If debugging is the process of removing bugs, then programming must be the process of putting them in.

– Edsger W. Dijkstra.

University of Alberta

IDENTIFICATION AND APPLICATION OF EXTRACT CLASS REFACTORINGS IN OBJECT-ORIENTED SYSTEMS

by

Marios-Eleftherios Fokaefs

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

Examining Committee

Eleni Stroulia, Computing Science

Ken Wong, Computing Science

Marek Reformat, Electrical and Computer Engineering

To my parents

Abstract

Software can be considered a live entity, as it undergoes many alterations throughout its lifecycle. Therefore, code can become rather complex and difficult to understand. More specifically 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. While metrics can identify classes with low cohesion, they usually cannot identify new or independent concepts. In this work, we propose a class decomposition method using an clustering algorithm based on the Jaccard distance between class members. The methodology is able to identify new concepts and rank the solutions according to their impact on the design quality of the system. The methodology was evaluated in terms of assessment by designers, expert assessment and metrics. The evaluation showed the ability of the method to identify new recognizable concepts and improve the design quality of the underlying system.

Acknowledgements

I would like to thank my supervisor Dr Eleni Stroulia who gave me the chance to bring my research onto the next level and broaden my professional and personal horizons. I am also grateful to her for giving me the opportunity to work right into the heart of Software Engineering research.

I am also grateful to my former supervisor Dr Alexander Chatzigeorgiou for introducing me into the wonderful world of research and academia. He has been a great mentor to me and this work has become what it is also thanks to his experience and insightful comments.

Special thanks are reserved to Dr Nikos Tsantalis who has been a mentor but most importantly a great friend. In spite of not having a teacher-student relationship, I owe to Nikos most of my knowledge around software engineering and research.

I would also like to thank all of my friends who have patiently listened to me rumbling about my work.

Last but not least, I owe a great amount of gratitude to my family who have supported me in every way conceivable.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software is being developed throughout its life cycle and may constantly change even well after its release. These changes may be *perfective*, *i.e.,* changes that improve the software, *e.g.,* by adding new features, *corrective*, *i.e.,* changes that repair defects or remove bugs, *adaptive*, *i.e.,* changes to adapt to changing environments, *e.g.,* new operating systems or new programming language features, and,*preventive*, *i.e.,* changes that improve the maintainability of the software. Furthermore, changes can occur because the development team or the programming practices may evolve.

Due to these changes the initial design of the system may deviate from its original rationale. If the changes are not properly monitored and the design is not adapted properly some very important design principles may be violated. This violation results in the appearance of design problems in the code also known as "bad smells" [10]. To remove bad smells from the code Opdyke [19] 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.

This work deals with a specific bad smell called "God Class" [10]. In object-oriented programming, there is a design principle which suggests that a class should implement only one concept [18]. From a 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 complex, unwieldy, inelegant, difficult to understand and maintain modules, which in the case of object-oriented software may manifest as large and non-cohesive "God Classes". 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 Lazy 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, the designer 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, the designer can either move functionality from the "God Class" closer to the data of other classes or split the class by extracting a cohesive and independent piece of functionality [10] [5]. The latter is a refactoring called "Extract Class".

We aim at identifying "Extract Class" opportunities by employing a clustering technique. As Tzerpos and Holt [28] suggest, clustering methods have a great potential of being used in various Software Engineering fields. Wiggerts [30] 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 [25]. 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 identification of a problematic God class and the choice of a specific refactoring to improve it are not trivial tasks and can require the investment of substantial effort by the developers. Even when a case requiring the application of the Extract Class refactoring has been identified, its actual application to the code base is not trivial considering the need of ensuring (a) the syntactic correctness of the codebase and (b) the preservation of the original (*i.e.,* before the refactoring) observable behavior of the system. This is why it is desirable to have IDEs support the process of suggesting on potentially beneficial Extract Class refactorings and applying them. Unfortunately, such support is not currently available for Extract Class refactorings. For example, Eclipse[1] allows the user to extract only the attributes in a new class and then move the methods by performing the Move Method refactoring individually, thus, introducing some unnecessary changes. Therefore, the identification methodology and the application of the refactoring were implemented in an Eclipse

[1]At the time the paper was written, Eclipse was in version 3.6.

plugin that addresses this need; namely, it automatically identifies Extract Class refactoring opportunities, orders them according to their anticipated benefits to the system design, and after the developer selects one, it applies it in a fully automated manner.

The contributions of this thesis 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. Using a clustering algorithm, which has the ability to identify *conceptually* related groups of entities, the proposed methodology tries to identify distinct concepts interleaved with one or more other concepts and defined in a single class. The goal of the method is to distinguish the various concepts and separate them in different classes. The identification methodology is discussed in Section 3.1 and its ability to identify new concepts is evaluated by human assessment in Sections 5.1 and 5.2.

2. *Behavior preserving refactoring solutions for the identified problematic classes.* When performing refactorings, one has to make sure that the changes will not affect the functionality of the system or the syntactic correctness of the code. In these lines, the suggested refactorings are subject to preconditions, which are discussed in Section 3.1.3, in order to ensure that no problems will be caused to the system. Furthermore, instead of suggesting a series of refactorings as *one* optimal solution, the developer has the possibility to select one refactoring each time and apply them in a stepwise manner. This is because in Extract Class refactorings, a suggestion might be a good design solution but might not agree with the developer's conceptual criteria. This is in contrast to works discussed in Section 2.2.

3. *Ranking of the identified refactoring opportunities* based on their impact on the design quality as measured by the *Entity Placement* metric [27]. Because a set of suggested refactorings is provided by the method, it is important to guide the developers as they decide which suggestion should be preferred.

To rank the suggestions, the methodology employs Entity Placement, a combined metric that can express coupling and cohesion. In Section 5.3, we evaluate its ability to quantify the impact on the design of the suggested refactoring.

4. *Automatic application of a selected refactoring*, so that it preserves the syntactic correctness of the system and its external behavior. While the identification of the Extract Class refactoring opportunities is a problem, the application of a refactoring is yet another important issue. In order to preserve the syntax and the behavior of the system, the developer has to check many things and make a lot of changes across the project, if the refactoring was to be performed manually. The mechanics for the application of an Extract Class refactoring are described in Section 3.2.

The rest of thesis is organized as follows.

Chapter 2 presents a review of the related literature. We discuss works that use metrics, heuristics or visualizations to identify problematic classes in object-oriented systems, works that use clustering methods or concept analysis to remodularize software systems, recover architecture or transform a system from procedural to object-oriented and works that employ data mining techniques to identify key classes in a system.

Chapter 3 describes the identification of refactoring opportunities and the mechanics of the application of the suggested refactorings in details. We discuss the clustering technique used and explain why it was preferred over others. Then, we go on to discuss in detail the various parameters of the clustering algorithm: distance metric, distance threshold, linkage method. Then, we talk about the filtering of the suggested refactorings based on conceptual and behavior preserving preconditions. The chapter continues with a discussion about Entity Placement: its definition and how it was used to rank the results. Finally, we present all the changes that need to be performed to apply an Extract Class refactoring.

Chapter 4 presents the details about the implementation of the Eclipse plugin. We discuss all the libraries and tools that were required to implement the application of the refactorings. The chapter also presents an example of using the tool on a real project. It describes all the steps from the identification of refactoring opportunities to the application of the suggested refactorings.

Chapter5 presents the results of the evaluation process. We evaluated the proposed methodology by testing it on various systems. First, it was applied on two small systems and asked directly the designers for their opinion on the suggested refactorings. Second, we tried the methodology on JHotDraw (version 5.3), a larger and more extensively evaluated software. We applied a series of refactorings suggested by the tool and then asked a professional, who specializes in software quality evaluation, to tell us whether the suggestions constituted good design solutions and improved the understandability of the system. Third, we measured the impact of the applied refactorings on the design quality of the system using traditional coupling and cohesion metrics and compared the results with the entity-placement measurements.

Chapter6 concludes this work, summarizing its main points and the results of the evaluation process as well as discussing our future plans.

Chapter 2

Related Work

This chapter reviews the related literature and divides it in three groups. The first group is about identification of problematic classes with low cohesion or key classes that have a big portion of the system's functionality and are intensively maintained. The second group is about works that have used clustering in software remodularization, architecture recovery or migration of legacy systems. The final group is about works that identify Extract Class opportunities and suggest extraction solutions.

2.1 Identification of God Classes

Several research works have dealt with the problem of identifying God classes. Trifu and Marinescu [26] 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 [24] 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.* [7] propose a metrics oriented approach, where 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.* [5] 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.

Zaidman and Demeyer [31] propose a method to identify key classes in object oriented systems based on coupling. They employ HITS,a webmining algorithm. This algorithm has the ability to incorporate the indirect coupling between classes in the calculation of the total coupling. This is achieved by calculating the transitive closure of the class relationships. Their ultimate objective is to improve the understandability of the code and help new developers become familiar with a system by exploring the key classes.

In a similar work, Chatzigeorgiou [4] goes one step further as to illustrate what would be the effect of decomposing the key class on the overall quality of the system. He proposes to redistribute the functionality of the key class to the rest of the system. While both approaches (Move Method and Extract Class) constitute valid solutions to the Feature Envy bad smell according to Fowler, in the case discussed here, the decision whether the entities suggested to be extracted will actually be extracted in a new class or moved to an existing class of the system is left to the designer.

2.2 Clustering in Software Reengineering

In the Software Reengineering literature, there have been many works on remodularizing software systems. Mancoridis *et al.* [16] propose a novel methodology for remodularizing a software system. The aim is to produce good clusters in terms of high cohesion (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.* [6] 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 [16]. In a similar work, Shokoufandeh *et al.* [22] apply a spectral clustering algorithm in order to remodularize a system. They also view this process as an optimization problem using the Modularization Quality measure as an objective function.

Sartipi and Kontogiannis [21] 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.

van Deursen and Kuipers [29] are using clustering to identify objects in the concept of migrating legacy systems in object technology. They also use concept analysis to solve the same problem and they have identified two shortcomings of clustering: an element can exist only in one cluster and an element can arbitrarily go to different clusters in different runs of the clustering algorithm. In the problem presented here neither issue is a shortcoming. For the first, elements are preferred to be in only one cluster because attributes and methods should be declared only in one class. For the second, if an element is equally close to two clusters these clusters will eventually be merged in a higher threshold value and thus will be presented as a suggested extraction. Another difference between cluster and concept analysis is that the latter can identify all possible partitions. Up to a degree this problem is overcome by running the algorithm for multiple threshold values and merging the results accordingly.

2.3 Identification of Extract Class opportunities

Simon *et al.* [23] 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. The methodology proposed in this thesis 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 [12] 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.* [15] 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.

2.4 Remarks

Compared to the works of the first group, the goal of the proposed method is not to identify God classes. Every class of the system is examined with the aim to find those classes who can potentially benefit from an extraction, assuming that more than one distinct concepts are defined in the body of the class. If a suggested extraction improves the overall quality of the system in terms of cohesion and coupling, it is presented to the user. This way, a good opportunity can be identified even in a class which did not initially seem a problematic one. Moreover, these works stop at the identification of the problems and do not suggest any solutions, while the presented work offers a complete solution from identifying the opportunities to suggesting proper refactorings and finally applying these refactorings in an automatic manner.

The works in the second group propose remodularization of software modules in a higher level, like packages. The proposed method focuses on software remodularization on a class level. The criteria appropriate for these two types of remodularization are different. Classes may be reorganized into packages in order to have their releases coincide (as new features are requested from them during maintenance). Or they may be organized according to inheritance hierarchies. On the other hand, the requirement for reorganizing attributes and methods into new classes is to better express and communicate the conceptual model of the application domain.

Another difference between the philosophy behind these works and the proposed approach, lies in the fact that they suggest 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.

Chapter 3

Methodology

The goal of the proposed methodology is to improve the design and the understandability of an object-oriented system by means of identification and application of Extract Class refactorings. In this chapter, we discuss in details the identification methodology including all the issues we had to face with clustering (algorithm, distance metric, distance thresholds). We also describe how the illegal candidate classes were eliminated using preconditions and how the suggested refactorings were ranked using the Entity Placement metric. Finally, we describe the mechanics for applying an Extract Class refactoring.

3.1 Identification of Extract Class opportunities

The identification of the Extract Class opportunities consists of three parts. Firstly, dependency information from the code of the examined project is used in order to calculate distances between class members and then a clustering algorithm is applied to identify cohesive groups of entities that can be extracted as separate classes. Secondly, a set of rules is employed which ensure that the classes suggested to be extracted have a certain degree of functionality and the suggested refactorings preserve the behavior of the program. Finally, the suggested refactorings are ranked according to their impact on the design quality of the system.

The identification method 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. After all, a single threshold might not be sufficient to identify all problematic classes.

3.1.1 Identification of candidate classes

Although in the process many clustering algorithms were considered, the goal was to find an algorithm that would require minimum human input and minimum a priori knowledge on the dataset.

A partitioning algorithm like K-means was considered at first. In this family of algorithms the aim is to separate the entities in a number of clusters. However, this requires that the number of final clusters is known beforehand. In the problem of class partitioning, one cannot be sure about this number. 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 trying every single possible value for k can deteriorate the performance of the methodology dramatically. Partitioning algorithms are also not robust to noise, *i.e.,* 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.

Second, a density-based algorithm was considered since they are more robust to noise and require no knowledge about the structure of the dataset. The aim here is to identify dense areas of entities in the dataset. However, these algorithms are usually more parameterized. A density based algorithm, like DBSCAN [8], 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 an ε value can be defined like the distance threshold, it is not clear what the minimum number of entities in a neighborhood (*i.e.,* the minimum number of members in a class) should be.

Finally, spectral clustering algorithms were considered, since they require no input. These algorithms assume the existence of a graph and the aim is to partition this graph. However, such techniques work better with a weighted graph where the weight of each edge can be the aggregate number of uses (attribute accesses or method invocations) between the connected entities. In the Extract Class problem, the existence of a dependency is more important than its strength. In [22], a spectral clustering algorithm with unary weights is used, but it is combined with optimization techniques. Such option was not preferred since the aim of the methodology is to suggest a set of good design solutions rather than a single optimal partitioning solution.

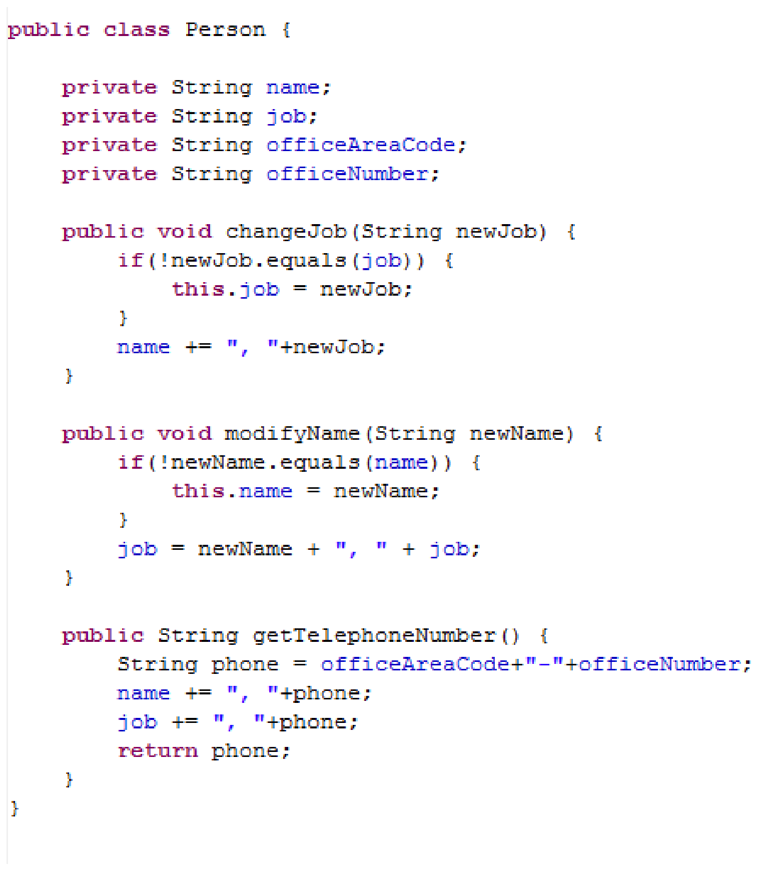
Eventually, a hierarchical agglomerative algorithm was used. This algorithm works as follows: Firstly, it assigns each entity to a single cluster. In each iteration it merges the two closest clusters. Finally, the algorithm terminates when all entities are contained in a single cluster. Eventually, the outcome of the algorithm is a hierarchy of clusters. To determine the actual clusters a threshold value for the minimum distance must be chosen 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 3.3. 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, namely the distance criteria according to which two clusters will be merged and the second is the distance threshold, namely the maximum distance above which no more clusters can be merged.

There are several methods used to determine the closest clusters including the maximum distance between the members of two clusters (*complete linkage*), the average distance (*average linkage*) or the minimum distance (*single linkage*). Maqbool and Babri [17] 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 the presence of a dependency is considered to be more important than its strength. According to Anquetil and Lethbridge [2] complete linkage favors more cohesive clusters, the single linkage less coupled clusters and average linkage is somewhere in-between. As this 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, the 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, the emphasis is on coupling so that the newly created class be as loosely coupled as possible to the original one.

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

The distance metric used is the Jaccard distance, which according to Anquetil and Lethbridge [2] produces good results in software remodularization. To define the Jaccard distance between two class members the notion of *entity sets* is employed as defined in [27]. 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. The definition of the entity set is extended so that it also contains the entity itself. This is done so that the condition $d_{ij} = 0$ iff $i = j$ where d_{ij} is the Jaccard distance between entities i and j is preserved. Without this extension, the 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. Attributes which are refer-



```
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 3.1: Source Code of a Synthetic Example

ences to other classes are also included in the entity set of a method. A reference is essentially a pipeline through which foreign entities are accessed. Since the goal of the methodology is to examine a class as a closed environment references are considered as local attributes and any foreign entities that might be accessed through references are excluded from the entity sets of class members. Based on the defined of the entity sets, the Jaccard distance between two entities α and β with entity sets A and B respectively is calculated as follows:

$$d_{\alpha,\beta} = 1 - \frac{|A \cap B|}{|A \cup B|} \tag{3.1}$$

3.1.2 An illustrative example

To better understand the methodology, we will illustrate its application on a simple synthetic example shown in Figure 3.1. This example has a class with four attributes:

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

and three methods:

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

Table 3.1 shows the distance matrix for this example and Figure 3.2 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. The methodology identified both clusters. Figure 3.3 shows the dendrogram produced by the clustering algorithm.

Table 3.1: Distance Matrix for the Class of Figure 3.1.

	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

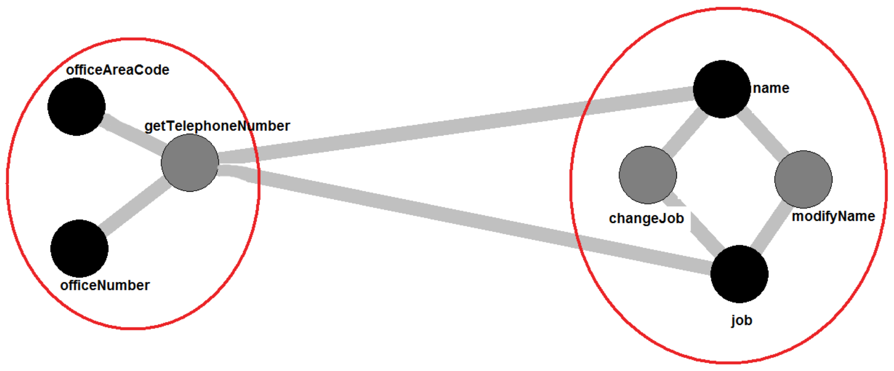


Figure 3.2: Graph corresponding to the Class of Figure 3.1.

3.1.3 Eliminating illegal candidate classes

According to Martin Fowler's definition refactoring is a technique that alters the internal structure of the system but not its external behavior [10]. 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;

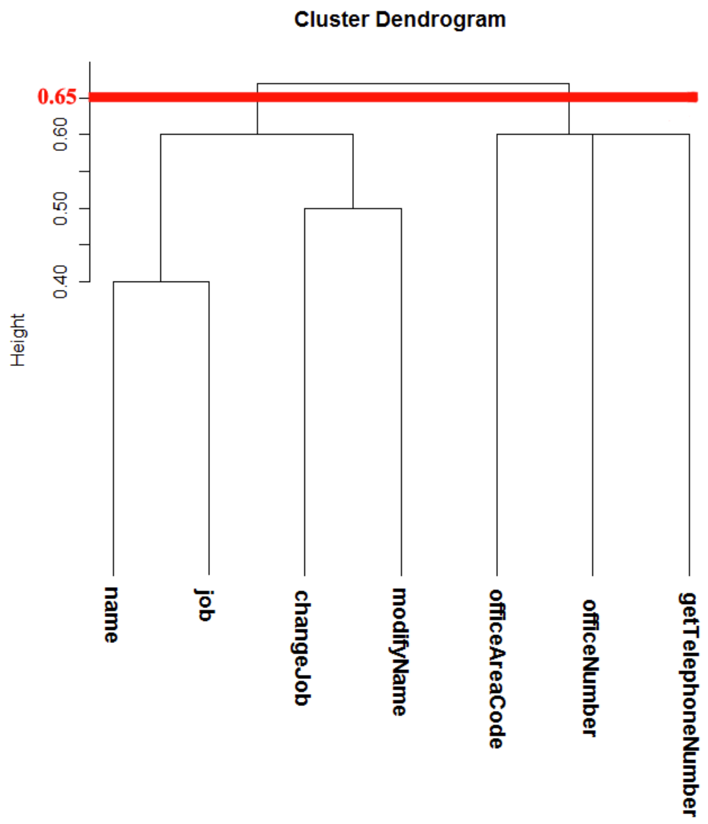


Figure 3.3: Dendrogram Resulting from the Application of Hierarchical Algorithm for the Class of Figure 3.1.

- 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

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

3.1.4 Ranking of the suggested refactorings

To give an estimate of the impact of each suggested solution on the design quality of the program, a novel metric called *Entity Placement* [27] is used, 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 calculates the distances of the entities belonging to a class from the class itself (cohesion of the class) divided by the distances of the entities not belonging to the class from the class itself (the coupling of the class). The entity-placement value for a system is the weighted average of the class values. The reason Entity Placement was chosen over traditional cohesion and coupling metrics is exactly its combinatorial nature. When performing an Extract Class refactoring it is anticipated that the cohesion of the system will increase (since a group of strongly related entities is extracted) and its coupling to deteriorate (because a new class with dependencies to other classes is introduced). A refactoring is a good design solution if the improvement of cohesion is significantly greater than the deterioration of coupling. Entity Placement can achieve that as it evaluates the overall design quality of the system in terms of coupling and cohesion. It is also based on Jaccard distance using entity sets, thus making it compliant to to the identification method.

In the calculation of EP according to [27], 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 entity-placement value 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:

$$EntityPlacement_C = \frac{\frac{\sum_{e_i \in C} distance(e_i, C)}{|entities \in C|}}{\frac{\sum_{e_i \notin C} distance(e_i, C)}{|entities \notin C|}} \tag{3.2}$$

The entity-placement value for a system is the weighted average of the entity-placement values of the classes belonging to the system. The formula is:

$$EP_{System} = \sum_{C_i} \frac{|entities \in C_i|}{|all\ entities|} EP_{C_i} \tag{3.3}$$

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

1. Create an empty class entity set.
2. For each extracted entity change its origin class from the source class to the new class.
3. Update the entity sets of all the entities that access or are accessed by the extracted entities.
4. Insert the extracted entities in the entity set of the new class.
5. 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.

3.2 Application of the Extract Class Refactorings

Refactoring, as defined by Fowler [10], is a local change, which alters the internal structure of the system's code base but does not affect its observable behavior. To ensure that the latter constraint is fulfilled, a set of preconditions is defined that ensure - up to a certain degree - the stability of the system's functionality as discussed in the previous chapter. At the same time, it must be ensured that the changes effectuated by the refactoring do not introduce any syntactic errors.

To apply the refactoring Eclipse's Java Development Toolkit (JDT) was used. JDT offers the ability to deconstruct the code into the corresponding abstract syntax tree (AST). Then it is easy to manipulate the tree by adding, deleting or changing nodes. ASTRewrite (a special JDT class) helps monitor the changes, stores them in a queue and then performs them directly on the source code. To preview the changes the Preview Wizard in Eclipse Language Toolkit (LTK) was used. The mechanics of the refactoring are summarized in Algorithm 1.

Algorithm 1 Extract Class Mechanics

```
 1: Remove extracted entities from the source class.
 2: Create the new class.
 3: Add the required import declarations.
 4: for all the extracted fields do
 5:     Add the extracted field in the new class.
 6:     Create public accessors (getters and setters) for the extracted fields in the
        new class.
 7: end for
 8: for all the extracted methods do
 9:     if there exists an assignment of a field of the source class or an invocation of
        a method of the source class then
10:         Add a parameter of source class type to the extracted class.
11:         Replace field assignments with setters.
12:         Modify the method invocations to source class so that they are invoked
            through the introduced parameter.
13:         Replace "this" with the parameter.
14:     end if
15:     if there exists a source class field access then
16:         Add a parameter of the same type as the field.
17:     end if
18:     Modify invocations of any other extracted method, if necessary.
19:     Add the method in the new class.
20:     if the method is invoked (normal or super method invocation) by a class other
        than the source then
21:         Leave a delegate of the method in the source class.
22:     end if
23: end for
24: Add a reference of the new class in the source class.
25: Modify the accesses of any member of the new class in the source class.
26: Provide public accessors for the attributes of the source class and change the
    modifiers of the methods of the source class if necessary.
```

In the first steps, the algorithm removes the extracted entities from the source class (line 1) and adds the new class in the same package as the source class (line 2). Next, the bodies of the extracted methods and the types of the extracted fields are inspected and the algorithm adds the required import declarations in the new class (line 3). Next, the extracted fields are added in the new class as private attributes (line 5) with public accessors (line 6), in order to preserve the encapsulation principle. Before adding the extracted methods to the new class, a few steps must be taken (lines 8-23). First, if the extracted method assigns a variable or invokes a method of the source class, it is likely that it may change the state of source class instances. Therefore, the source class must be passed as a parameter to the new method when it is added to the newly extracted class, so that the same change can be feasible (line 10). On the other hand, if an attribute of the source class is only read, it suffices to add a parameter of the type of the accessed attribute in the new method (line 16); in this way, the method does not unnecessarily increase the coupling between the source and the extracted class. Because these changes may alter the method's signature, the algorithm has to modify the invocations of this method in the rest of the extracted methods (line 18). As the final step, the newly modified methods are added in the newly extracted class (line 19).

Having dealt with the changes in the new class, it's now time to change the source class as well. First, the algorithm checks if the extracted methods are also invoked by a third class (other than the source or the new class). If this is true, the original source method is turned into a method that delegates to the extracted one, so that its public interface does not change (line 21). Then, a field having the type of the newly created class is added in the source class (line 24), it is instantiated, and the accesses of any members of the new class are appropriately modified in the source class (line 25). For example, if a method's signature is changed, its invocations in the source class need to be modified as well. Finally, if a member of the source class needs to be accessed by the extracted class, public accessors (in case of attributes) might need to be added and modifiers might need to change (in case of methods) (line 26).

Revisiting the illustrative example

If this algorithm is applied for the example of Section 3.1.2 the following changes would happen:

1. The attributes *officeAreaCode* and *officeNumber* and the method *getTelephoneNumber()* are removed from class *Person*.

2. The class *TelephoneNumber* is created.

3. The extracted attributes are added to the new class and public accessors are created for them.

4. The method *getTelephoneNumber()* is added to the new class.

5. A parameter of type *Person* is added to the extracted method, because it accesses the attributes *name* and *job* from class *Person*.

6. The assignments of attributes *name* and *job* are replaced by setter invocations in the extracted method.

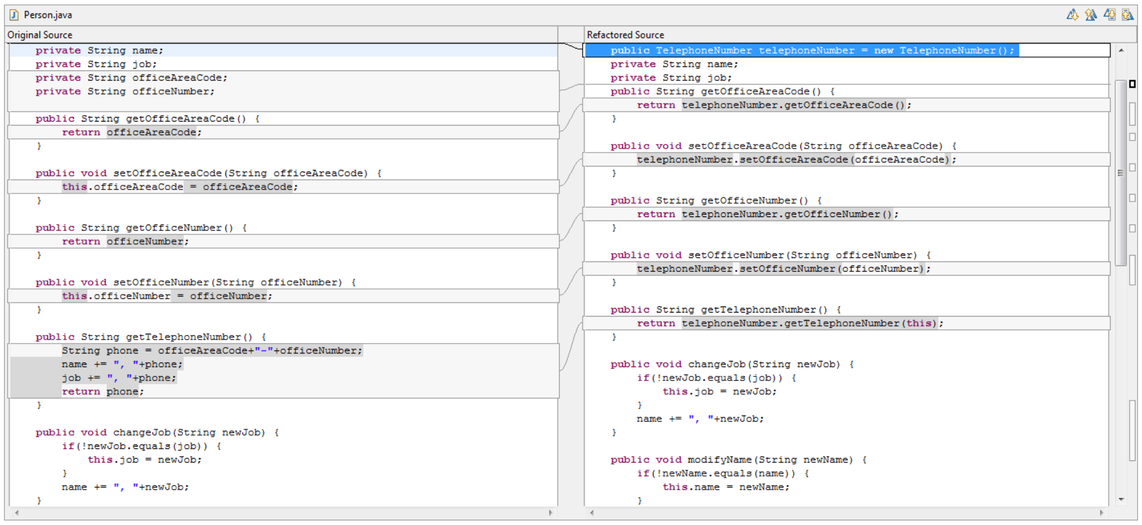
7. In the source class, a reference to the new class is added and it is initialized.

8. The extracted attribute accesses are changed appropriately (assignments are replaced with setter invocations and field accesses with getter invocations).

9. A delegate of the extracted method is left in the source class.

10. Public accessors are added in the source class for the attributes *name* and *job* because the extracted method accesses them.

Figure 3.4 shows part of the changes that happened in the source class via the Preview Wizard. Figure 3.5 shows the code for the extracted class.



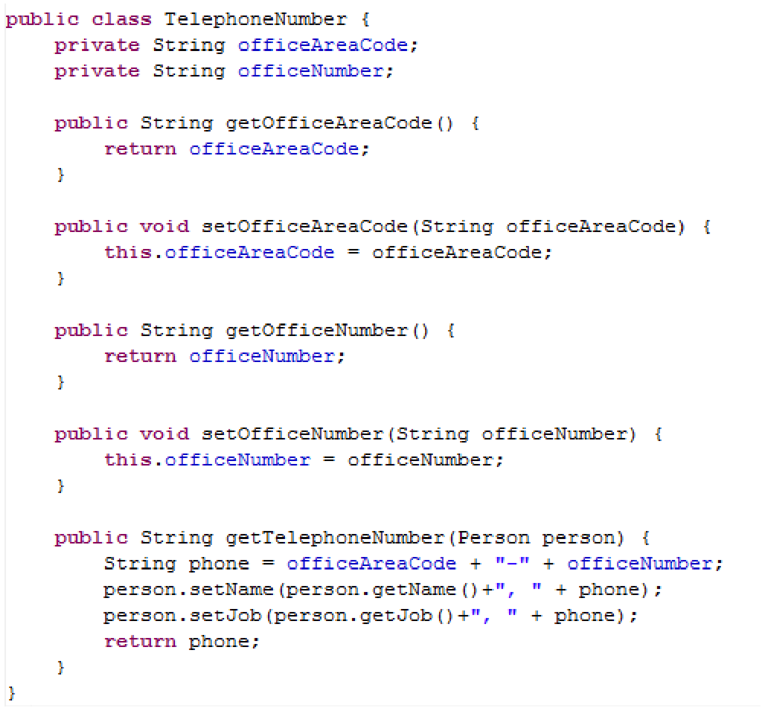
Original Source

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

    public String getOfficeAreaCode() {
        return officeAreaCode;
    }

    public void setOfficeAreaCode(String officeAreaCode) {
        this.officeAreaCode = officeAreaCode;
    }

    public String getOfficeNumber() {
        return officeNumber;
    }

    public void setOfficeNumber(String officeNumber) {
        this.officeNumber = officeNumber;
    }

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

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

Refactored Source

```
    public TelephoneNumber telephoneNumber = new TelephoneNumber();
    private String name;
    private String job;
    public String getOfficeAreaCode() {
        return telephoneNumber.getOfficeAreaCode();
    }

    public void setOfficeAreaCode(String officeAreaCode) {
        telephoneNumber.setOfficeAreaCode(officeAreaCode);
    }

    public String getOfficeNumber() {
        return telephoneNumber.getOfficeNumber();
    }

    public void setOfficeNumber(String officeNumber) {
        telephoneNumber.setOfficeNumber(officeNumber);
    }

    public String getTelephoneNumber() {
        return telephoneNumber.getTelephoneNumber(this);
    }

    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;
        }
```

Figure 3.4: Changes introduced in class *Person*.



```
public class TelephoneNumber {
    private String officeAreaCode;
    private String officeNumber;

    public String getOfficeAreaCode() {
        return officeAreaCode;
    }

    public void setOfficeAreaCode(String officeAreaCode) {
        this.officeAreaCode = officeAreaCode;
    }

    public String getOfficeNumber() {
        return officeNumber;
    }

    public void setOfficeNumber(String officeNumber) {
        this.officeNumber = officeNumber;
    }

    public String getTelephoneNumber(Person person) {
        String phone = officeAreaCode + "-" + officeNumber;
        person.setName(person.getName()+", " + phone);
        person.setJob(person.getJob()+", " + phone);
        return phone;
    }
}
```

Figure 3.5: The code of class *TelephoneNumber*.

Chapter 4

JDeodorant tool extension

In this section, we describe the details about the implementation of the methodology as an extension for the JDeodorant plugin. We present what toolkits and which features of the JDeodorant tool were used and how. Finally, we present a usage scenario where step-by-step instructions are given on how to use the tool and the state of the examined system (JHotDraw5.3) at each step is presented.

4.1 Implementation of the Eclipse plugin

The implementation of the tool was heavily based on the preexisting infrastructure of the JDeodorant plugin for the identification of the refactoring opportunitites and on the JDT and LTK libraries of Eclipse for the application of the refactorings.

4.1.1 Representation of Java elements in JDeodorant

JDeodorant uses a dual system to represent the Java elements in a project. In the first, elements are named as *XXObject* (*e.g.,* MethodObject) and in the second as *MyXX* (*e.g.,* MyMethod). The *Object* collection is a representation of the initial system and its purpose is not to allow any changes to the initial system in order to produce clean copies for all the virtual applications of the suggested refactorings. The *My* collection is the clean copy of the initial system and serves as a lightweight version of the *Object* equivalent. It holds only the necessary information for the dependency between the elements to construct the Entity sets. In this level, the system can change as the virtual application is performed. After that, the *My* copy

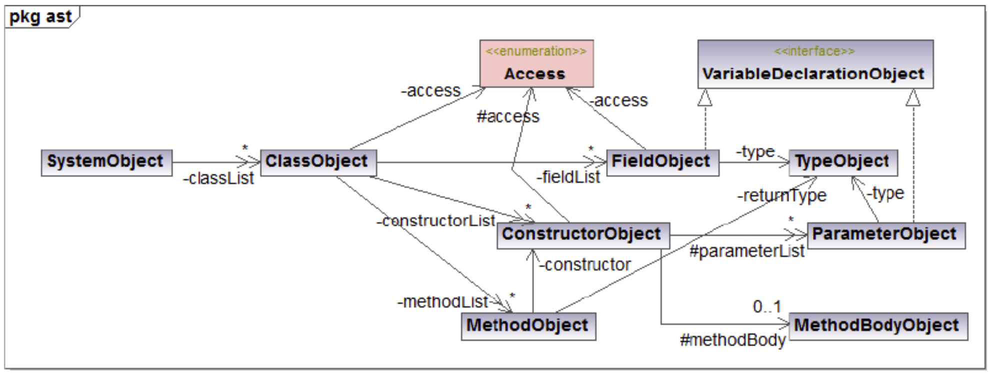


Figure 4.1: The JDeodorant *Object* representation of Java elements.

of the system is discarded and a new one is produced from the initial *Object* system. Figure 4.1 shows the relationships between the different *Object* classes. Each *My* class has a direct reference to the corresponding *Object* class.

4.1.2 Representation of Java elements in JDT

JDT uses two levels to represent Java elements: the Java model and the Abstract Syntax Tree (AST). The Java model is a high level representation of a Java project as shown in Figure 4.2. It merely contains information about the structure of the project and no information about the implementation details (*e.g.,* the bodies of the methods). For this reason, the AST is employed which decomposes the source code to the lowest level of detail. Through the AST, the source code can be manipulated.

An interesting feature of AST is the binding information. Each element has a binding based on its type, *e.g., ITypeBinding* for types, *IMethodBinding* for methods. Bindings contain a large amount of information for elements even if this information is not visible in the AST node. For example, a binding of a method invocation contains information about the method declaration, such as its visibility or the types of its arguments. The binding information is primarily used for matching purposes, for example to match a field suggested to be extracted with a field access in the code.

There is a direct mapping between the JDeodorant representations and the JDT representation as shown in Table 4.1. Although there is not a direct reference between the two representation, JDeodorant offers the *ASTInformation*, a class that

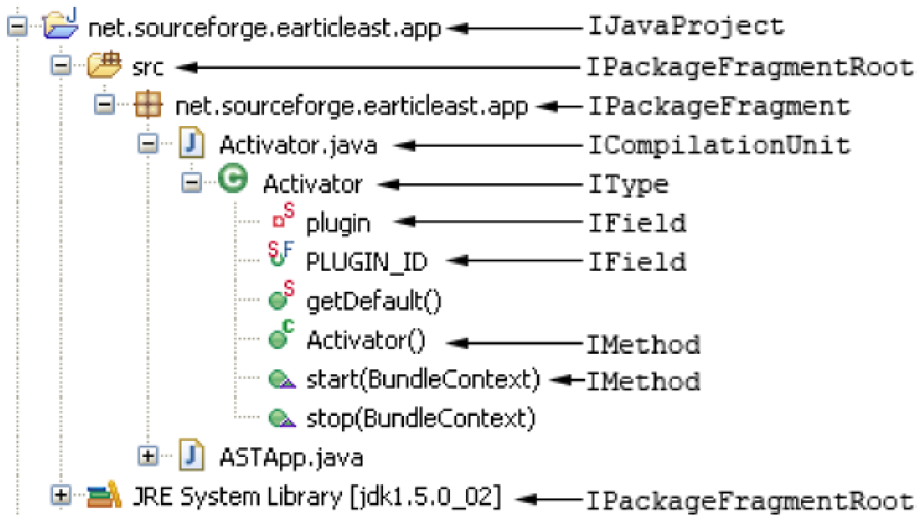


Figure 4.2: An example of Java model for a project.

serves as a mediator so the respective AST node can be retrieved from an *Object* instance. The ASTInformation is used to make the transition from the identification to the implementation.

Table 4.1: A mapping between the JDeodorant and the JDT representations.

Java element in JDeodorant	**Java element in JDT**
ClassObject	TypeDeclaration
MethodObject, ConstructorObject	MethodDeclaration
FieldObject	VariableDeclarationFragment
ParameterObject	SingleVariableDeclaration
AbstractStatement	Statement
AbstractExpression	Expression
LocalVariableDeclarationObject	VariableDeclaration
MethodInvocationObject	MethodInvocation
SuperMethodInvocationObject	SuperMethodInvocation
FieldInstructionObject	SimpleName
LocalVariableInstructionName	SimpleName
ClassInstanceCreationObject	ClassInstanceCreation
ArrayCreationObject	ArrayCreation

4.1.3 Tuning the performance in the identification process

The identification process is the most time consuming part of the tool. A few ways to improve the performance of this process are employed. The first has to do with the virtual application. Normally, when the virtual application is performed, the method recalculates the distances between all the entities in order to calculate the Entity Placement metric and evaluate the impact that the suggested refactoring would have if it was actually applied. However, in this process there is a big number of redundant recalculations. Since in the Extract class refactoring only the source class, the extracted class and their members would change, there is no reason to recalculate all the distances. Therefore, during the identification the entities (classes or members) that have changed are marked and only their distances from the rest of the system are calculated. This significantly improves the speed of the identification process.

When the developer presses the identification button a cycle is initiated: first, the selected project is parsed, then the distances of the initial system are calculated and finally, the identification is performed. When the developer performs a series of refactorings on the same project the parsing of the whole project is redundant. Thus, the tool stores the selection of the project. Moreover, after the application of a refactoring the classes changed are also stored in the memory. Therefore, when the developer selects the same project for identification only the changed class are parsed.

Another way not to improve the performance of the tool but facilitate the user speed up the process is the ability to select different levels of the project's structure to identify refactoring opportunities. The user can apply the identification process on a whole project, on a package or on a single class. Since the performance of the tool depends on the number of suggestions, the process is sped up if it is applied on fewer classes.

4.1.4 Implementation of the interface

To implement the Eclipse plugin we used the Plugin Development Environment (PDE). An item was added in the Bad Smells menu introduced by JDeodorant in [9]. The plugin consists mainly by a view with a table where the suggested refactorings are presented. The table is in a tree-table format, which means that each row is the root of a tree. The refactorings are grouped by the source class in which they were identified. The source class is the root of the tree and the refactoring suggestions are the children.

The rows of the table correspond to candidate refactorings. The columns of the table correspond to the source class and the resulting entity-placement value if the suggested refactoring is applied. The table is sorted by the entity-placement value in ascending order in two levels: in the subresults level and in the groups level. Each group is assigned the best entity-placement value of its subresults. The suggested refactorings that produce a worse value than the current system are discarded.

4.2 Usage Scenario in JHotDraw5.3

In this section we present a usage scenario of the tool on the JHotDraw system. At this point it is assumed that the user has already installed the plugin, launched Eclipse in the Java perspective and JHotDraw (version 5.3) has been imported in the workspace. Installation instructions and a video based on this scenario can be found at [1]. Table 4.2 shows the details of the scenario. Figure 4.3 shows the Extract Class component of the JDeodorant tool. The parts of the tool are annotated in the order they appear in the scenario.

The usage of the tool is quite straightforward. First of all, the “God Class” view needs to be activated by selecting it from the Bad Smells menu, which is in Eclipse’s menubar. Next, the module to be examined needs to be selected. The user can select either a project or a package or even a single class to inspect for extract-class opportunities. For particularly large projects, the last two options can be used to facilitate the identification process. To select a project simply click it on the Package explorer. To select a lower level module, expand the project tree and

Table 4.2: Usage Scenario

Steps	**Results**	**Comments**
1. Select the “Bad Smells”→“God Class” menu item.	The “God Class” view opens in the workspace.	
2. Select the “Package Explorer” → “JHotDraw5.3”project → “CH .ifa.draw.applet” package.	In the “God Class” view, the identification button is enabled.	
3. Click the identification button.	Three progress bars will appear: Parsing Java project, Calculating distances and Identification and virtual application of Extract class refactoring opportunities, in that order. Once the identification process has ended, the table is populated with suggestions and the “apply” button is enabled.	
4. Click the “expand” icon, to the left of the second row.	The row is expanded and the actual suggested refactorings appear underneath.	The second row contains refactorings suggested for the “DrawApplet” class.
5. Double-click on the first suggestion of the expanded row.	The file of the source class opens in the Java editor and the entities to be extracted are highlighted.	If followed, the first suggestion will extract the following entities: fSimpleUpdate, fUpdateButton, createButtons(JPanel), setSimpleDisplayUpdate() and setBufferedDisplayUpdate()
6. Select the “apply” button.	An input window pops up.	
7. Give the name “DisplayUpdate” in the input window.	No change is visible to the user at this point.	From the extracted entities it can be seen that the new class is responsible for the update of the display. Of course the choice of the name lies with the designer.
8. Select the “Preview” button from the input window.	The preview window appears.	
9. Check the changes in the preview and, if you agree with them, select “OK”.	The changes are performed and the new class file opens in the Java editor.	After the application of the refactoring the table is not updated. You will have to reselect the “identify” button.
10. To undo the refactoring, make sure the focus is on the package explorer and select Undo from the Edit menu.	The changes will be undone and the new class file will close from the editor.	

select the desired package or class. For this example, the whole JHotDraw project will be selected. JHotDraw is a system with 249 classes, 380 attributes (1.526 per class) and 2176 methods (8.739 per class).

After selecting the module to be examined, the identification button (with the "i" icon) is activated in the "God Class" view. Next, the user clicks the button to initiate the identification process and three progress bars appear (Table 4.2, step 3): one for parsing the project, one for calculating the Jaccard distances of the initial system and one for the identification process. In case the same module (project, package or class) is selected after the application of a refactoring, the tool will only parse the changed and introduced (new) classes.

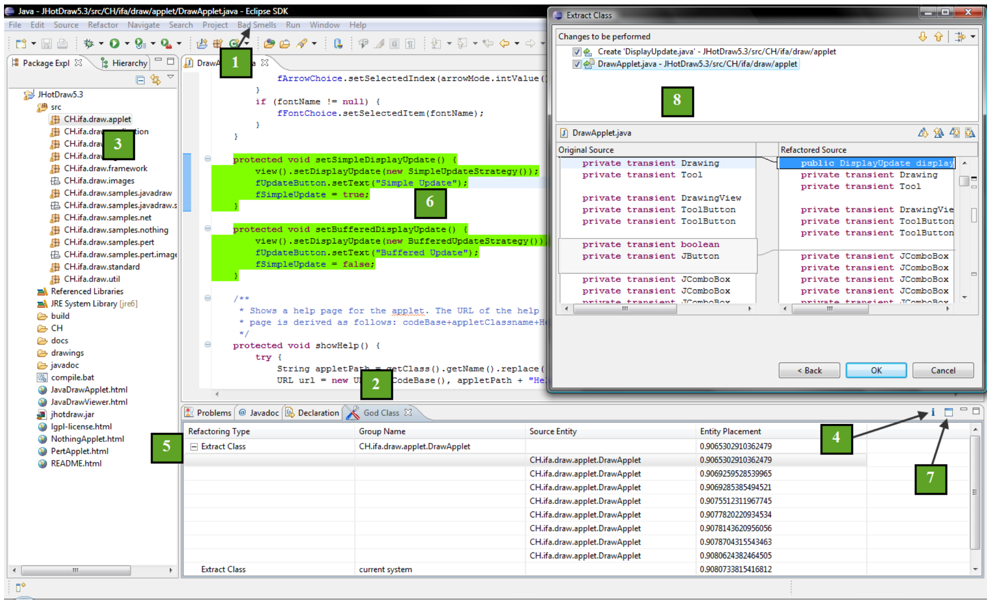
After the identification process is finished, the table of the view is populated and the results are ranked as described in Section 4.1.4. If the user wants to inspect one of the suggestions, the row of the table that corresponds to the source class will have to be expanded and then the user should double-click on the row that corresponds to the suggestion. The class members suggested to be extracted will be highlighted in the editor. The user can then navigate through the members by selecting the markers on the right side of the editor. At this point, no changes have been introduced to the system.

In this example, the table is populated with 35 source classes that are suggested to undergo extractions. The first suggestion is about the *UndoManager* class. If it is expanded, it can be seen that 8 extractions are suggested. If each one of them is selected and the user browses through the members suggested to be extracted, it becomes obvious that only two suggestions (1 and 2) are conceptually meaningful. These refactorings are also the best in terms of the Entity Placement metric, although this is not always the case. The class is about handling undo and redo activities. Thus, it is obvious that this class already specifies two different concepts. Refactoring 1 suggests to extract the redo functionality into a new class and refactoring 2 to extract the undo functionality. From the rest of the suggestions in this group, 4 of them (4,5,6 and 7) are subsets of the good suggestions, thus lacking some members that would complete the suggested concepts. 2 of the remaining suggestions (3 and 8) contain one more entity which would be better to be left in the source class. So, the first two suggestions identify new concepts and improve the design quality of the system. Here, the choice would be to extract the redo functionality, since it has a slightly better entity-placement value.

Having selected the refactoring to be applied, the application button is now activated. If it is clicked, an input page pops up prompting the user to give a name to the new class. The new class will be called “RedoActivityCollection”. Now, there are two possibilities: either apply the refactoring directly or preview the changes. If the “Preview” button is selected, the Preview Wizard will appear. From the top of the wizard, the user can select the files that changed. In the bottom of the wizard there is a split pane with the two versions of the class: on the left, there is the original version and on the right, the refactored version. The changes are marked and can be browsed using the markers on the right vertical ruler. To apply the refactoring, the user selects the OK button on the wizard. The refactoring is automatically applied and the new class opens in the Java editor of Eclipse.

After the changes, the examined system now has 250 classes, 381 attributes (1.524 per class) and 2182 methods (8.728 per class). As it can be seen the number of attributes increased because a reference to the new class was added in the source class. The same happened to the number of methods, since 6 methods had to be left as delegates. However, the average number of both decreased since we added a new class. If the identification button is clicked again, in the new results it can be seen that UndoManager, still exists as a set of suggestions. This is because the tool would still recommend the extractions about the undo functionality. However, now the suggestions are further down the ranking and there are better solutions to be performed.

In order to undo an Extract Class refactoring, the user just has to set the focus on the Package Explorer and select “Undo Extract Class” from the Edit menu. All the changes are reverted and the system returns to its previous state. If the identification button is selected again, the results will be the same as the first time.



```
            fArrowChoice.setSelectedIndex(arrowMode.intValue(
        }
        if (fontName != null) {
            fFontChoice.setSelectedItem(fontName);
        }
    }

    protected void setSimpleDisplayUpdate() {
        view().setDisplayUpdate(new SimpleUpdateStrategy());
        fUpdateButton.setText("Simple Update");
        fSimpleUpdate = true;
    }

    protected void setBufferedDisplayUpdate() {
        view().setDisplayUpdate(new BufferedUpdateStrategy());
        fUpdateButton.setText("Buffered Update");
        fSimpleUpdate = false;
    }

    /**
     * Shows a help page for the applet. The URL of the help
     * page is derived as follows: codeBase+appletClassname+He
     */
    protected void showHelp() {
        try {
            String appletPath = getClass().getName().replace(
            URL url = new U    CodeBase(), appletPath + "He
```

Create 'DisplayUpdate.java' - JHotDraw5.3/src/CH/ifa/draw/applet
DrawApplet.java - JHotDraw5.3/src/CH/ifa/draw/applet

```
Original Source                         Refactored Source
    private transient Drawing               public DisplayUpdate display
    private transient Tool                  private transient Drawing
                                            private transient Tool
    private transient DrawingView
    private transient ToolButton            private transient DrawingVie
    private transient ToolButton            private transient ToolButton
                                            private transient ToolButton
    private transient boolean
    private transient JButton               private transient JComboBox
                                            private transient JComboBox
    private transient JComboBox             private transient JComboBox
    private transient JComboBox             private transient JComboBox
```

Refactoring Type	Group Name	Source Entity	Entity Placement
⊟ Extract Class	CH.ifa.draw.applet.DrawApplet		0.9065302910362479
		CH.ifa.draw.applet.DrawApplet	0.9065302910362479
		CH.ifa.draw.applet.DrawApplet	0.9069259528539965
		CH.ifa.draw.applet.DrawApplet	0.9069285385494521
		CH.ifa.draw.applet.DrawApplet	0.9075512311967745
		CH.ifa.draw.applet.DrawApplet	0.9077820220934534
		CH.ifa.draw.applet.DrawApplet	0.9078143620956056
		CH.ifa.draw.applet.DrawApplet	0.9078704315543463
		CH.ifa.draw.applet.DrawApplet	0.9080624382464505
Extract Class	current system		0.9080733815416812

Figure 4.3: JDeodorant in action. 1) The “Bad Smells” menu. 2) The “God Class” view. 3) The Project Explorer. 4) The identification button 5) The expand icon and the tree-table format 6) The highlighted editor 7) The apply button 8) The preview wizard.

Chapter 5

Evaluation

In this chapter, we present the results of the evaluation on the performance of the identification methodology. More specifically, we were interested to see if the methodology can actually identify new concepts which were improperly defined in another class. Second, we wanted to know if the suggested refactorings would actually be applied by the developer if they were suggested by a tool. Moreover, we wanted to confirm that the suggested refactorings have a positive impact on the understandability of the code and the design quality of the system. Finally, we evaluated the ability of the Entity Placement metric to quantify the impact of the performed refactoring to the design quality of the system.

The designer and expert assessment address the first two issues. The third issue is covered by the expert assessment and the metrics comparison and, finally, the evaluation of the Entity Placement metric is covered by the comparison of its values with those of traditionally used cohesion and coupling metrics.

5.1 Designer Assessment

For the first part of the evaluation, the proposed methodology was applied to two projects and the results were discussed with the designers. For the first project, the results were discussed between the author, who was also one of the designers of the project, and another designer of this project. For the second project, the results were evaluated by the designer of the second project with the help of a developer with knowledge of the proposed tool. 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?**

5.1.1 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 the method. While the method was 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.

It was observed that for thresholds higher than 0.7 the methodology just kept increasing the size of the already proposed clusters thus making them less meaningful. Moreover, suggestions that would cause the entity-placement value of the system to deteriorate were not discussed. We asked one of the designers of the project to provide feedback on the suggested refactoring opportunities. The results are summarized in Table 5.1.

Table 5.1: 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 the 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 16 (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. The 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 6 suggestions on the ground that the proposed change would not have a significant impact on the design quality or the understandability of the code. 6 suggestions were rejected because the proposed class did not describe a separate concept or that such a change would unnecessarily increase the coupling of the system. Finally, in 7 cases 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 the 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 the method 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.

5.1.2 SelfPlanner

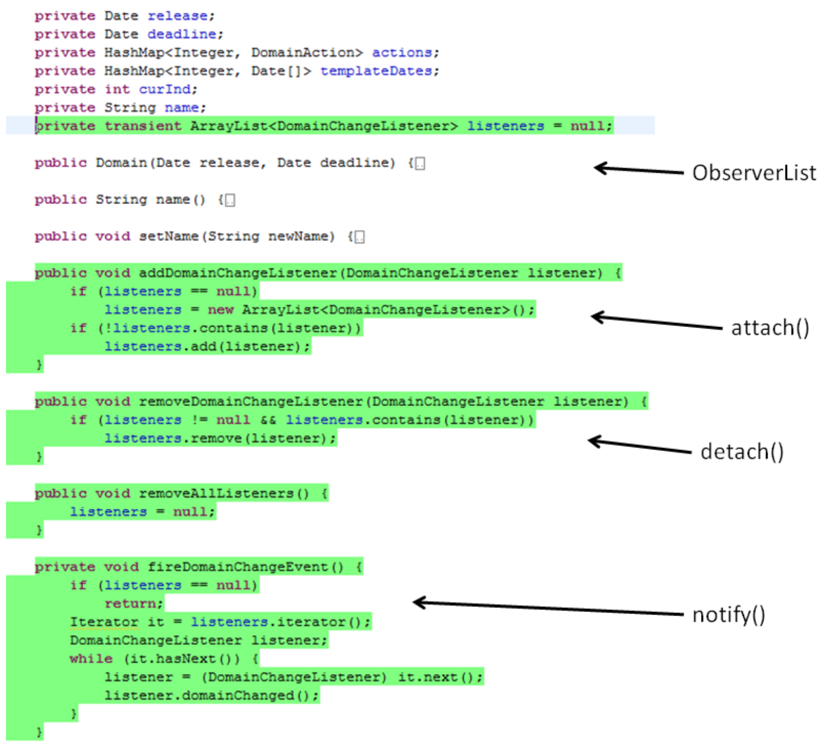
SelfPlanner [20] is an intelligent web-based calendar application that plans the tasks of a user using an adaptation of the Squeaky Wheel Optimization framework [13]. It is the outcome of a research project of the Artificial Intelligence Group at the department of Applied Informatics, University of Macedonia, 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 the 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 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 5.2.

Table 5.2: Results for SelfPlanner

Total Suggestions	Assigned Names	Applied
14	12	9

The 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



```
private Date release;
private Date deadline;
private HashMap<Integer, DomainAction> actions;
private HashMap<Integer, Date[]> templateDates;
private int curInd;
private String name;
private transient ArrayList<DomainChangeListener> listeners = null;

public Domain(Date release, Date deadline) {□

public String name() {□

public void setName(String newName) {□

public void addDomainChangeListener(DomainChangeListener listener) {
    if (listeners == null)
        listeners = new ArrayList<DomainChangeListener>();
    if (!listeners.contains(listener))
        listeners.add(listener);
}

public void removeDomainChangeListener(DomainChangeListener listener) {
    if (listeners != null && listeners.contains(listener))
        listeners.remove(listener);
}

public void removeAllListeners() {
    listeners = null;
}

private void fireDomainChangeEvent() {
    if (listeners == null)
        return;
    Iterator it = listeners.iterator();
    DomainChangeListener listener;
    while (it.hasNext()) {
        listener = (DomainChangeListener) it.next();
        listener.domainChanged();
    }
}
```

Figure 5.1: The *Subject* role of an Observer pattern as identified by the tool.

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 [11]. 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 [11]. This case is illustrated in Figure 5.1.

From the twelve conceptually meaningful Extract Class refactoring opportunities that were identified for SelfPlanner, the 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 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.

5.2 Expert Assessment

The second part of the evaluation was performed on the JHotDraw system (version 5.3). We applied 16 of the suggested refactorings and asked a professional in the business of software quality assessment, to give his expert opinion and answer three questions for each refactoring:

1. **Q1:** Does the extracted class describe a new entity?
2. **Q2:** Would you actually perform this refactoring, if a tool suggested it?
3. **Q3:** Does it improve the understandability of the code?

Table 5.3: Evaluation Results on JHotDraw5.3

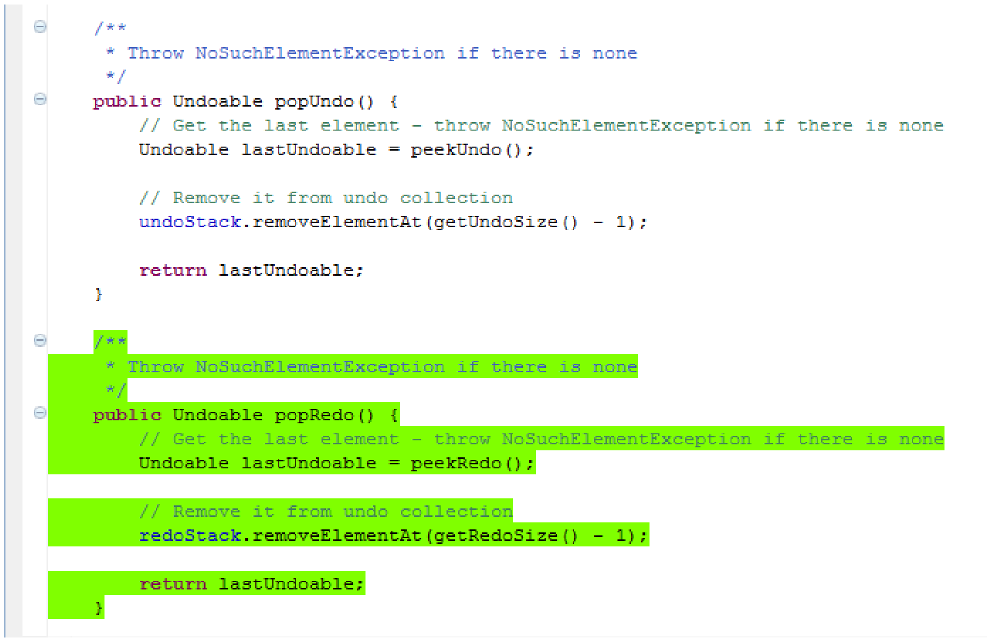
Total Refactorings	New Entity	Applied	Improved Understandability
16	12	9	9

Table 5.4 presents the expert's answers for the 16 refactorings applied on JHotDraw5.3 and Table 5.3 summarizes the expert assessment. In 12 of the total 16 cases (75%) (cases 1, 2, 5, 6, 7, 8, 9, 10, 11, 12, 13 and 14), the evaluator confirmed that the classes suggested to be extracted describe a separate concept or entity. An interesting point here was that the expert identified, on top of the 12 cases, two more classes that could be used as utility or helper classes (cases 15 and 16). Although, they do not actually describe a new concept they can still be extracted as new classes. In 9 of the 16 cases (56%) (cases 1, 2, 5, 7, 9, 11, 12, 14 and 15), the expert agreed that he would perform the refactoring if it was suggested by a tool. Interestingly, in 3 cases he claimed that he would not have been able to identify the refactoring opportunities manually. In 9 out of 16 cases (cases 2, 5, 7, 9, 11, 12, 13, 14 and 15), the expert notes that the performed refactorings have a positive impact on the understandability of the system.

During the evaluation, we discovered a few by-products of the method. In some cases, the code suggested to be extracted was duplicated in the same or other

Table 5.4: Expert's answers for refactorings applied on JHotDraw5.3

No.	Q1	Q2	Q3
1.	Yes	No due to two reasons: 1) I would not have seen the interconnection between these methods and 2) I would keep the implementation of the undo and redo functionality similar	Not really since two similar features are implemented in two different ways.
2.	Almost, the only strange thing I see is that there are still two buttons left in the DrawApplet-class. I would expect a button manager to handle all buttons	Yes (but then including the other buttons).	A bit, some non-essential methods are moved out. However, since the rest of the class is a bit chaotic this refactoring does not have a big impact
3.	No, it is basically a vector with an init method	No. The initialization of the vector is tightly coupled to the Nodefigure class so extracting this initialization from this class only creates confusion.	No, it has a negative impact.
4.	No, it would be if the 'SleeperThread' would also have been moved, but now it is just a class that provides initialization for a single field.	No, not without moving the sleeperThread as well. However, since the private methods that are extracted are not used I would probably just remove the complete code.	No impact on the understandability, this class is still to chaotic to be impacted by this chance.
5.	Yes	No, because I would not be able to see the connection between the fields and methods. But if a tool would offer this refactoring I would accept it.	A bit, although this class is also rather cluttered so this small refactoring does not really attract attention
6.	A bit, since the fields are never extracted from the class it is more a initializer-helper-class instead of an entity.	No, the extracted methods are not big, complicated or out-of-place in the old class.	No, it has a negative impact due to the fact that the extracted methods receive the same argument twice (this is based as second and third parameter).
7.	Yes	Yes, the result is a concrete class that groups related attributes and makes sure that the original class is less cluttered.	Yes, although again the impact is not that big
8.	A bit I think it is a Set (since items are not inserted twice).	No, the refactoring is too small to be useful and the same result can be achieved by using a standard construct.	No, it's confusing to see a construct used as a different construct.
9.	Yes, an 'OverlayContainer' (or 'ContainerOverlay')	Manually: no, this is because the impact is not that high. If a tool would suggest it I would probably perform the operation	A bit, although the impact is rather low due to the low number of statements in the classes.
10.	Similar to refactoring 8	Same as refactoring 8	Similar to refactoring 8
11.	Yes	Yes, it is a small and clear concept that can be re-used elsewhere	Yes, the naming is intuitive and there is less code to worry about
12.	Yes, it encapsulates some concept and hides away details about an (apparently ugly) hack	Probably, but only if I knew I could use re-use it somewhere else. Otherwise I don't think it would be worth the effort.	Yes, the details of a hack are somewhat hidden.
13.	A bit, it encapsulates two fields and their initialization behavior. Unfortunately, I cannot think of a good description for the extracted entity.	No, this refactoring moves several methods (addBackground, removeBackground) into the extracted class without providing access to these methods from within StandardDrawingView. This actually changes the API of this class, which is not desirable.	Yes, although again the impact is not that big due to the size of the retaining class.
14.	Yes, an ImageCache	Yes, it can encapsulates the cache-behavior in a straight-forward manner	Yes
15.	It is more of an utility class instead of an entity (since it does not contain any state)	Yes, it groups together related functionality	Yes, by extracting long initialization methods the original class becomes (a bit) cleaner
16.	I do not really see an entity in this new class, it looks to me like they are helper methods grouped together instead of a real entity.	Probably not, due to the fact that it creates circular dependencies between the two classes. This makes determining what happens in these methods rather hard.	No, it has a negative impact because of the introduction of circular dependencies.



```
/**
 * Throw NoSuchElementException if there is none
 */
public Undoable popUndo() {
    // Get the last element - throw NoSuchElementException if there is none
    Undoable lastUndoable = peekUndo();

    // Remove it from undo collection
    undoStack.removeElementAt(getUndoSize() - 1);

    return lastUndoable;
}

/**
 * Throw NoSuchElementException if there is none
 */
public Undoable popRedo() {
    // Get the last element - throw NoSuchElementException if there is none
    Undoable lastUndoable = peekRedo();

    // Remove it from undo collection
    redoStack.removeElementAt(getRedoSize() - 1);

    return lastUndoable;
}
```

Figure 5.2: The two methods, *popRedo* (suggested to be extracted) and *popUndo*, are almost identical.

classes. For example, the class *UndoManager* contains “undo” and “redo” functionality. The code for both activities is exactly the same. Figure 5.2 illustrates this example. The tool suggests two extractions: one for the undo and one for the redo activity. Another approach would be to extract one of them in a new class and replace both instances of the duplicated code with references to the extracted class. This type of duplication detection is outside the scope of this tool at this point, but we are considering it as a potential extension. The proposed changes can be manipulated further by the user to achieve better results. In another interesting case, the extracted code was totally disconnected from the rest of the class and was not used anywhere else, indicating possibly dead code that needs to be removed.

Furthermore, we examined a subcase of a specific class (DrawApplet) which seems to be a problematic one. This class has 42 methods and is used to create an applet. There were 4 suggestions only for this class and the expert’s opinion on them was: *“Overall, the four refactorings have helped in reducing the complexity and improving the readability of the DrawApplet-class. After these refactorings are applied, it is easier to start improving the code by (for example) also removing the circular dependencies on the DrawingViewHandle-class. Although the refactorings* *are not perfect, they are very valuable in starting up the refactoring of the complete class. The remaining work can now be done by a novice developer, something which was not possible with the original DrawApplet class. ".*

5.3 Metrics comparison for JHotDraw5.3

In this part of the evaluation, the impact of the 16 performed refactorings on JHotDraw5.3 was measured in terms of coupling and cohesion. The metrics that are used are popular in literature: Message Passing Coupling (MPC) [14] and Connectivity [3]. The values of these metrics were then compared to Entity Placement.

MPC for a class C is defined as the number of invocations of methods not implemented in class C by the methods of class C. Connectivity for a class C is defined as the number of method pairs of class C where one method invokes the other or both access a common attribute of class C, over the total number of method pairs of class C.

Figure 5.3 presents the progression of MPC (coupling), Connectivity (cohesion) and the Entity Placement metric. The first value on each chart represents the value of the respective metric on the initial system. To improve the design quality of a system, the goal is to reduce the coupling and the entity-placement value and increase the cohesion of the system.

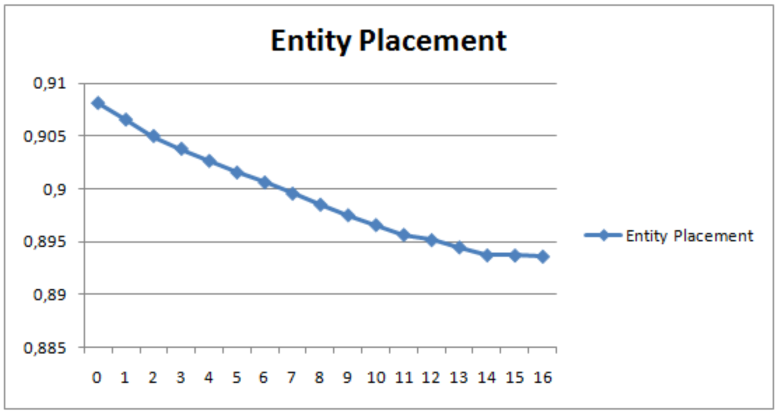
From the charts, it can be seen that all three metrics follow the expected trends, which means that the overall quality of the system was improved. However, in coupling a couple of unexpected values can be observed. As far as MPC is concerned, there are two possible situations. First, entities which still bear some dependencies with the rest of the source class might be suggested for extraction. In this case, the coupling is expected to increase (cases 1,2). Second, entities that are completely disconnected from the rest of the source class may be extracted. In this case, the coupling will stay the same, but because the number of classes will increase the average coupling of the system will decrease.

We compared the two metrics with the Entity Placement metric in terms of correlation. A strong negative correlation between Entity Placement and Connec-

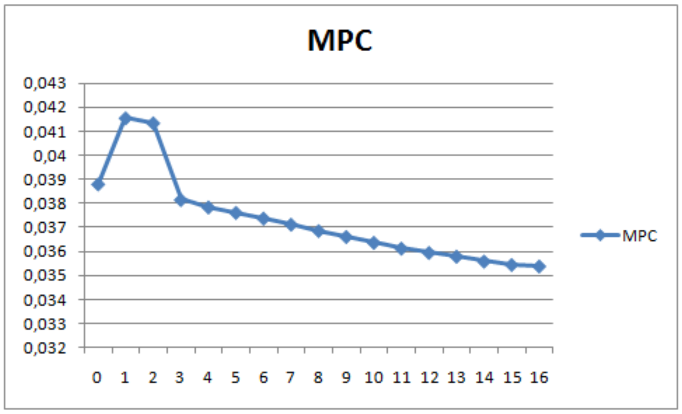
Table 5.5: Correlation between Entity Placement and MPC and Connectivity

EP-Conn	**EP-MPC**
-0.9935	0.8925

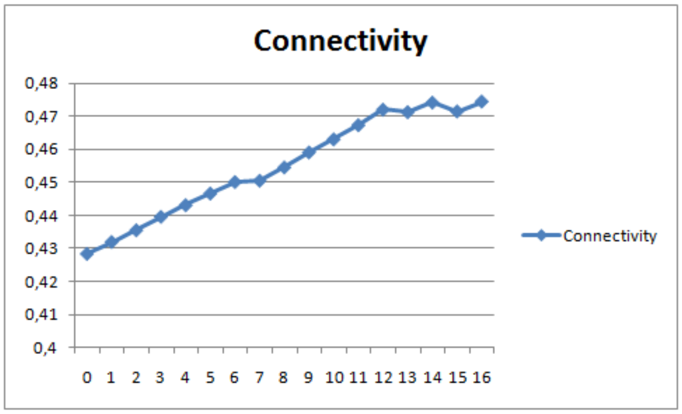
tivity and a strong positive correlation between Entity Placement and MPC were observed. Thus, it can be argued that Entity Placement is a sufficient metric to evaluate the effect of Extract Class refactorings.



(a) Entity-placement value progression.



(b) MPC value progression.



(c) Connectivity value progression.

Chapter 6

Conclusions

In this work, we proposed a novel approach to improve the design quality of an object-oriented system by applying Extract Class refactorings. To identify the refactoring opportunities, a hierarchical agglomerative clustering algorithm was used based on the Jaccard distance between class members, because of the ability of clustering algorithms to identify conceptually related groups of entities. The resulted suggestions are ranked according to the Entity Placement metric. The mechanics of the Extract Class refactoring were also described so that it preserves the system's behavior and its syntactical correctness.

We implemented the methodology as an extension for the JDeodorant Eclipse plugin. The user can identify and apply Extract Class refactorings in a simple way. The tool also shows to the user what are the extracted entities by highlighting them in the Java editor and what are the changes that will be performed using a Preview Wizard. The use of the tool comprises of simple steps and the interface is intuitive enough for the user to understand the refactoring. The input required is minimal which makes the tool suitable to be used by novice users as well. Techniques to improve the performance have been used so that the identification and refactoring process is performed much faster than if it was performed manually.

We evaluated the proposed methodology on various systems in terms of assessment by designers, expert assessment and metrics. The evaluation process showed that the methodology can produce meaningful and conceptually correct suggestions and extract new concepts. Furthermore, the expert confirmed that a good percentage of the proposed refactorings were good solutions that also improved the understandability of the code. Finally, it was proved that the suggested refactorings improve the design of the system in terms of coupling and cohesion and that Entity Placement is a good evaluator of the impact of the performed refactorings.

In the future, we would like to explore the possibility of combining the identification method with others. In some cases, as it was seen in the evaluation process, the suggestions could have been much better and more complete if the methodology was combined with others, like code duplication detection techniques. This would enable the method to identify identical or similar extractions even across classes and suggest a global solution to improve the design of more than one classes at once. Another possibility we would like to explore is the use of conceptual metrics in the methodology. These metrics take into account the names of class members. Because these metrics are sensitive to naming conventions, it would be better to use them in the ranking of the results and not in the identification of the refactoring opportunities, where the dependency information gives sufficiently good results.

We also plan to improve the interface of the tool with visualizations. It is believed that visualizations are a good tool to increase the awareness of the user. At the moment, the methodology is a black box to the user. We would like to change that by visualizing the classes as graphs where the distances and the members suggested to be extracted will be shown, so that the user can better understand why each extraction is suggested.

Bibliography

[1] JDeodorant. http://jdeodorant.com/.

[2] N. Anquetil and T. Lethbridge. Experiments with Clustering as a Software Remodularization Method. *6th Working Conference on Reverse Engineering*, 1999.

[3] L. C. Briand, J. W. Daly, and J. Wüster. A Unified Framework for Cohesion Measurement in Object-Oriented Systems. *Empirical Software Engineering*, 3(1):65–117, 1998.

[4] A. Chatzigeorgiou. Mathematical Assessment of Object-Oriented Design Quality. *IEEE Transactions on Software Engineering*, 29(11):1050–1053, November 2003.

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

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

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

[8] 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.

[9] 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.

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

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

[12] 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.

[13] D. E. Joslin and D. P. Clements. Squeaky Wheel Optimization. *Journal of Artificial Inteligence Research*, 10:375–397, 1999.

[14] W. Li and S. Henry. Object-Oriented Metrics that Predict Maintainability. *Journal of Systems and Software*, 23(2):111–122, 1993.

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

[16] 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. *6th International Workshop on Program Comprehension*, pages 45–52, 1998.

[17] O. Maqbool and H. A. Babri. The Weighted Combined Algorithm: A Linkage Algorithm for Software Clustering. *8th European Conference on Software Maintenance and Reengineering*, pages 15–24, March 2004.

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

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

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

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

[22] 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.

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

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

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

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

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

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

[29] A. van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. *21st Intl Conf. Software Engineering*, pages 246–255, 1999.

[30] T. A. Wiggerts. Using Clustering Algorithms in Legacy Systems Remodularization. *4th Working Conference on Reverse Engineering*, 1997.

[31] A. Zaidman and S. Demeyer. Automatic identification of key classes in a software system using webmining techniques. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(6):387–417, 2008.