes/associations

Some design smells are identified by *many* methods, attributes or associations. We added 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* / ac* (without or with association class)* – class in role1 has *many* association with classes in role2 (e.g., *ClassRole1 as* ClassRole2*)

Before run of fingerprinting algorithm, value of *many* is determined according to an actual system model.

B. Similarity Scoring Algorithm

1) Names of methods

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). This method could be very compute-intensive for large models.

2) Association class

We added another matrix for one-to-many (1:n) relationships (i.e., relationships with association class or association class is not needed) and original association matrix is used for many-to-many (n:m) relationships.

3) Many methods/attributes/associations

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

It was not needed 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).

IV. Evaluation

We tested identification of design smells *Duplicated Code, Multiple Inheritance, Large Class, Cyclic inheritance, Missing Association Class and Poltergeist* in simple model. Success rate for both algorithms was 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. However some design smells could be identified more effectively by other methods.

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 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*.

Acknowledgment

This work was partially supported by the Scientific Grant Agency of Slovak Republic, grant No. VG1/0971/11/2011-2014. This contribution/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] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, (1999). ISBN 0201485672.

[2] O. Kaczor, Y. G. Gueheneuc, S. Hamel, "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.

[3] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, S. T. Halkidis, "Design pattern detection using similarity scoring," in IEEE Trans. Softw. Eng., vol. 32, no. 11, pp. 896–909, 2006.

[4] Y. G. Gueheneuc, H. Sahraoui, F. Zaidi, "Fingerprinting Design Patterns," Proceedings of the 11th Working Conference on Reverse Engineering, pp. 172–181, November 08-12, 2004.