Decomposing Object-Oriented Class Modules Using an Agglomerative Clustering Technique

Marios Fokaefs
Department of Computing Science
University of Alberta
Edmonton, Canada
fokaefs@cs.ualberta.ca

Nikolaos Tsantalis, Alexander Chatzigeorgiou
Department of Applied Informatics
University of Macedonia
Thessaloniki, Greece
nikos@java.uom.gr, achat@uom.gr

Jörg Sander
Department of Computing Science
University of Alberta
Edmonton, Canada
joerg@cs.ualberta.ca

Abstract

Software can be considered a live entity, as it undergoes many alterations throughout its lifecycle. Furthermore, developers do not usually retain a good design in favor of adding new features, comply with requirements or meet deadlines. For these reasons, code can become rather complex and difficult to understand. More particularly in object-oriented systems, classes may become very large and less cohesive. In order to identify such problematic cases, existing approaches have proposed the use of cohesion metrics. However, while metrics can identify classes with low cohesion, they cannot identify new or independent concepts. Moreover, these methods require a lot of human interpretation to identify the respective design flaws. In this paper, we propose a class decomposition method using an agglomerative clustering algorithm based on the Jaccard distance between class members. Our methodology is able to identify new concepts and rank the solutions according to their impact on the design quality of the system. Finally, our method has been evaluated by two independent designers who were asked to comment on the suggestions produced by our technique on their projects. The designers provided feedback on the ability of the method to identify new concepts and improve the design quality of the system in terms of cohesion.

1. Introduction

Developing software can be a very long procedure. During this period, the code undergoes many changes and manipulations and may become large and complex. In addition, developers usually work under time pressure and may neglect some very important design principles. The violation of these principles results in the appearance of design problems in the code also known as "bad smells" [7]. To remove bad smells from the code Opdyke [14] proposed the process of refactoring. According to this notion, the designer introduces some small formalized changes to the code which are supposed to solve the design problems and at the same time preserve the behavior of the program.

In this paper, we deal with a specific bad smell called "God Class". In object-oriented programming, there is a design principle which suggests that a class should implement only one concept [13]. From the maintenance point of view, this can also mean that a class should have only one reason to change. The violation of this principle results in large and complex classes or "God Classes" as they are more commonly known. This violation can occur in two ways: either a class holds a lot of the system's data in terms of number of attributes (Data God Class) or it has a great portion of the system's functionality in terms of number and complexity of methods (Behavioral God Class). In the first case, we can either redistribute the attributes of the "God Class" or move functionality (*i.e.,* methods) from other classes closer to the data. In the second case, we can either move functionality from the "God Class" closer to the data of other classes or split the class by extracting a cohesive and inde-

978-1-4244-4828-9/09/$25.00 2009 IEEE

pendent piece of functionality [7] [2]. The latter is a refactoring called "Extract Class".

In this work, we aim at identifying "Extract Class" opportunities by employing a clustering technique. As Tzerpos and Holt [23] suggest, clustering methods have a great potential of being used in various Software Engineering fields. Wiggerts [24] gives a detailed survey of clustering methods and shows how they are or can be used for software remodularization. More specifically, clustering methods can identify conceptually meaningful groups of similar entities [20]. Therefore, in this paper we evaluate the performance of a hierarchical agglomerative clustering technique on the "God Class" problem. The intuition behind using clustering in this case is that clusters may represent cohesive groups of class members (methods and attributes) that have a distinct functionality and can be extracted as separate classes.

The contributions of the proposed methodology are the following:

1. Identification of new concepts. We define a concept as a distinct entity or abstraction for which a single class provides a description.

2. Suggestion of behavior preserving refactoring solutions for the identified problematic classes.

3. Ranking of the identified refactoring opportunities based on their impact on the design quality as measured by the Entity Placement metric [22]

To evaluate the proposed methodology we have extended an existing design flaw identification tool called JDeodorant [6]. This tool, which is implemented as a plug-in for the Eclipse platform, parses the code using an AST (Abstract Syntax Tree) parser and identifies bad smells. The methodology has been evaluated by two independent designers, who provided feedback on the refactoring opportunities identified for their systems. The designers assessed the potential of the methodology to identify new concepts and improve the design quality of their systems in terms of cohesion.

The rest of the paper is organized as follows: Section 2 provides an overview of the related work. In Section 3, we analyze the proposed methodology in depth. The results of the evaluation are presented and discussed in Section 4 and finally Section 5 concludes the paper.

2. Related Work

Several research works have dealt with the problem of identifying God classes. Trifu and Marinescu [21] propose a metrics-based method. They define God classes as "large, non-cohesive classes that have access to many foreign data" and use a formula based on complexity, cohesion and coupling metrics to determine whether a class is problematic or not. This method suffers from the fact that the thresholds for the metrics are empirically or statistically determined and thus may differ from system to system.

Tahvildari and Kontogiannis [19] propose two *quality design heuristics* and use a diagnosis algorithm based on complexity, cohesion and coupling metrics to identify design flaws. In this case, the thresholds are less tight and vaguely defined (high/low) and may require human input.

DuBois *et al.* [4] propose a metrics oriented approach meaning that their main aim is to improve the metrics rather than use them as an identification criterion. For the identification, they use "guidelines" based on conceptual and macroscopic criteria. For the God classes, the respective guideline says: "Separate the responsibilities. Extract those groups of methods and attributes that neither use nor are used by other methods or attributes". This method offers no automation whatsoever and the guidelines are not formalized in a way that would allow a degree of automation.

Finally, Demeyer *et al.* [2] suggest some conceptual criteria to identify the God classes. According to this approach, a god class is a low cohesive and memory consuming class. It usually has abstract names like "Controller", "Manager", "Driver" or "System". Any change to the system may lead to a change of this class. It is often called the "heart of the system". In most of the cases, it is the hardest class to maintain.

All of the above methods might be able to identify problematic classes, but it remains uncertain whether they can produce conceptually correct improvement suggestions which are meaningful to the designer.

In the Software Reengineering literature, there have been many works on remodularizing software modules. Mancoridis *et al.* [11] propose a novel methodology for remodularizing a software system. The aim is to produce good clusters in terms of high cohesiveness (within the clusters) and low coupling (between the clusters). They mainly use hill-climbing and genetic algorithms. They produce the Module Dependency Graph based on the source code and then apply clustering on the resulting graph. They employ the Modularization Quality measure to evaluate the clusters that the algorithm produces. This measure favors Intra-connectivity and penalizes Inter-connectivity. After finding a sub-optimal partition using the hill-climbing and genetic algorithms, they build a hierarchy of the clusters using the Hierarchical clustering algorithm.

Doval *et al.* [3] consider the problem of identifying a good partitioning as an optimization problem. They propose a genetic algorithm as a means to partition large software systems using as an objective function the Modularization Quality measure defined in [11]. In a similar work, Shokoufandeh *et al.* [17] apply a spectral clustering algo-

rithm in order to remodularize a system. They also view this process as an optimization problem using the Modularization Quality measure as an objective function.

The main difference between the philosophy behind these works and our approach, lies in the fact that they result in a single solution that is close to the optimal one, which the designer should accept or reject in its entirety. On the contrary, the proposed method is essentially a stepwise approach, that extracts a set of refactoring suggestions ranked by an appropriate metric.This offers the advantage of gradual change of a system, allowing the designer to assess the conceptual integrity of the refactoring suggestions at each step.

Sartipi and Kontogiannis [16] propose a semi-supervised clustering framework for recovering the software architecture. They analyze the source code to retrieve the component similarity, they cluster the components and, finally, the user assigns the remaining modules to their closest clusters or reallocates the modules among the clusters. They employ the maximal association property (i.e., maximum number of shared features) to introduce two new similarity measures, namely association between entities and mutual association between components. During the clustering phase the user may select among a set of main seeds, around which the new cluster will be built, or manually create a cluster.

All of the above methodologies propose remodularization of software modules in a higher level, like package or file level. Our method focuses on software remodularization on a class level.

Simon *et al.* [18] suggest that visualization techniques can be used to identify Extract Class opportunities. This methodology defines dependency sets for each type of class members (attributes and methods) in order to calculate the Jaccard distance between class members. Using mapping techniques, the entities are visually presented and then it is upon the designer to decide whether there is an opportunity to extract a class or not. The issue with visualization is that the real spatial structure of the classes is unknown. Moreover, visualizing large classes can overwhelm the designer and make it difficult for him to identify clear partitions. Our methodology goes one step further by identifying clusters of cohesive entities and presenting them to the designer ranked according to their impact on the design of the whole system.

In a recent work, Joshi and Joshi [9] consider the problem of classes with low cohesion as a graph partitioning problem. They focus on improving class cohesion by examining lattices based on the dependencies between attributes and methods. A shortcoming of this method, as identified by the authors, is that for large systems the lattices can become very complex and thus it is more difficult for the designer to inspect the lattice visually and identify problematic cases. Moreover, while this method focuses on improving the cohesion of a class, it neglects the conceptual criteria of the suggestions. The classes suggested to be extracted can only contain methods, which are insufficient to describe a concept. Finally, there is no indication that the suggested refactorings will not affect the behavior of the program.

Finally, De Lucia *et al.* [10] propose a methodology that takes into account both structural and conceptual criteria. They build a weighted graph of the class methods based on structural and semantic cohesion metrics, which then is split using a MaxFlow-MinCut algorithm to produce more cohesive classes. However, the semantic cohesion metric is based on the names of classes and entities which can be arbitrary and thus the results highly depend on the naming policies used by the developers of a project. Furthermore, by bipartitioning the graph it is possible to loose potential clusters. For example, a class might consist of more than two cohesive subclasses which could not be identified by splitting the class. Finally, the attributes are not considered during the calculation of the graph, but they are moved to the extracted class. This might have undesirable effects on the coupling of the system.

3. Methodology

The goal of the proposed methodology is the identification of Extract Class refactoring opportunities. The methodology is applied to every class of a system regardless of its cohesion. In this way, there is no need for defining thresholds according to which a class will be examined or not. In addition, a single threshold might not be sufficient to identify all problematic classes.

The identification of the Extract Class opportunities consists of two main parts. Firstly, we use dependency information from the code of the examined project in order to calculate distances between class members and we apply a clustering algorithm to identify cohesive groups of entities that can be extracted as separate classes. Secondly, we employ a set of rules which assure that the classes suggested to be extracted have a certain degree of functionality and the suggested refactorings preserve the behavior of the program. Finally, we rank the suggested refactorings according to their impact on the design quality of the system.

3.1 Clustering

The clustering algorithm we use is a hierarchical agglomerative algorithm. This algorithm works as follows: Firstly, it assigns each entity to a single cluster. In each iteration it merges the two closest clusters. Finally, the algorithm terminates when all entities are contained in a single cluster. Eventually, the outcome of the algorithm is a hierarchy of clusters. To determine the actual clusters we need to choose a threshold value for the minimum distance as a

cut-off value. The hierarchy of the clusters is usually represented by a dendrogram. An example of a dendrogram is shown in Figure 3. The leaves of the tree represent the entities, the root is the final cluster and the intermediate nodes are the actual clusters. The height of the tree represents the different levels of the distance threshold in which two clusters were merged.

In a hierarchical agglomerative clustering algorithm there are two aspects that one should pay attention to. The first is the linkage method and the second is the distance threshold.

There are several methods used to determine the closest clusters including the maximum distance between the members of a clusters (*complete linkage*), the average distance (*average linkage*) or the minimum distance (*single linkage*). Maqbool and Babri [12] also propose the Weighted Combined Algorithm as a linkage method. This algorithm takes into account the strength of the dependency between two entities (*i.e.,* the number of times an entity is used by another entity). Although this approach may be good for remodularization in higher level, in lower levels we consider the presence of a dependency to be more important than its strength. According to Anquetil and Lethbridge [1] complete linkage favors more cohesive clusters, the single linkage less coupled clusters and average linkage is somewhere in-between. As our method is based on class member dependencies, the cohesion of the newly created classes is expected to be of a fair or very good level. In other words, as all entities will be connected, our algorithm is guaranteed to produce fairly cohesive classes. Furthermore, cohesion is a controlled variable due to the distance threshold that will be discussed next. In contrast with cohesion, coupling is an uncontrolled variable. By selecting the single linkage method, we have put emphasis on coupling so that the newly created class be as loosely coupled as possible to the original one.

In our methodology, we do not define a fixed threshold for the minimum distance, on the contrary we apply the algorithm for several threshold values (ranging from 0.1 to 0.9) and we present to the user all the possible results. To minimize the amount of information given to the user we exclude any duplicate suggestion. A duplicate suggestion occurs when two clusters, which were produced using a different threshold value, contain exactly the same entities. However, we do not exclude similar suggestions (*i.e.,* when the entities of a cluster are a subset of another cluster's entities) because it is not certain whether a higher threshold can produce better results than a lower one or vice versa.

We chose the agglomerative method over a partitioning algorithm like K-means, because it is not easy to select the number of clusters, while a distance threshold can be easily estimated. K-means, as it is traditionally used in data mining, requires that the entities are placed in a feature space (*i.e.,* space dimensions correspond to the data object attributes). However, the spatial structure of a class cannot be estimated accurately. This is because it is not known how many dimensions are sufficient to map class members in space and what these dimensions represent. Furthermore testing every single possible value for k can deteriorate the performance of the methodology dramatically. Partitioning algorithms are also not robust to noise. By noise we mean entities that are too far from the others and cannot be included in any cluster. As it turns out object-oriented classes usually produce rather sparse similarity matrices (*i.e.,* a lot of zero values) which correspond to a large amount of noise.

We did not choose a density-based algorithm because density based algorithms are usually more parameterized. A density based algorithm, like DBSCAN [5], needs two parameters: the ε-neighborhood which defines a radius of a point around which a dense subgroup (not a cluster) can be defined and the $MinPts$ which corresponds to the minimum points that need to exist in a subgroup. Then, the subgroups are iteratively merged to form the dense clusters. While we could define an ε value similar to the distance threshold, it is not clear what the minimum number of entities in a neighborhood should be.

The distance metric we use is the Jaccard distance, which according to Anquetil and Lethbridge [1] produces good results in software remodularization. To define the Jaccard distance between two class members we employ the notion of *entity sets* defined in [22]. According to this concept the entity set of an attribute contains all the methods that access this attribute and the entity set of a method contains all the methods that are invoked by this method and all the attibutes that are accessed by it. We extend the definition of the entity set so that it also contains the entity itself. We do that so that we can preserve the condition that says that $d_{ij} = 0$ iff $i = j$ where d_{ij} is the Jaccard distance between entities i and j. Without this extension, then two different entities that access or were accessed by the same other entities (*i.e.,* their entity sets were equal) would have a zero distance. We also include attributes which are references to other classes in the entity set of a method. A reference is essentially a pipeline through which foreign entities are accessed. Since we are interested in examining a class as a closed environment we consider references as local attributes and we exclude from the entity sets of class members any foreign entities that might be accessed through references. Based on the defined of the entity sets we calculate the Jaccard distance between two entities α and β with entity sets A and B respectively as follows:

$$d_{\alpha,\beta} = 1 - \frac{|A \cap B|}{|A \cup B|} \quad (1)$$

To better understand the methodology, we will illustrate its application on a simple synthetic example shown in Figure 1. In this example, we have a class with four attributes:

```
public class Person {

    private String name;
    private String job;
    private String officeAreaCode;
    private String officeNumber;

    public void changeJob(String newJob) {
        if(!newJob.equals(job)) {
            this.job = newJob;
        }
        name += ", "+newJob;
    }

    public void modifyName(String newName) {
        if(!newName.equals(name)) {
            this.name = newName;
        }
        job = newName + ", " + job;
    }

    public String getTelephoneNumber() {
        String phone = officeAreaCode+"-"+officeNumber;
        name += ", "+phone;
        job += ", "+phone;
        return phone;
    }
}
```

Figure 1: Source Code of a Synthetic Example

- a1 = name
- a2 = job
- a3 = officeAreaCode
- a4 = officeNumber

and three methods:

- m1 = changeJob
- m2 = modifyName
- m3 = getTelephoneNumber

Table 1 shows the distance matrix for this example and Figure 2 shows a graph representation of the class. In this graph, the nodes represent attributes and methods and the edges indicate that a dependency exists between two entities. Furthermore, in this graph representation the length of the edges is proportional to the distances between the class members. It is clear from the graph that there are two distinct clusters shown in circles. We were able to identify both clusters using the hierarchical algorithm and a distance threshold of 0.65. Figure 3 shows the dendrogram produced by the clustering algorithm.

3.2 Filtering the results

According to Martin Fowler's definition refactoring is a technique that alters the internal structure of the system

Table 1: Distance Matrix for the Class of Figure 1.

	a1	a2	a3	a4	m1	m2
a2	0.4					
a3	0.8	0.8				
a4	0.8	0.8	0.67			
m1	0.6	0.6	1	1		
m2	0.6	0.6	1	1	0.5	
m3	0.71	0.71	0.6	0.6	0.67	0.67

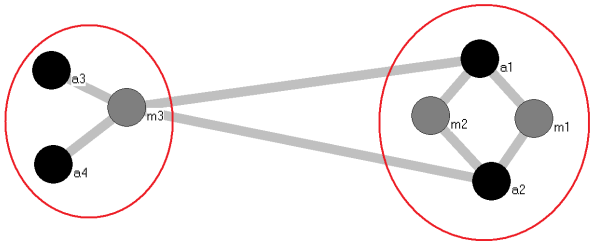

Figure 2: Graph corresponding to the Class of Figure 1.

but not its external behavior [7]. To this end, the proposed methodology makes sure that the classes suggested to be extracted have a certain degree of functionality and, at the same time, the suggested refactorings preserve the behavior of the program.

According to the preconditions required for assuring a certain degree of functionality, the class suggested to be extracted should:

- contain more than one entity. A single member cannot describe a concept sufficiently enough;
- contain at least one method. Data (*i.e.,* attributes) might be sufficient to identify a concept, but functionality (*i.e.,* methods) is essential for the definition of a class.

According to the preconditions required for behavior preservation issues, the class suggested to be extracted should:

- not contain a method that overrides any abstract or concrete method of the super class of the source class;
- not contain a method that makes any super method invocations;
- not contain a method that is synchronized or a method that contains a synchronized block since the move of a synchronized method could create concurrency problems to the objects of the source class

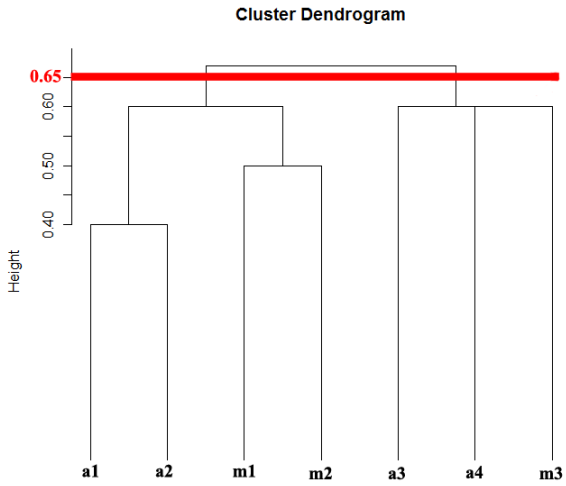

Figure 3: Dendrogram Resulting from the Application of Hierarchical Algorithm for the Class of Figure 1.

Violation of the behavior preservation preconditions might introduce compilation errors to the code or alter the external behavior of the program.

To give an estimate of the impact of each suggested solution on the design quality of the program, we use a novel metric called *Entity Placement* (EP) [22], in order to rank the refactoring suggestions and furthermore to filter out the suggestions that have a negative impact, *i.e.,* deteriorate the Entity Placement value of the original system. Entity Placement is a metric that combines the notions of coupling and cohesion. It is also based on Jaccard distance using entity sets, thus making it compliant to our method.

In the calculation of EP according to [22], the entity set of an attribute a contains all the methods that access attribute a, the entity set of a method m contains all the methods that are invoked by method m and all the attibutes that are accessed by m and, finally, the entity set of a class C contains all the methods and attributes that belong to C. Entity sets do not include:

- access methods (getters/setters)
- static attributes and methods
- delegate methods
- library entities

The value of Entity Placement for class C is the ratio of its average distance from the entities that belong to class C to its average distance from the entities that do not belong to class C. The respective formula is:

$$EntityPlacement_C = \frac{\frac{\sum_{e_i \in C} distance(e_i, C)}{|entities \in C|}}{\frac{\sum_{e_i \notin C} distance(e_i, C)}{|entities \notin C|}} \quad (2)$$

The value of the Entity Placement for a system is the weighted average of the Entity Placement values of the classes belonging to the system. The formula is:

$$EP_{System} = \sum_{C_i} \frac{|entities \in C_i|}{|all\ entities|} EP_{C_i} \quad (3)$$

To calculate the EP value without having to actually apply the refactoring on the source code, we apply it virtually according to the following procedure:

1. We create a new empty class.
2. For each extracted entity we change its origin class from the source class to the new class.
3. We update the entity sets of all the entities that access or are accessed by the extracted entitities.
4. We insert the extracted entities in the entity set of the new class.
5. We remove the extracted entities from the entity set of the source class.

Finally, the suggestions are presented to the user sorted in ascending order according to Entity Placement; the lower the value, the more positive the impact to the design quality of the system.

4. Evaluation

We applied our methodology to two projects and asked the designers to give us their feedback. The designers were asked to answer the following questions:

1. **Can you find a name for the cluster of entities suggested to be extracted?** This will indicate whether the method was able to identify a new concept.
2. **Would you apply the proposed refactoring?**

4.1 eRisk

eRisk is an electronic adaptation of the well-known board game, developed by a group of undergraduate students as part of a Software Engineering course. As undergraduate students are usually less experienced, the project, as expected, did not have a high design quality, a fact that posed certain challenges to our method. While we were

able to recover a large number of problematic cases, there were also some less meaningful suggestions mainly due to the lack of design discipline. For example, concepts are not well defined and they are highly coupled with each other.

We applied our method for seven threshold values ranging from 0.1 to 0.7. For thresholds higher than 0.7 the methodology just kept increasing the size of the already proposed clusters thus making them less meaningful. We also did not consider suggestions that would cause the Entity Placement value of the system to deteriorate. We asked one of designers of the project to provide feedback on the suggested refactoring opportunitits. The results are summarized in Table 2.

Table 2: Results for eRisk

Total Suggestions	Assigned Names	Applied
37	28	16

Out of the total 37 suggestions (without duplicates) the designer of eRisk was able to assign a name to 28 (75,6%) of the suggested clusters. This percentage is encouraging and can show that our method is able to identify new concepts and eventually improve the understandability of the code. However, not all of these classes were finally qualified to be applied by the designer.

In the second question, the designer of eRisk answered that he would apply 33 (43,2%) of the suggested refactorings. The main reason for accepting a suggestion was that it actually proposed a new concept, which should indeed be extracted in a separate class. Our methodology also identified two cases where the extracted entities were completely disconnected from the rest of the class (*i.e.,* they did not use or were used by any other entity of the class) which is a clear example of possible extraction.

The designer rejected the remaining the suggestions mainly on the ground that the proposed change would not have a significant impact on the desing quality or the undestandability of the code. Other reasons for rejecting a suggestion were that the proposed class did not describe a separate concept or that such a change would unnecessarily increase the coupling of the system. The designer also rejected several suggestions produced with a high threshold value, as they would extract a large portion of the data or the functionality of the source class. Finally, they were some cases that the designer claimed that he would apply the suggested refactoring if some more entities were included in the extracted class. However, this problem was solved as the threshold value was gradually increased.

During the application of our methodology on eRisk and the discussion with its designer, we were able to observe some very interesting by-products. In some cases, from the classes suggested to be extracted, the designer was able to identify that some of them could be reused in multiple places. An interesting outcome of this result was that the proposed classes were more like utilities. While they had a relatively low correlation to the general concept of the project, they could be used by several other classes in calculations or initializations.

In several cases, we also observed opportunities to extract an abstract superclass or an interface. In many of the GUI classes of the system the designer would use a method that would initialize the attributes of a container (panel or frame) and a method that would nullify the attributes to release memory resources. While the methods were slightly different from class to class, they had the same role, thus it would make sense to extract them in a higher level of the class hierarchy.

Furthermore, we were able to identify opportunities to move entities to another class instead of extracting them to a new one. This approach was supported by the fact that the designer assigned a name of an already existing class to the newly extracted one. Finally, there was a case where we identified an opportunity to extract an inner class. While the designer agreed that the extracted entities indeed constituted a different concept, the application of such a refactoring would dramatically increase the coupling of the system. Therefore, extracting the entities into an inner class would retain the original coupling of the system.

4.2 SelfPlanner

SelfPlanner [15] is an intelligent web-based calendar application that plans the tasks of a user using an adaptation of the Squeaky Wheel Optimization framework. It is the outcome of a research project of the Artificial Intelligence Group at the department of Applied Informatics, University of Macedonia, Greece. It consists of a planning engine developed in C++ and a client/server application developed in Java.

SelfPlanner can be considered as a rather mature project, since it has been constantly evolving for more than two years. Its developer is an experienced programmer who has a deep knowledge of object-oriented design principles. These facts justify the small number of Extract Class refactoring opportunities that were identified for SelfPlanner. More specifically, the application of our methodology resulted in 18 suggestions (excluding duplicates) with threshold values ranging from 0.3 to 0.7 (using value 0.1 as increment step). Threshold values below 0.3 (*i.e.,* 0.1 and 0.2) and over 0.7 (*i.e.,* 0.8 and 0.9) did not produce any results. Moreover, the application of 4 out of the total 18 suggestions would result in a higher Entity Placement metric value compared to the value corresponding to the current system (*i.e.,* it would deteriorate the design quality of the system as

measured by Entity Placement metric), and as a result these suggestions were excluded from the evaluation. The results are summarized in Table 3.

Table 3: Results for SelfPlanner

Total Suggestions	Assigned Names	Applied
14	12	9

The independent designer was able to assign a name to 12 (86%) clusters of entities that were suggested to be extracted as separate classes. The employed clustering technique was able to capture groups of methods that not only accessed common fields but more importantly had relevant functionality. For example, there was a case that the employed clustering algorithm successfully grouped all the entities which were related with the functionality of the *Subject* role in an Observer pattern instance [8]. The grouped entities were actually a field holding the *collection of Observers*, two methods playing the role of *attach* and *detach* operations, as well as a method playing the role of *notify* operation [8].

From the twelve conceptually meaningful Extract Class refactoring opportunities that were identified for SelfPlanner, the independent designer supported that 9 of them (64% in total) would have a positive impact on design quality if they were applied in source code. According to the independent designer, the reasons for adopting these specific refactoring suggestions are that they lead to the creation of new classes with sufficient functionality and at the same time they do not introduce extensive coupling between the source and extracted classes.

5. Conclusions

In this paper, we proposed a novel approach for identifying potential Extract Class refactoring opportunity using a hierarchical agglomerative clustering algorithm based on the Jaccard distance between class members. Our methodology is able to identify new concepts that can be extracted as separate classes. Instead of just identifying problematic cases, our methodology also proposes behavior preserving refactoring solutions to address such cases. The suggestions are ranked using the Entity Placement metric in order to assist the user to understand the impact of each refactoring on the design quality of the system.

Our evaluation process showed that our methodology can produce meaningful and conceptually correct suggestions and extract new concepts. In particular, for the two projects that our approach was applied to, it was able to identify a relatively large number of new concepts (75,6% and 86% respectively) that can be potentially extracted in new classes. Finally, the two designers argued that they would apply a decent number of the suggested refactorings (43,2% and 64%) as they would improve the understandability of their code and would facilitate the process of maintenance.

6. Acknowledgment

The authors would like to thank Dr Eleni Stroulia for her insightful comments and the two independent designers, Anastasios Alexiadis and Athanasios Monopavlidis for their invaluable assistance.

References

[1] N. Anquetil and T. Lethbridge. Experiments with Clustering as a Software Remodularization Method. In *WCRE '99: Proceedings of the 6th Working Conference on Reverse Engineering*, 1999.

[2] S. Demeyer, S. Ducasse, and O. M. Nierstrasz. *Object-Oriented Reengineering Patters*. Morgan Kaufman Publishers, 2002.

[3] D. Doval, S. Mancoridis, and B. S. Mitchell. Automatic Clustering of Software Systems Using a Genetic Algorithm. *Proceedings of the 5th International Conference on Software Tools and Engineering Practice*, 30 August - 2 September 1999.

[4] B. DuBois, S. Demeyer, and J. Verelst. Refactoring - Improving Coupling and Cohesion of Existing Code. *Proceedings of the 11th Working Conference on Reverse Engineering*, pages 144–151, November 8-12 2004.

[5] M. Ester, H. P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial database with noise. *International Conference on Knowledge Discovery in Databases and Data Mining*, August 1996.

[6] M. Fokaefs, N. Tsantalis, and A. Chatzigeorgiou. JDeodorant: Identification and Removal of Feature Envy Bad Smells. *23rd International Conference on Software Maintenance*, pages 519–520, October 2-5 2007.

[7] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring Improving the Design of Existing Code*. Addison Wesley, Boston, MA, 1999.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Boston, MA, 1995.

[9] P. Joshi and R. K. Joshi. Concept Analysis for Class Cohesion. *13rd European Conference on Software Maintenance and Reengineering*, pages 237–240, March 24-27 2009.

[10] A. D. Lucia, R. Oliveto, and L. Vorraro. Using Structural and Semantic Metrics to Improve Class Cohesion. *24th IEEE International Conference on Software Maintenance*, 2008.

[11] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner. Using Automatic Clustering to Produce High-Level System Organizations of Source Code. *Proceedings*

of the 6th International Workshop on Program Comprehension, pages 45–52, 1998.

[12] O. Maqbool and H. A. Babri. The Weighted Combined Algorithm: A Linkage Algorithm for Software Clustering. In *CSMR '04: Proceedings of the 8th European Conference on Software Maintenance and Reengineering*, pages 15–24. IEEE Press, March 2004.

[13] R. C. Martin. *Agile Software Developement: Principles, Patterns and Practices*. Prentice Hall, Upper Saddle River, NJ, 2003.

[14] W. F. Opdyke. Refactoring object-oriented frameworks. Ph.D. dissertation, 1992.

[15] I. Refanidis and A. Alexiadis. SelfPlanner: Planning your time! *ICAPS 2008 Workshop on Scheduling and Planning Applications*, 2008.

[16] K. Sartipi and K. Kontogiannis. Component Clustering Based on Maximal Association. *Proceedings of the IEEE Working Conference on Reverse Engineering*, October 2001.

[17] A. Shokoufandeh, S. Mancoridis, T. Denton, and M. Maycock. Spectral and Meta-Heuristic Algorithms for Software Clustering. *Journal of Systems and Software*, 77(3):213–223, September 2005.

[18] F. Simon, F. Steinbruckner, and C. Lewrentz. Metrics Based Refactoring. *Proceedings of the 5th European Conference on Software Maintenance and Reengineering*, pages 30–38, 2001.

[19] L. Tahvildari and K. Kontogiannis. A Metric-Based Approach to Enhance Design Quality Through Meta-Pattern Tranformations. *Proceedings of the 7th European Conference on Software Maintenance and Reengineering*, pages 183–192, March 26-28 2003.

[20] P. N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. Addison-Wesley, 2005.

[21] A. Trifu and R. Marinescu. Diagnosing Design Problems in Object Oriented Systems. *Proceedings of the 12th Working Conference on Reverse Engineering*, 2005.

[22] N. Tsantalis and A. Chatzigeorgiou. Identification of Move Method Refactoring Opportunities. *IEEE Transactions on Software Engineering*, 35(3):347–367, May/June 2009.

[23] V. Tzerpos and R. C. Holt. Software Botryology: Automatic Clustering of Software Systems. *Proceedings of the International Workshop on Large-Scale Software Composition*, 1998.

[24] T. A. Wiggerts. Using Clustering Algorithms in Legacy Systems Remodularization. In *WCRE '97: Proceedings of the 4th Working Conference on Reverse Engineering*, 1997.