The Journal of Systems and Software 85 (2012) 2241–2260

Contents lists available at SciVerse ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Identification and application of Extract Class refactorings in object-oriented systems

Marios Fokaefs[a,*], Nikolaos Tsantalis[a], Eleni Stroulia[a], Alexander Chatzigeorgiou[b]

[a] Department of Computing Science, University of Alberta, Edmonton, Canada
[b] Department of Applied Informatics, University of Macedonia, Thessaloniki, Greece

ARTICLE INFO

Article history:
Received 12 December 2010
Received in revised form 17 March 2012
Accepted 8 April 2012
Available online 25 April 2012

MSC:
10
10.040
10.060
10.070

Keywords:
Refactoring
Software reengineering
Object-oriented programming
Clustering

ABSTRACT

Refactoring is recognized as an essential practice in the context of evolutionary and agile software development. Recognizing the importance of the practice, modern IDEs provide some support for low-level refactorings. A notable exception in the list of supported refactorings is the “Extract Class” refactoring, which is conceived to simplify large, complex, unwieldy and less cohesive classes.

In this work, we describe a method and a tool, implemented as an Eclipse plugin, designed to fulfill exactly this need. Our method involves three steps: (a) recognition of Extract Class opportunities, (b) ranking of the identified opportunities in terms of the improvement each one is anticipated to bring about to the system design, and (c) fully automated application of the refactoring chosen by the developer. The first step relies on an agglomerative clustering algorithm, which identifies cohesive sets of class members within the system classes. The second step relies on the Entity Placement metric as a measure of design quality. Through a set of experiments we have shown that the tool is able to identify and extract new classes that developers recognize as “coherent concepts” and improve the design quality of the underlying system.

1. Introduction

Evolutionary software development is the most broadly adopted lifecycle process today. Software evolves throughout its lifecycle, even past its release, and, as a result, the as-is design of the system usually ends up deviating from its original rationale and violating design principles. Such violations manifest themselves as “bad smells” (Fowler et al., 1999) and refactoring becomes necessary to eliminate them (Opdyke, 1992). Refactoring is the process of introducing behavior preserving restructurings to the code, in order to improve its design and enable it to support further development.

This work is motivated by a specific bad smell called “God Class” (Fowler et al., 1999). In principle, a class should implement only one concept (Martin, 2003) and should only change when the concept it encapsulates evolves. The violation of this principle results in large, complex, unwieldy, inelegant, less cohesive and difficult to understand and maintain “God Classes”. Generally, there are two types of such classes: some hold a lot of the system’s data in terms of number of attributes (“Data God Classes” or “Lazy Classes”) and others implement a great portion of the system’s functionality in terms of many and frequently complex methods (“Behavioral God Classes”). In the first case, developers can redistribute the attributes of the “God Class” or move functionality (i.e. methods) from other classes closer to the data. In the second case, they can either move functionality from the “God Class” closer to the data of other classes or simplify the class by extracting a cohesive and independent piece of functionality (Fowler et al., 1999; Demeyer et al., 2002). The latter is a refactoring called “Extract Class”.

Our work is not trying to identify “God Classes”, but rather Extract Class refactoring opportunities in order to decompose large classes. The proposed method recognizes coherent packages of data and behaviors which if extracted into a new class would result in improving the overall system design and, at the request of the developer, automatically applies the “Extract Class” refactoring. To identify Extract Class opportunities, our method employs a clustering algorithm, as clustering has long been used for software remodularization (Tzerpos and Holt, 1998; Wiggerts, 1997). More specifically, the intuition behind using clustering in this problem is that clusters may represent cohesive groups of class members (methods and attributes) that have a distinct functionality and

* Corresponding author. Tel.: +1 780 886 6893.
E-mail addresses: fokaefs@ualberta.ca (M. Fokaefs), tsantalis@ualberta.ca (N. Tsantalis), stroulia@ualberta.ca (E. Stroulia), achat@uom.gr (A. Chatzigeorgiou).

0164-1212/$ – see front matter © 2012 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jss.2012.04.013

can be extracted as separate classes (Tan et al., 2005). In order to apply a clustering algorithm, we need to define a distance metric, which for our methodology is based on structural dependencies (i.e. field accesses and method invocations) between class members.

Semantic metrics have also been used to identify conceptually correlated software components (Maletic and Marcus, 2001; Marcus and Poshyvanyk, 2005; De Lucia et al., 2008). However, in order for these metrics to be reliable in the identification process, specific code conventions must be followed by the project developers. For instance, the developers must choose appropriate names (i.e. naming conventions) for variables and methods that reflect the concepts and the functionality that they represent. Moreover, good documentation practices (in-code and documentation comments) must be followed namely English language must be used in order to match the code language, spelling errors should be avoided since they can break the matching between comments and code elements. Furthermore, due to the dynamic nature of software projects (e.g. changes in the development processes and/or team members), these conventions may not be consistent throughout the project's lifecycle. In the evaluation of our work, we did investigate the accuracy of our approach using structural, semantic and a combination of structural and semantic measures as the distance metric for the clustering algorithm and discuss our findings.

To assess the potential design improvement that a candidate class extraction will bring about to the system, our method uses the *Entity Placement* metric (Tsantalis and Chatzigeorgiou, 2009), a ratio of the overall system cohesion over its coupling. Finally, the manual application of the Extract Class refactoring is not trivial, because of the difficulties arising from the human-driven analysis of both inter- and intra-dependencies of the extracted class members. For this reason, we have developed tooling within the Eclipse JDeodorant plugin (Fokaefs et al., 2011) to automatically apply such refactorings once the developer has agreed to do so.

This paper makes three novel contributions to the state of the art in supporting object-oriented design evolution.

1. *Identification of new concepts.* In the context of this work, we define a concept as a distinct entity or abstraction for which a single class provides a description and/or a set of attributes and methods that contribute together to the same task. Our method uses a clustering algorithm to identify *conceptually* related groups of entities (i.e. attributes and methods) within a single "God Class". The identified concepts (i.e. entity clusters) are considered as candidates for extraction.
2. *Ranking of the candidate refactorings* based on their anticipated impact on the design quality, as measured by the *Entity Placement* metric (Tsantalis and Chatzigeorgiou, 2009), a combined metric that captures both coupling and cohesion.
3. *Automatic application of a selected refactoring*, so that it preserves the syntactic correctness of the system and its observable behavior. The refactoring application process checks a list of preconditions before proceeding with the actual refactoring and ensures that all the appropriate transformations are applied in both the original and the new class.

The rest of the paper is organized as follows. Section 2 presents a review of the related literature. Section 3 describes the identification of refactoring opportunities and the mechanics of the application of the suggested refactorings in detail. Section 4 presents the results of the evaluation process. Section 5 concludes this work, summarizing its main points and the results of the evaluation process as well as discussing our future plans.

2. Related work

This section reviews the related literature in three different areas. The first group of related papers focuses on general methods about code smell detection. The second group focuses on research around methods for software remodularization, architecture recovery or migration of legacy systems, including several relying on clustering. The third group focuses on the identification of problematic classes with low cohesion or key classes that have a big portion of the system's functionality and are intensively maintained. Finally, the third group involves earlier research on identifying Extract Class opportunities and suggesting extraction solutions.

2.1. Code smell detection methods

Moha et al. (2010) introduced DECOR, a method for the specification and detection of code and design smells and DETEX, an instantiation of this method. First, a taxonomy and classification of smells is defined based on the key concepts in order to highlight the similarities and differences among smells. The specification of smells is performed using a domain-specific language (DSL) in the form of rules using the previous taxonomy. The rules describe the properties that a class must have to be considered a smell. The DSL allows the definition of properties for the detection of smells, including structural properties, naming properties and internal properties using metrics. The detection algorithms are automatically generated by parsing the rules defined in the specification process and they are applied on a model representation of the examined system produced by forward engineering or reverse engineering its source code.

Marinescu (2004) proposed the concept of detection strategies as a means to detect instances of a structural anomaly. A detection strategy is actually a composition of various metric rules (i.e. metrics that should comply with proper threshold values) combined with AND/OR operators into a single rule that expresses a design heuristic. The threshold values used in the metric rules were defined based on statistical data collected from more than 60 Java and 50 C++projects. The identified design problems can be eliminated based on corresponding restructuring strategies which informally describe (i.e. in textual form) the required actions that should be taken for the elimination procedure.

Munro (2005) attempted to address the issue of identifying the characteristics of a bad smell through the use of a set of software metrics. By using a predefined set of metric interpretation rules the software engineer can be provided with significant guidance for locating existing bad smells. Munro uses exactly the same detection approach as Marinescu (2004), i.e. a composition of metrics that should comply with proper threshold values combined with AND/OR operators into a single rule. His approach was evaluated on two case studies, a small sized hotel booking system and a medium sized Graph Tool system written in Java and for two code smells, namely Lazy Class and Temporary Field.

Van Emden and Moonen (2002) proposed an approach for the automatic detection and visualization of instanceof and typecast code smells. The "instance" of code smell appears as a sequence of conditional statements that test an object for its type, while the typecast code smell appears when an object is explicitly converted from one class type into another. To this end, they developed a prototype code smell browser, named jCOSMO, which visualizes the detected code smells in the form of a graph. In this graph, the code smells are represented as additional nodes connected to the code entities that they appear in. In this way it is possible to discriminate which parts of the system have the largest number of code smells and would benefit the most from restructuring.

2.2. Software remodularization

The problem of software remodularization has been discussed in the context of all types of software systems. Mancoridis et al. (1998) proposed a method for remodularizing a software system, in terms of "good" clusters with high cohesion (within the clusters) and low coupling (between the clusters). The method produces the *Module Dependency Graph* based on the source code and then applies clustering on the resulting graph, using the *Modularization Quality* measure to evaluate the produced clusters. This measure favors intra-module connectivity and penalizes inter-connectivity. After finding a suboptimal partition using a combination of hill climbing and genetic algorithms, the method builds a hierarchy of clusters using a hierarchical clustering algorithm.

Doval et al. (1999) consider the problem of identifying a good partitioning as an optimization problem. They propose a genetic algorithm as a means of partitioning large software systems using as an objective function the modularization quality measures (Mancoridis et al., 1998). In a similar work, Shokoufandeh et al. (2005) apply a spectral clustering algorithm in order to remodularize a system, adopting the same measure as the objective function.

Sartipi and Kontogiannis (2001) propose a semisupervised clustering framework for software architecture recovery. The process starts by analyzing the source code to compute component similarity. 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. Using these metrics, sufficiently similar components are clustered together and, finally, the user manually assigns the remaining components to clusters or reallocates the modules among the clusters. 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 (1999) use clustering and concept analysis for the purpose of migrating legacy systems to object-oriented technologies. They identify two shortcomings of clustering: an element can exist only in one cluster, and an element may arbitrarily be assigned to different clusters in different runs of the clustering algorithm. Neither of these problems are relevant to the task of extracting classes. In this context, elements "should" be in only one cluster because attributes and methods should be declared only in one class, and if an element is equally close to two clusters these clusters will eventually be merged, resulting into a more complex class to be extracted. Another difference between cluster and concept analysis is that the latter method can identify all possible partitions. Up to a degree this problem is overcome in our work by presenting all the clusters identified by the algorithm and merging the results accordingly.

The aforementioned methods focus on remodularization of software, around larger modules, like, for example, packages, while our method focuses on software remodularization at the class level. The criteria appropriate for these two types of remodularization are different. Classes may be organized into packages according to their release plan (classes in a single package should evolve and be released together), or according to the inheritance hierarchy in which they belong (a package may contain the hierarchy tree of a single class). 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.

2.3. Identification of "God Classes"

There is substantial earlier research around the problem of identifying "God Classes", or problematic complex modules more generally.

Trifu and Marinescu (2005) 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 belongs in that category or not. The fundamental shortcoming of this method is that it requires thresholds for the considered metrics, which have to be empirically or statistically determined for any given system.

Tahvildari and Kontogiannis (2003) 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 user-defined input.

DuBois et al. (2004) propose a set of "guidelines" based on conceptual and macroscopic criteria for improving the system design. Their guideline for simplifying "God Classes" advises to "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. (2002) suggest some conceptual criteria for identifying "God Classes". They are usually incohesive and memory consuming classes. They usually have abstract names like "Controller", "Manager", "Driver" or "System". Any change to the system may cause changes to these classes. They are often called the "heart of the system" and, in most of the cases, they are hard to maintain.

All of the above methods are likely able to identify problematic classes and improve system design metrics, but they do not produce specific design improvement suggestions which are meaningful to the designer.

Chatzigeorgiou (2003) and Chatzigeorgiou et al. (2004) apply the Hyperlink Induced Topic Search (HITS) algorithm in order to evaluate the quality of object-oriented design models. The algorithm is extended in order to account for the number of discrete messages exchanged between classes. The principal eigenvectors of matrices, derived from the adjacency matrix containing the number of exchanges messages between classes, are used to identify and quantify "God Classes" that deviate from the principle of distributed responsibilities. Furthermore, the non-principal eigenvectors are also employed in order to identify dense communities of classes in a system that are well-separated from one another and possibly constitute reusable components.

Xanthos (2006) employ a technique from algebraic graph theory known as spectral graph partitioning. In this approach an object-oriented system is represented as a graph where nodes stand for the classes and the edges stand for the discrete messages exchanged between the classes. The resulting graph is recursively bi-partitioned until one of the produced subgraphs is less cohesive than its parent graph. This is determined by examining if the number of internal edges of each subgraph (i.e. intra-exchanged messages) exceeds the number of external edges (inter-exchanged messages). If the external edges are more than the internal the algorithm stops.

Zaidman and Demeyer (2008) propose a method that employs HITS to identify key classes in object oriented systems based on coupling. This algorithm has the ability to incorporate the indirect coupling between classes in the calculation of the overall system 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.

Khomh et al. (2009) propose an approach based on Bayesian Belief Networks (BBNs) to specify design smells and detect them in programs. Within the context of design smell detection, a BBN is a directed acyclic graph, where nodes correspond to either an input (e.g. a metric value for a given class) if there are no incoming edges, to a decision step if there are incoming edges (e.g. is a class part of a smell given the values of its parent nodes?), or to an output node if there are no outgoing edges. A directed edge between two nodes indicates a probabilistic dependency between the starting and the ending nodes. Eventually, the output of a BBN is a probability that a class is part of a design smell. In this way, it is possible to sort the candidate classes for a given design smell and prioritize the inspection of classes with higher probability. Their approach is evaluated for the detection of the Blob antipattern by building a BBN model on two open-source projects, namely Xerces and GanttProject. The estimation of precision and recall is performed by comparing the results of the model with manually located smells. An apparent disadvantage of probabilistic models, like Bayesian networks, is that the required probabilities result based on a training set, which within the context of design smell detection corresponds to metric values for classes which have been already determined as valid instances. Obviously, the training set affects the classification results of the model on the actual data set.

Vaucher et al. (2009) perform an exploratory analysis of the "life cycle" of God Classes on two open-source projects, namely Xerces and Eclipse JDT. They use the same Bayesian approach as Khomh et al. (2009) for detecting the presence of God Classes in systems and ranking them. Furthermore, they study the evolution of the detected God Classes in the examined systems. More specifically, they study the way that God Classes are introduced in and removed from the systems and how they evolve over time. Finally, they have built a prediction model on Xerces project that predicts the likelihood of creating a God Class given a specific code change. The prediction model can be used to prevent the introduction of God Classes in future versions of the systems.

As far as these works are concerned, we should point out that the goal of our method is not to distinguish between "God Classes" and other classes. Every class of the system is equally inspected to assess whether it might potentially benefit from an extraction of some of its members. If a suggested extraction improves the overall quality of the system in terms of its cohesion and coupling, as combined in the Entity Placement metric, it is presented as a refactoring opportunity to the developer. This way, a good opportunity can be identified even in a class, which may not seem problematic by means of traditional cohesion metrics.

2.4. Identification of Extract Class opportunities

Simon et al. (2001) propose visualization based techniques for identifying 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 fundamental shortcoming of visualization based approaches is that there is no good spatial metaphor for laying out the classes and they do not scale up. In our work, new candidate classes are identified as cohesive clusters of entities and they are ranked according to their anticipated impact on the design of the whole system.

Joshi and Joshi (2009) 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 pointed out by the authors, is that, for large systems, the lattices can become very complex and thus it is more difficult for the designer to visually inspect the lattice and identify problematic cases. Moreover, while this method focuses on improving the cohesion of a class, it neglects to consider the conceptual coherence of the suggested extracted classes, which can only contain methods. Finally, this method does not guarantee that the suggested refactorings will not affect the behavior of the program.

De Lucia et al. (2008) propose a methodology that takes into account both structural and conceptual criteria. Their method builds a weighted graph of the class methods based on structural and semantic cohesion metrics, which then is split using a Max-flow Min-cut algorithm to produce more cohesive classes. The semantic cohesion metric is based on the names of classes and entities, which, in poor designs, can be arbitrary and thus the results highly depend on the naming conventions used by the developers of a project. Furthermore, by bipartitioning the graph it is possible to miss potential clusters. For example, a class might consist of more than two cohesive subclasses which could not be identified by splitting the class. Moreover, 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. Finally, the weights of the different metrics are statistically determined based on the specific characteristics of the examined system and there no systematic way of defining default thresholds is provided. The work by Bavota et al. (2011) provided an extended experimental evaluation where the findings of the approach on an open-source project were evaluated by graduate students.

In a similar work, Bavota et al. (2010) propose a simple decomposition technique to identify Extract Class refactoring opportunities. They use the same set of metrics as in the work by Bavota et al. (2011) to calculate the cohesion between methods. Then based on the calls between methods they find chains of methods by calculating the transitive closure of their dependencies. The chains that are above a minimum cohesion and are of a minimum length are suggested as possible extractions. The use of chains allows the technique to possibly identify more than two extractions. This work suffers from some of the limitations found in its predecessors, such as the need for threshold definition and the exclusion of attributes from the partitioning process which can lead to non-optimal solutions.

Bavota et al. (2010) propose a game-theory approach for identifying Extract Class opportunities. In this approach, the two candidate classes, in which a candidate God Class may be decomposed, "compete" against each other over the methods of the original source class. At each round, each "player" is trying to obtain a method that will increase its cohesion and not increase its coupling based on structural and semantic similarity measures. This method suffers from several shortcomings. First, it assumes that the source class should be divided in two new classes, where our method allows for the decomposition of a God Class in any number of smaller classes. Furthermore, the game is defined as a 2-player general-sum game and it is known that finding particular Nash equilibria in general-sum games is a hard problem (PPAD-complete) (Daskalakis et al., 2009). And although it is proven that there exists a Nash equilibrium for every game, this equilibrium is not guaranteed to be a pure Nash equilibrium. In the Extract Class problem the two players cannot mix over their actions because they have to take a single method at each turn. Thus, it is not guaranteed that at each iteration there will be a pair of actions for the two players to select.

In our previous work (Fokaefs et al., 2009), we have already presented the identification of Extract Class opportunities using a clustering algorithm. The differences compared to our previous work are:

- improvement in the application of the hierarchical agglomerative algorithm so that it does not require user-defined inputs;
- additional preconditions;

- detailed mechanics for the application of the Extract Class refactoring;
- investigation of the suitability of Entity Placement as a ranking criterion; and
- enhancement of our evaluation process.

More specifically, with respect to the enhancement of the evaluation process we included two additional types of experiments. In the first type, the evaluators were asked to manually identify concepts without neither having knowledge of our methodology nor having the assistance of our tool. Next, we compared their findings with the tool's suggestions in order to extract the precision and recall of our approach. In the second type, we provided the evaluator, who is a professional in software quality assessment, with a set of already applied refactorings on a well-known open-source system (JHotDraw) and asked him to provide his expert opinion on whether the newly created classes constituted meaningful and valid concepts and if the applied refactorings improved the understandability of the code.

More generally, there are two differences between this body of work and our method. First, they stop at the identification of the problems and do not suggest specific solutions, whereas our method offers a complete solution to the problem, from identifying the opportunities to suggesting proper refactorings and finally applying these refactorings in an automatic manner. Second, they attempt to identify a single "optimal" solution, which the designer should accept or reject in its entirety. On the contrary, our method is essentially a stepwise approach, that extracts a set of ordered refactoring suggestions. This offers the advantage of gradual change of a system, allowing the developer to assess the conceptual integrity of the refactoring suggestions at each step.

3. Methodology

Our Extract Class refactoring method consists of three steps: (a) the identification of the refactoring opportunities, (b) their ranking based on the improvement they are anticipated to bring about to the system, and (c) the actual automated application of the refactoring chosen by the developer.

3.1. Identification of Extract Class opportunities

The process of identifying Extract Class opportunities consists of two steps. First, each class is analyzed in order to extract dependency information among the class members, so that we can calculate distances between them. This information is then used by a clustering algorithm, which identifies cohesive groups of entities that can be extracted as separate classes. Second, the classes identified as candidates to be extracted are filtered by applying a set of rules that evaluate whether these classes have sufficient functionality and whether the suggested refactorings would preserve the behavior of the original program.

Note that 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. Clustering algorithm and distance metric

Our objective in designing a clustering algorithm for the identification of candidate classes for extraction was to require minimum input by the developers and minimum a priori knowledge about the system under examination.

We first considered a partitioning algorithm, like k-means. This family of algorithms requires as input the number of desired clusters and assumes that each object is placed in a feature space, where the space dimensions correspond to the object attributes. None of these knowledge assumptions are realistic in our case. First, it is impossible to know how many, if any at all, concepts might be intertwined within a single class implementation. And if one were to run the algorithm for every single possible value of k, the performance of the process would dramatically deteriorate. Second, it is unclear in terms of what attributes one might describe the class members in order to place them within a multi-dimensional space. 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), because not everything is connected to everything, a fact that corresponds to a large amount of noise. By nature, partitioning algorithms will cluster all entities in spite of how far some of them might be. Furthermore, these algorithms require an initial partitioning and then the clustering is updated until it reaches an optimal level of fitness. However, the resulting clustering is highly affected by this initial configuration. First, different initial partitions might produce different clusterings. Second, because of the initial partitioning the algorithm might fall in a local minimum and never converge to the preset optimal fitness level. For this reason, another stop condition is needed which is the number of iterations. However, it is not easy to define this number because it depends on many factors including the nature of the examined dataset.

Next, we considered a density-based algorithm, 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 also require a priori knowledge of several parameters. A density based algorithm, like DBSCAN (Ester et al., 1996), needs two parameters: (a) ε-neighborhood, which defines a radius around a point, within which a dense subgroup (not a cluster) can be defined, and (b) *MinPts*, which corresponds to the minimum number of points for each subgroup. In this particular software remodularization problem, it is not clear what the minimum number of entities in a neighborhood may represent and thus, it is difficult to define one. Furthermore, it is not easy to define an ε value and if we try different values, we will have to run the clustering algorithm for each of these values.

Finally, we considered graph partitioning algorithms, which can be either spectral methods such as Algebraic Connectivity (Holzrichter and Oliveira, 1999) and Principal Component Analysis (Jolliffe, 1986) or flow-based such as the Max-flow Min-cut (Cormen et al., 2001) algorithms. In these algorithms, we assume the existence of a weighted graph on which we perform recursive bipartitioning in order to identify the corresponding clusters. The problem with recursion is that we need to define a stop criterion based on an objective function. This transforms the problem into an optimization one, as it is being addressed by Shokoufandeh et al. (2005). In our case, we do not want to view the problem as an optimization one, since our method aims to suggest a set of possible refactorings rather than a single optimal one.

Eventually, we decided to adopt a hierarchical agglomerative algorithm. This algorithm starts by assigning each class member 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, which forms the root of a hierarchy of clusters. The actual clusters can be determined at the merging points. The hierarchy of the clusters is usually represented by a dendrogram (an example is shown in Fig. 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 in which two clusters were merged.

From the examined algorithms, the hierarchical agglomerative algorithm is the only one that satisfies all of the following criteria:

1. *It is deterministic* as it requires no random initializations (unlike *k*-means) and it always produces the same results.
2. *It is finite* as it does not require user-defined stop conditions (unlike recursive graph bi-partitioning and *k*-means) and it produces a final output in finite time.
3. *It is fully automatic* as it does not require user-defined input (unlike *k*-means and DBSCAN).

An important aspect of the hierarchical agglomerative clustering algorithm is the *distance merging criterion* according to which the algorithm selects which clusters to merge. There are several methods for determining the two closest clusters, such as (a) the maximum distance between the members of two clusters (*complete linkage*), (b) the average distance (*average linkage*), or (c) the minimum distance (*single linkage*). According to Anquetil and Lethbridge (1999) complete linkage favors more cohesive clusters; single linkage favors 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. In contrast with cohesion, coupling is an uncontrolled variable. Thus, we chose to adopt the single linkage method, in order to lessen the coupling between the newly created class and the original one.

The *distance metric* used by our algorithm is the Jaccard distance, which according to Anquetil and Lethbridge (1999) produces good results in software remodularization. To define the Jaccard distance between two class members the notion of *entity sets* is employed, borrowed by Tsantalis and Chatzigeorgiou (2009).

The entity set of an attribute *a* contains:

- the attribute *a* itself;
- the methods directly accessing *a* that belong to the same class with *a*;
- the methods accessing *a* through public accessors (getter and setter methods).

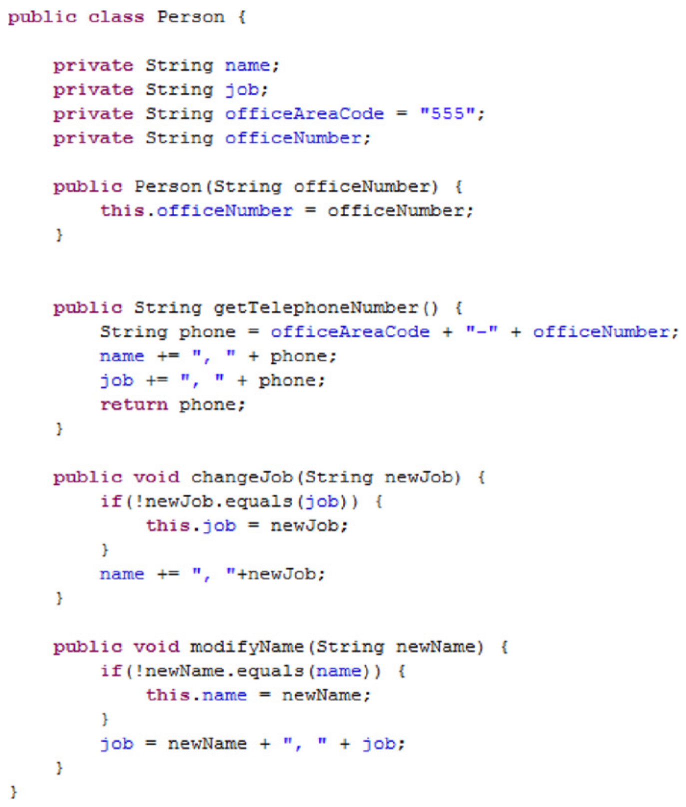
The entity set of a method *m* contains:

- the method *m* itself;
- the attributes accessed by *m*;
- the methods accessed by *m*.

The reason that the entity itself is included in its entity set is so that the condition $d_{ij} = 0$ if $i = j$ 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. In this way, we ensure that the "identity of indiscernibles" condition, which should hold for a proper metric (Pontryagin and Arkhangel'skii, 1990), is satisfied.

Attributes which are references to other classes are considered as entities and 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. Static attributes and methods are considered as entities and they are added to entity sets, because, although they are not instance members, they can still be accessed by instance methods. Hence, they contribute to a concept or to a distinct piece of functionality.

Constructors are neither considered as entities nor are added in entity sets, since they are special purpose methods used to create objects of the class they belong to and as such they cannot be removed from this class. Getter and setter methods are also neither considered as entities nor added to entity sets, because the



```
public class Person {

    private String name;
    private String job;
    private String officeAreaCode = "555";
    private String officeNumber;

    public Person(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;
    }

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

Fig. 1. A synthetic example.

attributes that they provide access to are already added to the entity set (i.e. we do not include both the attribute and its public accessors). Delegate methods are neither considered nor added to entity sets, because the methods to which they delegate are already added to the entity set (i.e. we do not include both the delegator and the delegatee methods). Finally, access to attributes or methods of classes outside the system boundary (e.g. library classes) is not taken into account, because we want to preserve the similarity between entities with respect to the context of the examined system.

Based on the definition of the entity sets, the Jaccard distance between two entities α and β with entity sets *A* and *B* respectively is calculated as follows:

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

3.1.2. An illustrative example

To better understand the methodology, we will illustrate its application on a simple synthetic example, of a class with four attributes and three methods, shown in Fig. 1.

Table 1 shows the distance matrix for this example and Fig. 2 shows a graphical representation of the class. In this graph, the squares represent attributes, the circles represent methods and the edges indicate that a dependency exists between two entities. Furthermore, the length of the edges is proportional to the distances between the class members. Applying the hierarchical clustering algorithm on this class we obtained the dendrogram shown in Fig. 3. It is easy to see that there are 5 merging points (whose height level is shown with vertical lines) that produced an equal number of clusters. These clusters are:

- $C_1 = \{name, job\}$ at the merging point with height 0.4;
- $C_2 = \{modifyName(), changeJob()\}$ at the merging point with height 0.5;

Table 1
Distance matrix for the class of Fig. 1.

	name	*job*	*officeAreaCode*	*officeNumber*	*changeJob*()	*modifyName*()
job	0.4					
officeAreaCode	0.8	0.8				
officeNumber	0.8	0.8	0.67			
changeJob()	0.6	0.6	1	1		
modifyName()	0.6	0.6	1	1	0.5	
getTelephoneNumber()	0.71	0.71	0.6	0.6	0.67	0.67

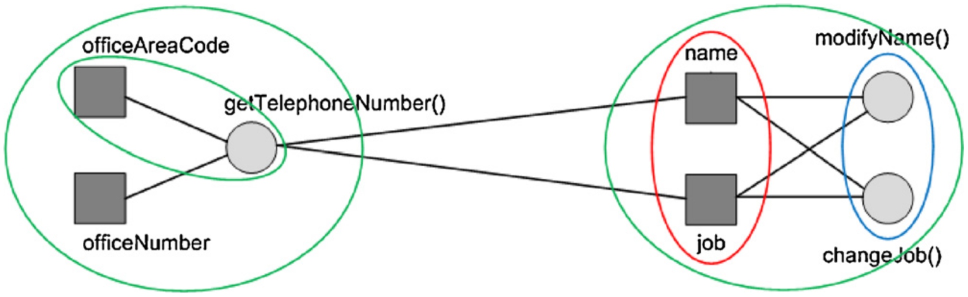


Fig. 2. Graph corresponding to the class of Fig. 1.

- C_3 = {*name*, *job*, *modifyName*(), *changeJob*()} at the merging point with height 0.6;
- C_4 = {*officeAreaCode*, *getTelephoneNumber*()} at the merging point with height 0.6;
- C_5 = {*officeAreaCode*, *officeNumber*, *getTelephoneNumber*()} at the merging point with height 0.67.

Clusters C_1 and C_2 were later rejected for different reasons that are going to be discussed in the following sections.

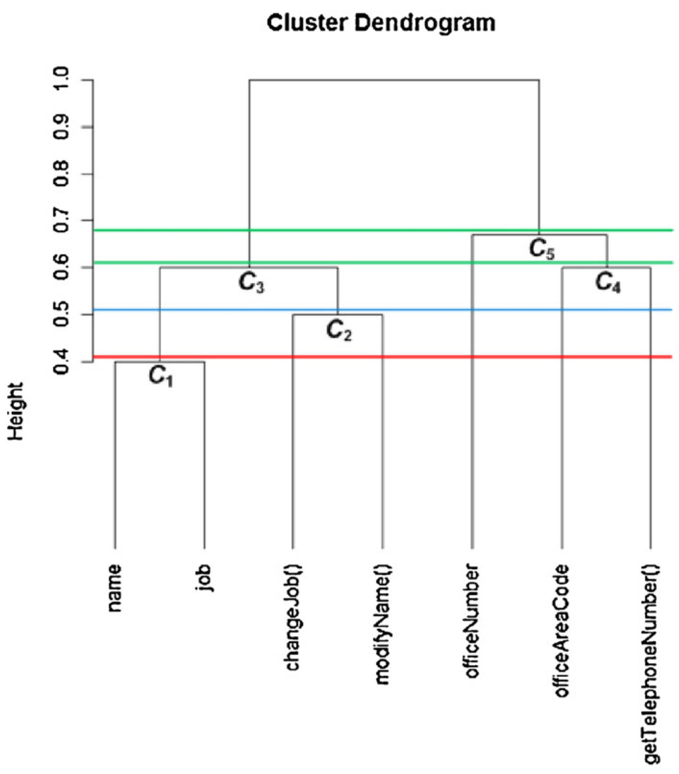


Fig. 3. Dendrogram resulting from the application of hierarchical algorithm for the class of Fig. 1.

3.1.3. *Detection of extractable concepts*

As we have already mentioned, the goal of our methodology is to identify conceptually similar and meaningful groups of class members that can be extracted into separate classes. Our approach for detecting the extractable concepts consists of three main steps:

1. Apply the hierarchical agglomerative clustering algorithm to get the dendrogram which demonstrates how the clusters were formed.
2. Get the clusters before the last merging point. The reason for selecting this particular merging point is that these clusters have the highest merge distance between them in the dendrogram. This means that the class members included in these final clusters access very few common members or none at all. We call these clusters *general concepts*, because they constitute high-level discernible concepts that will eventually be used to organize the refactoring suggestions. In the dendrogram of the illustrative example (Section 3.1.2) shown in Fig. 3, the general concepts are clusters C_3 and C_5. Our method can split a class in more than two sub-classes, since there can be more than two general concepts.
3. For each general concept, examine the corresponding subtree to identify the extractable concepts, which are the actual refactoring opportunities. The reason for further analyzing the general concepts, rather than suggesting them as the final refactoring opportunities, is primarily for the purpose of completeness. To achieve completeness we aim to capture not only the general concepts, but also subconcepts that might constitute better design solutions. Subconcept is a cluster that has been formed at an earlier stage than the general, describes a meaningful concept by itself and has more than one element. The subconcepts can be obtained by iteratively visiting the merging points starting from the leaf nodes of the subtree up to the root node (representing the general concept). At each merging point being traversed, we distinguish two cases:
   (a) If at least one of the child clusters forming the parent cluster (at the merging point) consists of a single class member, then both child clusters are rejected as not being extractable concepts. The reason we reject the non-single-element cluster is because, while it fulfills one of the conditions to qualify as a subconcept (i.e. it has more than one elements), it fails to fulfill the other condition; it does not constitute a complete

concept, because a single other class member is sufficiently related to this cluster, so that it is merged with it in the very next step.

(b) If each of the merged child clusters consists of two or more class members, then they are both accepted as extractable concepts.

Eventually, the extractable concepts include the general concepts and their subconcepts that were identified by the aforementioned iterative process. In the illustrative example shown in Fig. 3 (Section 3.1.2), the extractable concepts are the general concepts represented by clusters C_3 and C_5 and clusters C_1 and C_2 as subconcepts of cluster C_3.

3.1.4. Eliminating illegal candidate classes

Refactoring is conceived as a code restructuring that does not affect its external behavior (Fowler et al., 1999). To this end, our method inspects the classes identified by the clustering as candidates to be extracted to assess (a) whether they have a sufficient substantive functionality and (b) whether the suggested refactorings would indeed preserve the behavior of the program.

The rules imposed to ensure a certain degree of functionality are as follows:

- The class to be extracted should contain more than one entity. A single member cannot describe a concept sufficiently enough.
- The class to be extracted should 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.

The preconditions[1] required for behavior preservation are as follows:

- Abstract methods should not be extracted for two reasons. First, it can break polymorphic method invocations. Second, the extraction of an abstract method would force the extracted class to be declared as abstract. As a result, the field holding a reference to the extracted class could not be initialized with an object having the type of the extracted class.
- Fields that have a visibility higher than private and are used by a class other than the source should not be extracted. Since public and protected fields can be directly accessed by third classes, their extraction would require the modification of these classes. This would violate the local nature of the Extract Class refactoring (i.e. changes should affect only the original class). Alternatively, a more complex solution would be to create public accessors in the original class that delegate to the corresponding accessors in the extracted class.
- Methods that override an abstract or a concrete method of the super class of the original class should not be extracted. Extracting a method that overrides an abstract method would cause compilation errors since the original class would no longer provide an implementation for the abstract method. Similarly, extracting a method that overrides a concrete method would affect the behavior of the original class and the classes that were using the extracted method since the source class would inherit the behavior of the method defined in its super class.
- The class to be extracted should not contain a method that makes any super method invocations.
- Methods that are synchronized or contain a synchronized block should not be extracted since according to Schäfer et al. (2010) extracting "a synchronized method [. . .] can result in the method acquiring a different lock when executed". In the same work, the authors propose to transform the synchronized method with a method containing a synchronized block that explicitly refers to the appropriate lock. This solution deteriorates the understandability of the code.

Violation of the behavior preservation preconditions might introduce compilation errors to the code or alter the external behavior of the program. In the example of Section 3.1.2, the first cluster C_1 was rejected because it only contained attributes.

3.2. Ranking and presentation of refactoring opportunities based on their impact on design quality

Once a set of candidate refactorings has been identified through the aforementioned clustering and filtering process, they are ranked according to their potential impact on the system's design quality. The ranking of the proposed solutions is important, especially in the cases of systems presenting a large number of refactoring opportunities, since it allows the developers to focus on parts of the software that their design would benefit more from preventive maintenance activities.

To estimate the anticipated design improvement for each suggested refactoring, we use the *Entity Placement* (Tsantalis and Chatzigeorgiou, 2009) metric. The Entity Placement metric 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 reason we chose the Entity Placement metric over traditional cohesion and coupling metrics is exactly its compositional 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 will deteriorate (because a new class with dependencies to other classes is introduced). A candidate refactoring is a good design solution if the improvement of cohesion is significantly greater than the deterioration of coupling. Therefore, it is less probable that well-designed classes are proposed for decomposition. The Entity Placement metric can measure this relative change as it evaluates the overall design quality of the system in terms of both coupling and cohesion.

The definition of the entity sets used in the calculation of the Entity Placement and the definition of the entity sets used in the calculation of the distance metric employed in the clustering algorithm are significantly different. For the Entity Placement we used the entity sets exactly as defined by Tsantalis and Chatzigeorgiou (2009), while for the calculation of the distance metric we used the entity sets as defined in Section 3.1.1. Furthermore, the distance between an entity and a class is only employed in the calculation of Entity Placement and not for the purpose of clustering.

The Entity Placement value for a class C (EP_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 the class.

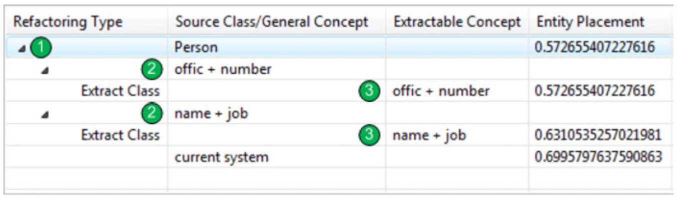
$$EP_C = \frac{\sum_{e_i \in C} distance(e_i, C)/|entities \in C|}{\sum_{e_i \notin C} distance(e_i, C)/|entities \notin C|} \quad (2)$$

The Entity Placement value for a system ($EP_S ystem$) is the weighted average of the Entity Placement values of the classes belonging to the system.

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

where e is the entity set of a class member, C is the entity set of a class and $distance(e, C)$ is the Jaccard distance between entity e and class C, exactly as defined by Tsantalis and Chatzigeorgiou (2009).

[1] A formal definition of these preconditions following Opdyke's notation (Opdyke, 1992) can be found in the work by Tsantalis and Chatzigeorgiou (2009).



Refactoring Type	Source Class/General Concept	Extractable Concept	Entity Placement
◢ ①	Person		0.572655407227616
◢ ②	offic + number		
Extract Class	③	offic + number	0.572655407227616
◢ ②	name + job		
Extract Class	③	name + job	0.6310535257021981
	current system		0.6995797637590863

Fig. 4. Presentation of identified refactoring opportunities.

To calculate the Entity Placement value without having to actually apply the refactoring on the source code, our method adopts a virtual application procedure.

1. First, an empty class entity set is created.
2. For each extracted entity, its origin class is changed from the source class to the new class.
3. The entity sets of all the entities that access or are accessed by the extracted entities are updated.
4. The extracted entities are inserted in the entity set of the new class.
5. The extracted entities are removed from the entity set of the source class.

Once this procedure is completed, the new Entity Placement value for the system that would result from the application of the specific refactoring is computed. It should be noted that the lower the Entity Placement value, the better the resulting design is anticipated to be.

Our methodology may produce multiple suggestions per class according to the number of extractable concepts identified by the algorithm. As already mentioned, at a first level, the candidates that violate preconditions are eliminated. Second, candidates that deteriorate the design quality of the system in terms of Entity Placement are excluded. In the example of Section 3.1.2, the second cluster C_2 was excluded because it produced a worst Entity Placement value for the system.

Regarding the presentation of the identified refactoring opportunities, we use three levels as shown in Fig. 4. The first level shows the classes that were suggested to be refactored, the second level shows the general concepts identified for each class and the third level shows all the extractable concepts identified for each general concept. The labeling of the general and extractable concepts is based on term frequency. We tokenize the names of the extracted entities and we calculate their frequency. Then, we find the terms with the maximum frequency and concatenate them using the plus (+) symbol. The label for an extractable concept is produced from the entities that are specific to this concept, while the label for a general concept is calculated from the collection of the terms of all the extractable concepts it contains.

The extractable concepts are sorted in ascending order according to their Entity Placement value relatively to their sibling concepts within the same general concept. Then, the elements in each preceding level are assigned the minimum Entity Placement value of the elements they contain, for example, the general concept is assigned the minimum value of its extractable concepts and the class is assigned the minimum value of its general concepts. Finally, all elements of the table are sorted in ascending order relatively to the elements of the same level.

3.3. Application of the Extract Class refactoring

We used Eclipse's Java Development Toolkit (JDT) for the application of the refactoring. 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. We used the Preview Wizard in Eclipse Language Toolkit (LTK) to preview the changes. 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:  Sort the extracted methods according to Algorithm 2.
9:  for all the extracted methods do
10:   if there exists an assignment of a field of the source class or an
    invocation of a method of the source class then
11:     Add a parameter of source class type to the extracted class.
12:     Replace field assignments with setters.
13:     Modify the method invocations to source class so that they are
    invoked through the introduced parameter.
14:     Replace "this" with the parameter.
15:   end if
16:   if there exists a source class field access then
17:     Add a parameter of the same type as the field.
18:   end if
19:   Modify invocations of any other extracted method, if necessary.
20:   Add the method in the new class.
21:   if the method is invoked (normal or super method invocation)
    by a class other than the source then
22:     Leave a delegate of the method in the source class.
23:   end if
24: end for
25: Add a reference of the new class in the source class.
26: Modify the accesses of any member of the new class in the source
    class.
27: Provide public accessors for the attributes of the source class and
    change the modifiers of the methods of the source class if
    necessary.
```

Algorithm 2. Examination order of extracted methods

```
1:  Set of extracted methods and their invocations to other extracted
    methods.
2:  for each method that does not invoke any extracted method do
3:    Assign a level equal to 0.
4:  end for
5:  for each method that invokes other extracted methods do
6:    Assign a level equal to $-1$.
7:  end for
8:  while there exists at least one extracted method having a level
    equal to $-1$ do
9:    for each extracted method $m$ having a level equal to $-1$ do
10:     if all the methods invoked by $m$ have a level $-1$ then
11:       Assign to $m$ a level equal to the maximum level of the
    invoked methods +1.
12:     end if
13:   end for
14: end while
15: Sort the extracted methods according to their level in ascending
    order.
```

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). The extracted fields are then 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 8–24). First, the order according to which the extracted methods will be examined is significant, since the existence of call dependencies between extracted methods may cause additional alterations in their signatures (e.g. if an extracted method a invokes another extracted method b, it may be required

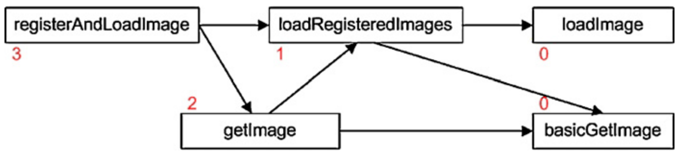


Fig. 5. Invocation dependencies among the extracted methods in class `Iconkit`.

to introduce additional parameters in the signature of *a* in order to pass them as arguments to the invocation of *b*). As a result, the extracted methods should be examined in the appropriate order to ensure that the arguments of the extracted method invocations will match the final parameter list of the corresponding methods. Algorithm 2 describes the algorithm used for determining the correct examination order of the extracted methods. The algorithm employs the notion of level to represent the dependency level of a given method in a chain of invocations. Level values are actually used to determine the examination order of the extracted methods, in the sense that a method should be examined only if all methods having a lower level have been already examined.

For the sake of simplicity, this algorithm presupposes that there do not exist cyclic invocation dependencies among the extracted methods. The existence of cyclic dependencies would cause the algorithm to fall into infinite recursion. To overcome this problem, we first identify all strongly connected components (Tarjan, 1972) (i.e. cycles) in the directed graph representing the dependencies among the methods. In the case an examined edge (i.e. method invocation) connects two nodes (i.e. methods) belonging to the same strongly connected component, we compute the average level of the methods that they depend on and have already obtained a level value, and promote the method corresponding to the minimum average.

The application of the algorithm will be demonstrated on a real example taken from open-source project JHotDraw 5.3. Fig. 5 illustrates the invocation dependencies among methods extracted from class Iconkit. As it can be observed from this figure, the level of each method is equal to the maximum level of the methods that it invokes plus one. The methods that do not invoke any of the extracted methods have a level equal to zero. The extracted methods should be examined according to their level in ascending order to ensure that the arguments of the extracted method invocations will match the final parameter list of the corresponding methods.

After the extracted methods are sorted, we will examine what members they access from the source class. If an extracted method assigns a variable or invokes a method of the source class, it is likely that it may change the state of the 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 is feasible in the new system (line 11). 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 17); 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 19). As the final step, the newly modified methods are added in the newly extracted class (line 20).

Having dealt with the changes in the new class, it is 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 class or the newly extracted 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 22). Then, a field having the type of the newly created class is added in the source class (line 25)and it is initialized by invoking the default constructor. If the extracted fields are initialized in the source class, then they should be initialized in the extracted class as well. There are two cases where the fields can be initialized: if they are initialized where they are declared, then the initialization expression is extracted along with the declaration to the extracted class; if they are initialized in the source class constructor, then the assignment statement is replaced by a setter invocation. The accesses of any members of the new class are appropriately modified in the source class, so that they can be accessed by the newly added reference of the extracted class (line 26). 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 27).

Let us now revisit our example in Section 3.1.2 and apply the chosen refactoring.

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.

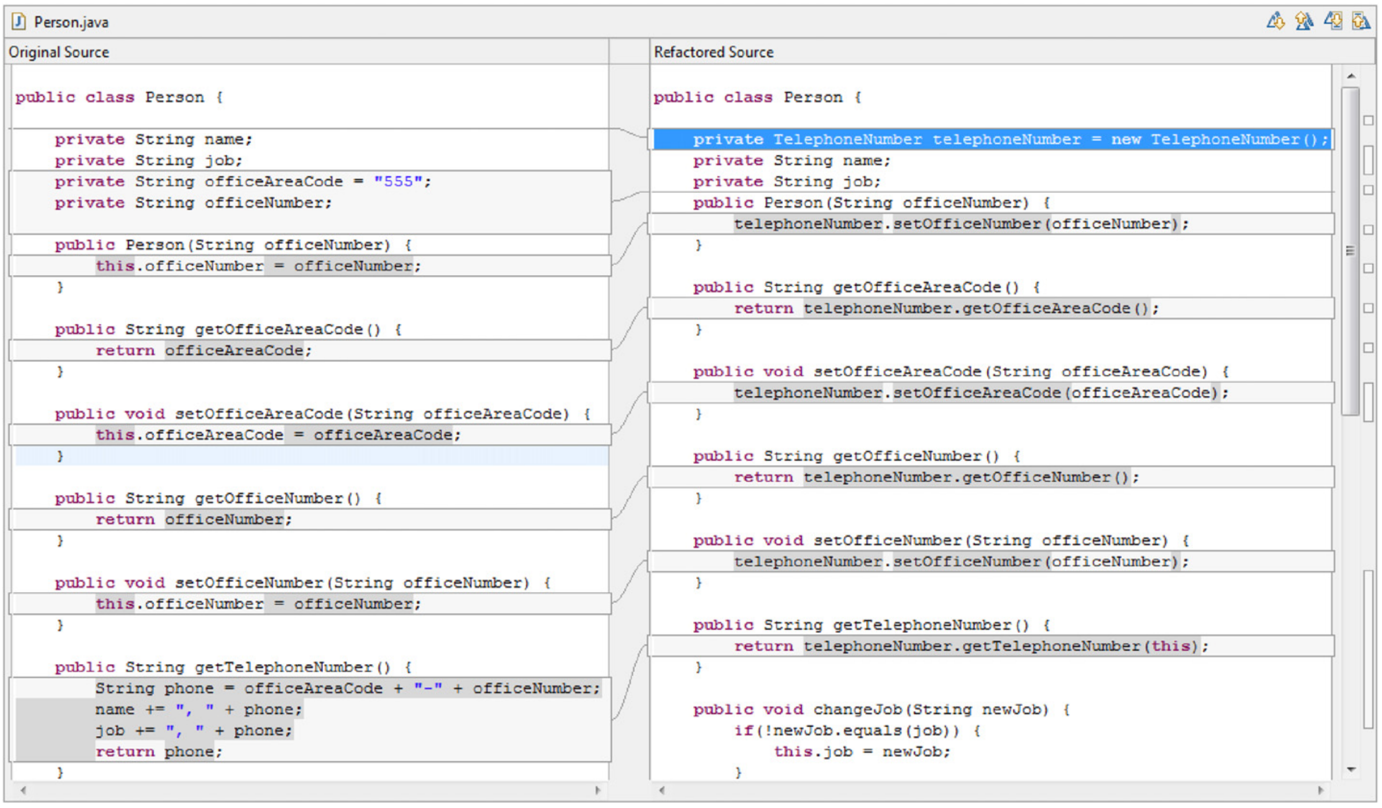
Fig. 6 shows some of these changes in the source class via the Preview Wizard. Fig. 7 shows the code for the extracted class.

As of version 3.6, Eclipse allows the user to perform "Extract Class" refactorings, however, in multiple levels. First, the user must extract the fields in a new class and then perform a series of "Move Method" refactorings. This adds to the required human effort to apply the refactoring. Therefore, we propose our own application algorithm so that the refactoring can be applied in a single step. Furthermore, Eclipse does not have the ability to automatically identify Extract Class refactoring opportunities. As a result the proposed approach provides a more adequate support for all the steps of the Extract Class refactoring process (identification, behavior preservation, assessment of impact to the quality, application) This makes the proposed tool and methodology a more efficient way to perform such a refactoring.

4. Evaluation

In this section, we present the results of our evaluation for the proposed technique. We performed three types of evaluations:

1. For the first type, we asked the designers of three systems to manually identify extractable concepts and used their findings as the ground truth. We then extracted the precision and the recall of our approach (based only on structural measures as the distance metric for the clustering) and two alternative approaches (based only on semantic measures and on a combination of



```
public class Person {

    private String name;
    private String job;
    private String officeAreaCode = "555";
    private String officeNumber;

    public Person(String officeNumber) {
        this.officeNumber = 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 class Person {

    private TelephoneNumber telephoneNumber = new TelephoneNumber();
    private String name;
    private String job;
    public Person(String officeNumber) {
        telephoneNumber.setOfficeNumber(officeNumber);
    }

    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;
        }
```

Fig. 6. Changes introduced in class *Person*.

```
public class TelephoneNumber {

    private String officeAreaCode = "555";
    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;
    }
}
```

Fig. 7. The code of class *TelephoneNumber*.

semantic and structural measures). In this experiment, we had two objectives. The first one was to see if our methodology can actually identify new concepts which were improperly embedded in another class. The second was to investigate the effect of semantic metrics on the results of the identification process of our methodology.

2. In the second type, we applied a series of Extract Class refactorings as suggested by our tool to a well-known open-source project and consulted with an expert quality assessor. In this experiment, we were interested to know if the suggested refactorings would actually be applied by the developer and to confirm that they have a positive impact on the understandability of the code and the design quality of the system once they are applied.
3. In the third type, we compared the progression of the Entity Placement metric, after the sequential application of the refactorings in the second experiment, with the progression of traditional cohesion and coupling metrics. In this experiment, we wanted to evaluate the ability of the Entity Placement metric to quantify the impact of the performed refactorings on the design quality of the system.

4.1. *Evaluation of precision and recall*

For the first part of the evaluation, we asked from independent evaluators to manually identify concepts on software systems that were developed by each one of them individually. Each evaluator had knowledge only of her or his own system and was unaware of our methodology. The concepts identified by the independent evaluators were considered as a set of True Occurrences (TO), allowing the extraction of the precision and recall of our approach.

The three evaluators that participated in this experiment are graduate students (two MSc students and one PhD candidate). All three students' primary research field is Software Engineering and they have significant experience in object-oriented design (ranging from 6 to 12 years). The analyzed projects were developed in the context of their research conducted in the Service Systems Research Group in the Department of Computing Science at the University of Alberta. The students were unaware of the proposed technique in order to guarantee that there will be no bias in their judgment.

Table 2
Statistical information for examined systems.

Project	CLRServerPack	TPMSim	CoverFlow
Number of classes	33	161	103
Total number of methods (static)	242 (27)	1000 (57)	312 (3)
Total number of attributes (static)	78 (35)	542 (122)	183 (9)
Source lines of code	4652	12,631	3414
Number of classes suggested to be refactored	6	35	7
Average source lines of code per class	141	78	33
Average number of suggestions per class	5.66	2.74	2.14
Running time of the tool (ms)	484	1763	1076

The first project called CLRServerPack is a an API for a potential collaborative PDF annotating client (including a client) and it mainly manages the data storage in and retrieval from a database. It has been developed for 8 months and it is in a mature level. The second project named TPMSim is a framework for simulating services aware software and also contains a simulation engine built on that framework. It has been developed for 4 years of which the first two were the most active. The third project called TAPorWithCoverFlow is a web-based text analysis environment with integrates some text analysis web-services. It has been developed for 6 months and is still under development. Some statistical information for the three systems is presented in Table 2.

The results for this part are presented in three ways:

1. Individual calculation of precision and recall for each examined class in order to assess the accuracy of the tool for each special case (i.e. examined class). For the calculation of precision and recall, we need the following definitions:
   - True Positive (TP): a concept identified by the independent expert and also by the proposed technique.
   - False Positive (FP): a concept identified by the proposed technique, but not by the independent expert.
   - False Negative (FN): a concept identified by the independent expert, but not by the proposed technique.

   TP, FP and FN are calculated at a coarse-grained level, meaning that the concepts identified by the proposed technique and by the independent evaluator should exactly match, in terms of their methods and attributes.
2. We report the ratio of failed cases, calculated by (4), out of the total number of examined classes, where a *failure* is defined as a case where only one of the evaluator or the tool identified at least one an extractable concept for the examined class, but not the other. Practically, this means that either the tool or the evaluator identified a class as being problematic, but not both. Obviously, for this reason, either the precision or the recall cannot be calculated. In the particular cases where the tool identified a class as being problematic but the developers did not, the designers claimed that they were reluctant to decompose these classes. According to them, this was because these classes were designed to capture real-world objects. This was evident by the nature of the systems as they required to define network entities (like endpoints and servers in TPMSim) or describe entities that correspond to tables of relational databases (as in the case of CLRServerPack and CoverFlow).
3. As a *successful* case, we consider the case where the tool was able to identify a problematic class, a fact that was also confirmed by the developer. For the successful cases, we calculated the accuracy of the tool (in terms of precision and recall) in identifying the exact problems of the examined class (in terms of extractable concepts).

$$FailureRate = \frac{\#failures}{\#total\_examined\_classes} \tag{4}$$

As it can be observed from Table 2, the first and the third projects are small sized (with respect to lines of code), while the second one is medium sized. Furthermore, the third project has a large degree of modularization (33 lines per class) which justifies the low average number of refactoring suggestions per class. On the contrary the first project demonstrates a larger number of suggestions per class as it is evident by the comparatively smaller degree of modularization (141 lines per class). Finally, the second project shows a medium level of modularization, but we identified a relatively small number of suggestions per class. The selected projects cover a wide spectrum of design decisions with respect to modularization, a fact facilitates the generalization of our findings.

The analysis of these projects has been restricted to a selected number of classes presenting at least one possible extractable concept (according to the findings of the proposed tool), since the evaluation of the entire project would have been prohibitive with respect to time and effort by the evaluators.

4.1.1. *Using structural measures as distance metric*

The primary goal of this part of the experiment is to assess the ability of the proposed approach to match the concepts identified by human expertise to a large extent. A secondary goal is to quantify the difficulty of manually identifying Extract Class refactoring opportunities and applying the appropriate refactoring in terms of consumed time. For each identified concept the evaluators had to provide the entities that the concept comprise. The task of the authors was to record the findings of the evaluators and the time that took them to identify all concepts for a specific class.

As it can be observed from Tables 3–5, the tool had a *FailureRate* of 33.33% for CLRServerPack, 27.8% for TPMSim and 14.29% for CoverFlow. For the successful cases, the tool had a precision of 77.1%,

Table 3
Evaluation results on CLRServerPack with structural measures.

Class	Concepts	Ident. time (m:s)	TO	#Concepts by tool	TP	FN	FP	Precision (%)	Recall (%)	Failure
1. AnnotationDatabase	FindUpdate + delete	2:15	3	3	3	0	0	100	100	
2. PrivacyManagement. Notifications	Not found	1:20	0	1	0	0	1	0	N/A	√
3. PersonsManagement	Add Authenticate	0:28	2	2	2	0	0	100	100	
4. DatabaseController	Management Add	1:05	2	1	1	1	0	100	50	
5. User	Not found	0:57	0	2	0	0	2	0	N/A	√
6. BibTex	Title + abstract	0:48	1	12	1	0	11	8.33	100	
Average								77.1	87.5	
Failure rate										33.3

Table 4
Evaluation results on TPMSim with structural measures.

Class	Concepts	Ident. time (m:s)	TO	#Concepts by tool	TP	FN	FP	Precision (%)	Recall (%)	Failure
1. DocumentRange	Range + document + source	1:05	1	1	1	0	0	100	100	
2. DocumentModifier	Modified + document Log	1:49	2	2	2	0	0	100	100	
3. Clock	Pause + resume	2:00	1	3	1	0	2	33.33	100	
4. JobTracker	Endpoint + result	2:58	1	2	1	0	1	50	100	
5. Resource	Not found	2:59	0	1	0	0	1	0	N/A	√
6. Dashboard	Generate + chart	2:49	1	1	0	1	1	0	0	
7. NetworkedTPM	Job + server	5:21	1	2	1	0	1	50	100	
8. NetworkedEndPoint	Not found	5:11	0	3	0	0	3	0	N/A	√
9. SimPlayer	Graph + series Pie + chart	2:41	2	2	2	0	0	100	100	
10. Network	Not found	6:21	0	1	0	0	1	0	N/A	√
11. Data	Data + remaining	2:11	1	1	1	0	0	100	100	
12. NetworkedDepot	Not found	2:36	0	1	0	0	1	0	N/A	√
13. Database	Connect	2:00	1	3	1	0	2	33.33	100	
14. SimulationBuilder	XML + parse Object	3:52	2	8	0	2	8	0	0	
15. TPMSimulator	Table + database	3:10	1	2	1	0	1	50	100	
16. NetworkedCDSM	Not found	1:31	0	3	0	0	3	0	N/A	√
17. Metrics	Listener	1:08	1	2	1	0	1	50	100	
18. PlaybackControl	Slider	0:38	1	3	1	0	2	33.33	100	
Average								53.8	84.6	
Failure rate										27.8

Table 5
Evaluation results on CoverFlow with structural measures.

Class	Concepts	Ident. time (m:s)	TO	#Concepts by tool	TP	FN	FP	Precision (%)	Recall (%)	Failure
1. File	Result	1:04	1	2	1	0	1	50	100	
2. Result	Change Content	0:43	2	5	1	1	4	20	50	
3. Document	Change	0:33	1	1	1	0	0	100	100	
4. AbstractListWordsForm	RadioCombo + box	1:13	2	1	1	1	0	100	50	
5. AbstractDateFinderForm	Date Display	0:41	2	1	1	1	0	100	50	
6. ResultsPanel	Catalog	0:58	1	2	1	0	1	50	100	
7. CatalogController	Not found	0:57	0	1	0	0	1	0	N/A	√
Average								70	75	
Failure rate										14.29

53.8% and 70% in average for the three projects, respectively, and an average recall of 87.5%, 84.6% and 75%, respectively.

Another interesting observation was that all three evaluators were able to identify completely disconnected components from the rest of the class. These components are usually the best candidates for extraction since they do not bare any dependency from the rest of the class. The evaluators identified such components by using standard tools from Eclipse like the call hierarchy browser and the reference search feature.

Finally, our methodology was able to identify concepts only by using dependency information. We observed that the same concepts were also identified by the evaluators based on conceptual criteria like similar names. An interesting example of such a case is shown in Fig. A.1, where the extractable concept describes functionality about property changes. What is even more interesting about this example is that this particular concept is used in two more classes of the system, which means that extracting it in a new class would increase the reusability of the code.

During the examination of the projects we observed a few interesting cases where the extraction of specific concepts not only improved the understandability but also contributed to certain aspects of the design of the system. For example, there was a case that our approach successfully grouped all the entities which were related with the functionality of the *Subject* role in an Observer pattern instance (Gamma et al., 1995). 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 (Gamma et al., 1995). This case is illustrated in Fig. A.2.

Another set of interesting examples includes cases where the approach successfully managed to separate tangled concerns and responsibilities. In the example shown in Fig. A.3, the evaluator confirmed that our tool accurately suggested that the concept concerning the connection to a database be separated from the one about constructing SQL queries. Moreover, in the example of Fig. A.4 the tool proposed to separate some graphical components from their controllers. This way we make clear the bounds that separate the front-end of the application from its back-end.

4.1.2. *Using semantic measures as distance metric*

The goal of this part of the experiment is to investigate the effect of semantic measures, if used only by themselves in the identification process of our methodology. As the semantic measure, we use the cosine distance between the term frequency vectors of two entities (attributes or methods) e_i and e_j defined as:

$$dist(e_i, e_j) = 1 - \frac{\vec{e_i} \cdot \vec{e_j}}{\|\vec{e_i}\| \times \|\vec{e_j}\|} \qquad (5)$$

This measure extends the Conceptual Similarity between Methods (CSM) by Marcus and Poshyvanyk (2005) in two ways. First, apart from methods, we also compare the conceptual similarity between attributes, since they also participate in the identification process and, second, the term vectors also contain the frequency of term occurrence in an entity, in order to give additional weight to recurring terms.

For an attribute, its term vector contains the term or terms that comprise its name and words from its javadoc documentation (excluding tags). For a method, its term vector contains its name,

Table 6
Evaluation results on CLRServerPack with semantic measures.

Class	TO	#Concepts by tool	TP	FN	FP	Precision (%)	Recall (%)	Failure
1. AnnotationDatabase	3	2	2	1	0	100	66.7	
2. PrivacyManagement. Notifications	0	3	0	0	3	0	N/A	√
3. PersonsManagement	2	4	2	0	2	50	100	
4. DatabaseController	2	6	1	1	5	16.7	50	
5. User	0	6	0	0	6	0	N/A	√
6. BibTex	1	16	1	0	15	6.25	100	
Average						43.2	79.2	
Failure rate								33.3

Table 7
Evaluation results on TPMSim with semantic measures.

Class	TO	#Concepts by tool	TP	FN	FP	Precision (%)	Recall (%)	Failure
1. DocumentRange	1	1	1	0	0	100	100	
2. DocumentModifier	2	1	1	1	0	100	50	
3. Clock	1	4	1	0	3	25	100	
4. JobTracker	1	3	1	0	2	33.3	100	
5. Resource	0	2	0	0	2	0	N/A	√
6. Dashboard	1	1	1	0	0	100	100	
7. NetworkedTPM	1	2	1	0	1	50	100	
8. NetworkedEndPoint	0	4	0	0	4	0	N/A	√
9. SimPlayer	2	1	0	1	1	0	0	
10. Network	0	4	0	0	4	0	N/A	√
11. Data	1	0	0	1	0	N/A	0	√
12. NetworkedDepot	0	2	0	0	2	0	N/A	√
13. Database	1	3	1	0	2	33.33	100	
14. SimulationBuilder	2	12	0	2	12	0	0	
15. TPMSimulator	1	5	0	1	5	0	0	
16. NetworkedCDSM	0	3	0	0	3	0	N/A	√
17. Metrics	1	2	1	0	1	50	100	
18. PlaybackControl	1	1	1	0	0	100	100	
Average						46.5	62.5	
Failure rate								33.3

Table 8
Evaluation results on CoverFlow with semantic measures.

Class	TO	#Concepts by tool	TP	FN	FP	Precision (%)	Recall (%)	Failure
1. File	1	2	1	0	1	50	100	
2. Result	2	5	1	1	4	20	50	
3. Document	1	1	1	0	0	100	100	
4. AbstractListWords	2	0	0	2	0	N/A	0	√
5. AbstractDateFinder	2	1	1	1	0	100	50	
6. ResultsPanel	1	0	0	1	0	N/A	0	√
7. CatalogController	0	1	0	0	1	0	N/A	√
Average						43.8	75	
Failure rate								42.9

the identifiers of its parameters, accessed fields and declared or accessed local variables in its body, the names of the invoked methods and words from its javadoc documentation (excluding tags). In case of complex identifiers (i.e. identifiers that consist of multiple words) in the form of camel case strings (e.g. "aLocalVariable") or words separated by underscores (e.g. "A_STATIC_FIELD"), first, they need to be split in simple terms which are then stemmed. Finally, we exclude from the term vectors stop words, such as prepositions and articles.

As it can be observed from Tables 6–8, our approach using semantic measures had a *FailureRate* of 33.33% for CLRServerPack, 33.3% for TPMSim and 42.9% for CoverFlow. For the successful cases, it had a precision of 43.2%, 46.5% and 43.8% in average for the three projects, respectively, and an average recall of 79.2%, 62.5% and 75%, respectively.

4.1.3. Comparison between structural and semantic measures

Table 9 provides an overview of the accuracy achieved by the two considered distance metrics and a combination of them with respect to precision, recall and failure rate. As it can be seen, structural measures clearly outperform the semantic in all three measurements. An interesting observation is that the precision of the tool is negatively affected to greater extent than recall and

Table 9
Comparison between structural and semantic measures.

	CLRServerPack	TPMSim	CoverFlow
Precision (%)			
Structural	77.1	53.9	70.0
Semantic	43.2	46.5	43.8
Combined	42.7	31.5	43.8
Recall (%)			
Structural	87.5	84.6	75.0
Semantic	79.2	62.5	75.0
Combined	79.2	50.0	75.0
Failure rate (%)			
Structural	33.3	27.8	14.3
Semantic	33.3	33.3	42.9
Combined	33.3	33.3	42.9

failure rate, when semantic measures are employed, since they result in more extractable concepts. The reason is that the structural measures, as we have defined them, take into account not only dependencies within the examined class but also dependencies with entities from other classes as well. Practically, this means that two entities are similar not only if one depends on the other, but also if they depend on the same other entities. On the other hand, textual similarity can only be calculated between a pair of entities and is not affected by dependencies to third entities.

Within the context of this experiment and the examined projects, we can observe that dependency information is sufficient to find cooperating entities that contribute to a common task. On the other hand, there are some cases that semantic measures group together entities using or having similar identifiers, but not contributing to the same task. For example, as we can see in Fig. A.5, in class *SimPlayer* from project TPMSim, the designer identified two concepts: one responsible for drawing graphs and another one responsible for drawing charts. However, both methods *generateCharts*( ) and *generateGraph*( ) frequently refer to terms such as *chart*, *plot* and *generate* and as a result, the semantic measures merged the two concepts into one.

Apart from the pure use of structural or semantic measures, we also tried a combination of them giving both measures equal importance (i.e. 0.5 weight). As it can be seen from the table, the results were worse than when the two measures were used individually. As we studied the results, we realized that this is because one distance metric interfered with the results of the other. This produced extractable concepts augmented by unnecessary entities or merged with other concepts and as a result they did not match the concepts identified by the evaluators.

This part of the evaluation showed that structural measures are not only necessary to identify extractable concepts but also sufficient. This is because naming of identifiers is a critical task (as also discussed by Marcus and Poshyvanyk (2005)). Naming is a subjective manual task and thus prone to errors. The designer of a system chooses suitable (to his/her best judgment) names for entities *at the moment* they enter the system. This means that due to further development and/or software degradation, names may become deprecated and may need to change. On the other hand, dependency information clearly and at all times shows the purpose of a group of entities with respect to the system's functionality (assuming that the system operates as desired).

Table 10
Statistical information for JHotDraw 5.3.

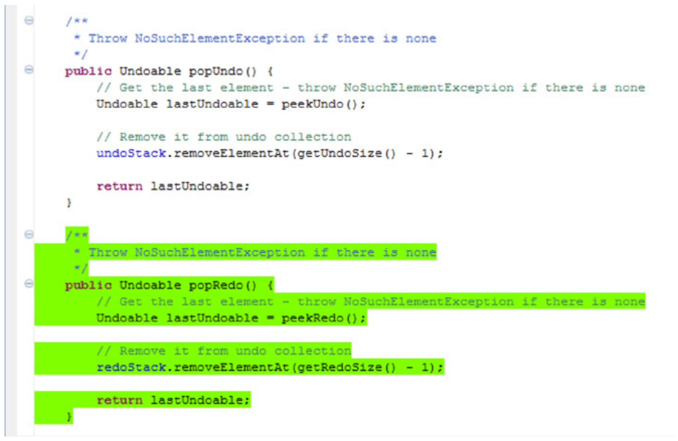
Number of classes	249
Number of methods (static)	2254 (78)
Number of attributes (static)	489 (109)
Source lines of code	14,611
Number of classes suggested to be refactored	32
Number of suggestions per class	2.5
Running time of the tool (ms)	2824

4.2. Expert assessment

The second part of the evaluation was performed on the JHotDraw system (version 5.3), which is a very well-known open-source system with complete and extensive documentation. Some statistical information for this system is presented in Table 10. We used our tool to identify refactoring opportunities for the initial version of the system. Next, for the class that was ranked as top (based on the sorting mechanism described in Section 3.2) we examined all suggested extractable concepts for that particular class and selected the most meaningful one to be refactored. After the application of each refactoring, this process was repeated on the new resulting version of the system. Eventually, we applied 16 of the suggested refactorings and contacted a professional in the business of software quality assessment, to provide his expert opinion. The professional had three years of experience in evaluating the software quality of industrial systems in the context of the services offered by an organization active in the domain of software quality assurance and certification (Deursen et al., 2003; Kuipers and Visser, 2004). The evaluator was asked to answer the following three questions for each applied refactoring.

Table 11
Evaluation results on JHotDraw 5.3.

No.	Name of class	Extracted entities	Q1	Q2	Q3
1.	util.UndoManager	redoStack, pushRedo( ), isRedoable( ), peekRedo( ), getRedoSize( ), popRedo( )	Yes	Yes	No
2.	applet.DrawApplet	fSimpleUpdate, fUpdateButton, createButton( ), setSimpleDisplayUpdate( ), setBufferedDisplayUpdate( )	Yes	Yes	Yes
3.	samples.net.NodeFigure	fConnectors, connectors( ), createConnectors( ), findConnector( ), initialize( )	No	No	No
4.	applet.DrawApplet	fSleeper, startSleeper( ), stopSleeper( )	No	No	No
5.	figures.TextFigure	fWidth, fHeight, fSizeIsDirty, textExtent( ), markDirty( )	Yes	Yes	Yes
6.	contrib.DragNDropTool	dragSource, fDragGestureRecognizers, createDragGestureRecognizer( ), destroyDragGestureRecognizer( )	Yes	No	No
7.	applet.DrawApplet	fFrameColor, fFilleColor, fTextColor, fArrowChoice, fFontChoice, createAttributeChoices( ), createColorChoice( ), setupAttributes( ), createFontChoice( )	Yes	Yes	Yes
8.	util.StorableInput	fMap, map( ), retrieve( ), readStorable( )	Yes	No	No
9.	util.FloatingTextField	fContainer, createOverlay( ), endOverlay( )	Yes	Yes	Yes
10.	util.StorableOutput	fMap, mapped( ), map( )	Yes	No	No
11.	util.StorableOutput	fIndent, incrementIndent( ), decrementIndent( )	Yes	Yes	Yes
12.	figures.InsertImageCommand. UndoActivity	myAffectedImageFigure, myAffectedImageName, setImageFigure( ), getImageFigure( )	Yes	Yes	Yes
13.	standard.StandardDrawingView	fBackgrounds, fForegrounds, addBackground( ), removeBackground( ), addForeground( ), removeForeground( )	Yes	No	Yes
14.	util.Iconkit	fMap, loadImage( ), basicGetImage( )	Yes	Yes	Yes
15.	application.DrawApplication	createEditMenu( ), createColorMenu( ), createArrowMenu( ), createFontMenu( ), createAlignmentMenu( ), createFontStyleMenu( ), createFontSizeMenu( ), createDebugMenu( )	No	Yes	Yes
16.	applet.DrawApplet	fDrawing, initDrawing( ), loadDrawing( ), readFromStorableInput( ), readFromObjectInput( )	No	No	No



```
/**
 * 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;
}
```

Fig. 8. The two methods, *popRedo* (suggested to be extracted) and *popUndo*, are almost identical.

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 11 summarizes the expert assessment for the 16 refactorings applied on JHotDraw 5.3. In 12 of the total 16 cases (75%), 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 13 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 (56.25%), 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, mainly due to the fact that the dependencies between the extracted class members were not easy to identify by manual inspection. In 9 out of 16 cases, 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 classes. For example, the class *UndoManager* contains "undo" and "redo" functionality. The code for both activities is exactly the same. Fig. 8 illustrates this example. The tool suggests two extractable concepts: one for the undo and one for the redo activity. A better approach would be to extract one of the m 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.

Interestingly, we noticed that for a specific class (DrawApplet) there were more than one refactorings accepted by the evaluator. Indeed, this class seems to be a problematic one as it has 42 methods. The expert's opinion on the three refactorings was: "*Overall, the three 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 not all refactorings are 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*".

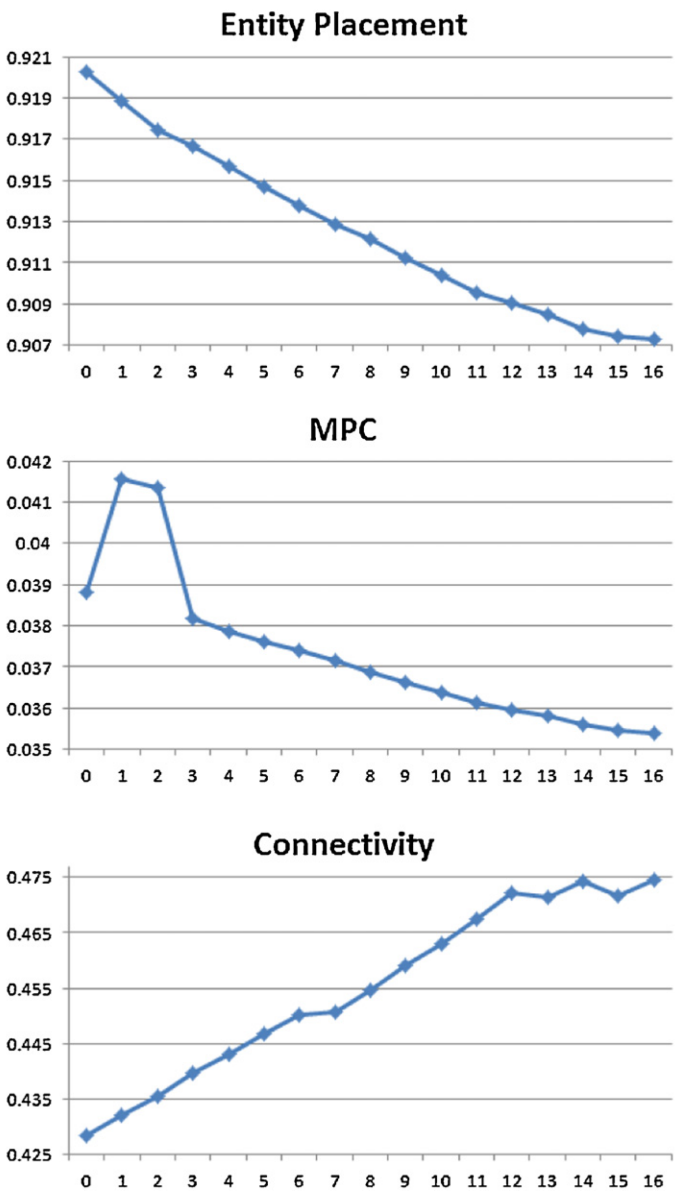


Fig. 9. Evolution of metrics when applying successive refactorings on JHotDraw 5.3.

4.3. Metrics comparison for JHotDraw 5.3

In this part of the evaluation, we measure the impact of the 16 performed refactorings on JHotDraw 5.3 in terms of coupling and cohesion using the message-passing coupling (MPC) (Li and Henry, 1993) metric and the Connectivity (Briand et al., 1998) metric respectively. The values of these metrics were then compared to Entity Placement, which is a metric based on Jaccard distance that captures both cohesion and coupling (as already described in Section 3.2).

The 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

Table 12
Correlation between Entity Placement and MPC and Connectivity.

EP-Conn	EP-MPC
−0.99356	0.88702

common attribute of class *C*, over the total number of method pairs of class *C*.

Although the aforementioned metrics capture the same design quality characteristics as the Entity Placement metric, namely coupling and cohesion, they have major differences in their definitions. Firstly, the MPC metric is based on an absolute count of messages and it is not normalized over a range of minimum and maximum values. On the other hand, Entity Placement is calculated based on distances which are normalized by definition and furthermore, it does not use absolute counts of dependencies since it is employing the notion of sets. Secondly, the Connectivity metric has a discrete binary nature: it considers two methods as either cohesive or not cohesive. On the contrary, Entity Placement has a continuous (within a range) nature, since it captures the degree of similarity between a method and the class it belongs to. Due to these differences, it is not naturally expected for these metrics to be correlated.

Our motivation behind the selection of the MPC metric is that it captures coupling at a more fine-grained level (method interactions) in contrast to other metrics such as CBO (Coupling Between Objects) and Coupling Factor that capture coupling at the class level. Moreover, Connectivity was chosen because it considers two methods as cohesive not only if they access common attributes but also if they invoke each other in contrast to traditional cohesion metrics such as LCOM (Lack of Cohesion Of Methods).

Fig. 9 presents the progression of the Entity Placement metric, MPC (coupling) and Connectivity (cohesion). The *x*-axis in each chart corresponds to the refactored versions of JHotDraw 5.3. The first value on each chart represents the value of the respective metric on the initial system. All metrics have been calculated at the system level. 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 trends expected assuming that overall quality of the system was improved. However, for coupling, a few 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 and 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 (Table 12). A strong negative correlation between Entity Placement and connectivity and a strong positive correlation between Entity Placement and MPC were observed. Thus, it can be argued that the Entity Placement metric is sufficient to evaluate the effect of Extract Class refactorings.

4.4. Threats to validity

4.4.1. Threats to internal validity

A threat to the internal validity of our study is related to the knowledge and expertise of the human evaluators on the examined systems. Inadequate knowledge could lead to limited ability to distinguish the existing concepts within a class and to assess the impact of the suggested refactorings on the maintainability and understandability of the systems. This threat has been partially mitigated by selecting evaluators who were the actual developers of the examined systems in the evaluation of precision and recall (Section 4.1) and by selecting an experienced professional on software quality assessment to provide his expert opinion on a very well-known and well-documented project (JHotDraw) in Section 4.2.

4.4.2. Threats to external validity

Since the experiments have been conducted employing a limited number of evaluators and a limited number of projects, our study lacks the ability of generalizing its findings beyond the selected experimental setup. This threat was partially alleviated by conducting two different types of experiments. In the first type, the evaluators were asked to manually identify extractable concepts without neither having knowledge of our methodology nor having the assistance of our tool. Next, we compared their findings with the tool's suggestions in order to extract the precision and recall of our approach. This allowed us to assess the ability of our approach to conform with human expertise. In the second type, we provided the evaluator, who is a professional in software quality assessment, with a set of already applied refactorings and ask him to provide his expert opinion on whether the newly created classes constituted meaningful and valid concepts and if the applied refactorings improved the understandability of the code. The purpose of this experiment was to assess the conceptually integrity of the refactoring suggestions produced by our approach.

These two different types of experiments covered all essential aspects of the refactoring process, which are adherence to human decisions, improvement of design quality and code understandability, satisfaction of human intuition on what a meaningful concept is.

5. Conclusions and future work

In this work, we proposed a novel method 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 was also described so that it preserves the system's behavior and its syntactical correctness.

We implemented our method as an extension for the JDeodorant Eclipse plugin. The tool shows the developer the candidate entities for extraction by highlighting them in the Java editor and illustrates the changes that will be performed using a Preview Wizard. The use of the tool comprises simple steps and the interface design follows the conventions of Eclipse, which should make it intuitive enough to most developers. The input required is minimal, which makes the tool suitable to novice developers as well.

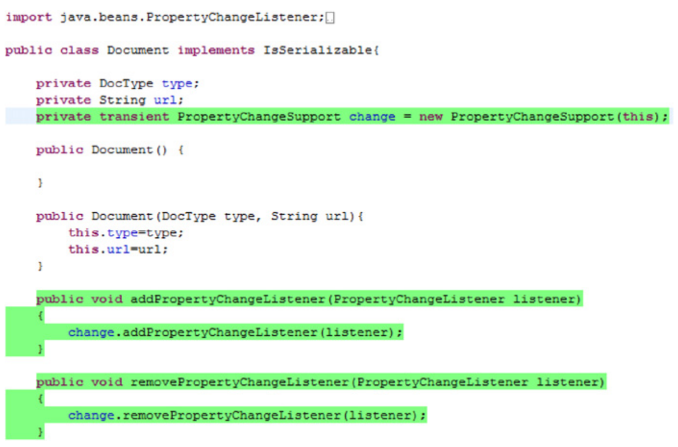
We evaluated the proposed methodology on various systems in terms of precision and recall (using structural, semantic and a combination of these measures as a distance metric), assessment by an expert and metrics. Through this process, we demonstrated that our method can produce meaningful and conceptually correct suggestions and extract classes that developers would recognize as meaningful concepts. We also showed that structural metrics (such as dependency between class members) are a necessary and sufficient criterion for the identification of extractable concepts from a class. Furthermore, the expert confirmed that a good percentage of the proposed refactorings were good solutions that also improved the understandability of the code. Finally, we demonstrated that the suggested refactorings improve the design of the system in terms of coupling and cohesion and that the Entity Placement is a good metric for evaluating the impact of the performed refactorings to the design of the system.

In the future, we would like to explore the possibility of refining our refactoring identification method. In some cases, as it was seen in the evaluation process, the suggestions could have been better and more complete if the clustering algorithm was combined with other methods, like code duplication detection techniques. This would enable the method to identify identical or similar extractable concepts even across classes and suggest a global solution to improve the design of more than one class at once. Finally, we also plan to improve the interface of the tool with visualizations to increase the awareness of the developer of the proposed change and its impact to the system. 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 extractable concept is suggested.

Acknowledgements

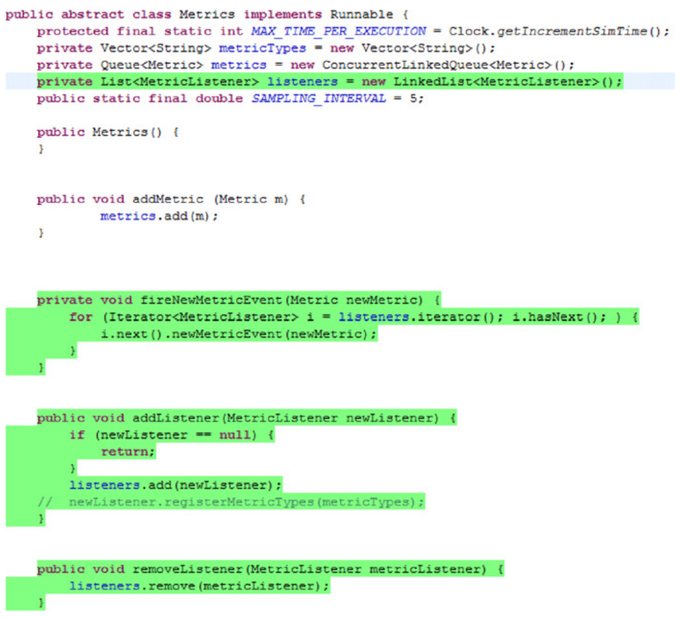
The authors wish to acknowledge the generous support of NSERC, iCORE and IBM. The authors would also like to thank the Software Improvement Group, and more specifically Eric Bouwers and Yiannis Kanellopoulos, for their participation in the evaluation process.

Appendix A.



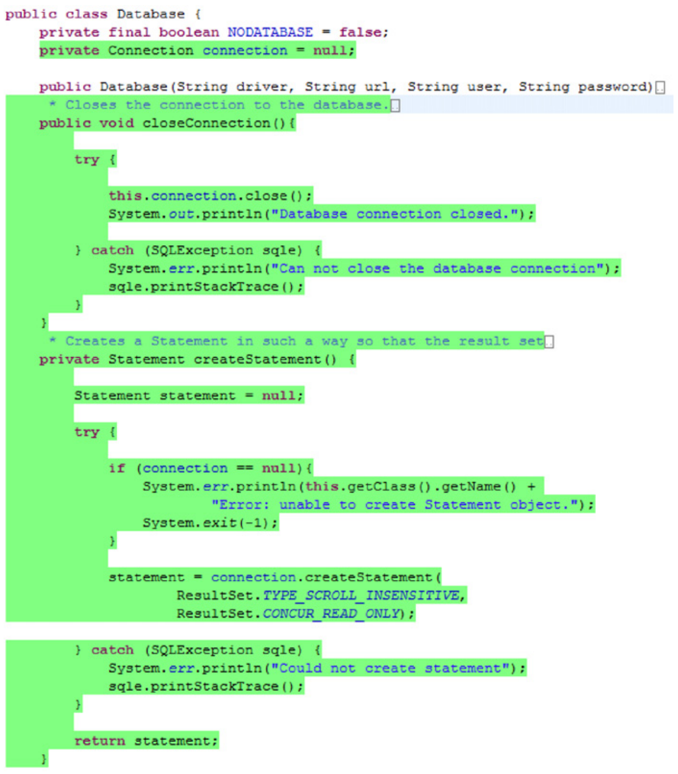
```
import java.beans.PropertyChangeListener;

public class Document implements IsSerializable{

    private DocType type;
    private String url;
    private transient PropertyChangeSupport change = new PropertyChangeSupport(this);

    public Document() {

    }

    public Document(DocType type, String url){
        this.type=type;
        this.url=url;
    }

    public void addPropertyChangeListener(PropertyChangeListener listener)
    {
        change.addPropertyChangeListener(listener);
    }

    public void removePropertyChangeListener(PropertyChangeListener listener)
    {
        change.removePropertyChangeListener(listener);
    }
```

Fig. A.1. Extraction of property change concept.



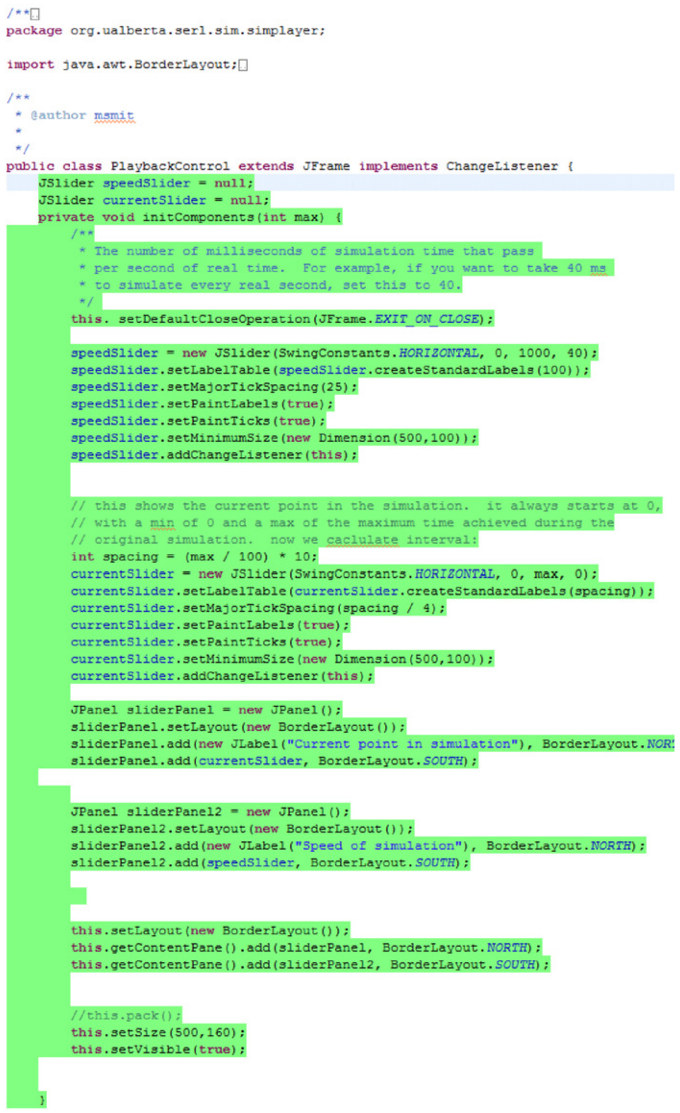
```
public abstract class Metrics implements Runnable {
    protected final static int MAX_TIME_PER_EXECUTION = Clock.getIncrementSimTime();
    private Vector<String> metricTypes = new Vector<String>();
    private Queue<Metric> metrics = new ConcurrentLinkedQueue<Metric>();
    private List<MetricListener> listeners = new LinkedList<MetricListener>();
    public static final double SAMPLING_INTERVAL = 5;

    public Metrics() {
    }


    public void addMetric (Metric m) {
            metrics.add(m);
    }



    private void fireNewMetricEvent(Metric newMetric) {
        for (Iterator<MetricListener> i = listeners.iterator(); i.hasNext(); ) {
            i.next().newMetricEvent(newMetric);
        }
    }



    public void addListener(MetricListener newListener) {
        if (newListener == null) {
            return;
        }
        listeners.add(newListener);
    //  newListener.registerMetricTypes(metricTypes);
    }



    public void removeListener(MetricListener metricListener) {
        listeners.remove(metricListener);
    }
```

Fig. A.2. The *Subject* role of an Observer pattern as identified by the tool.



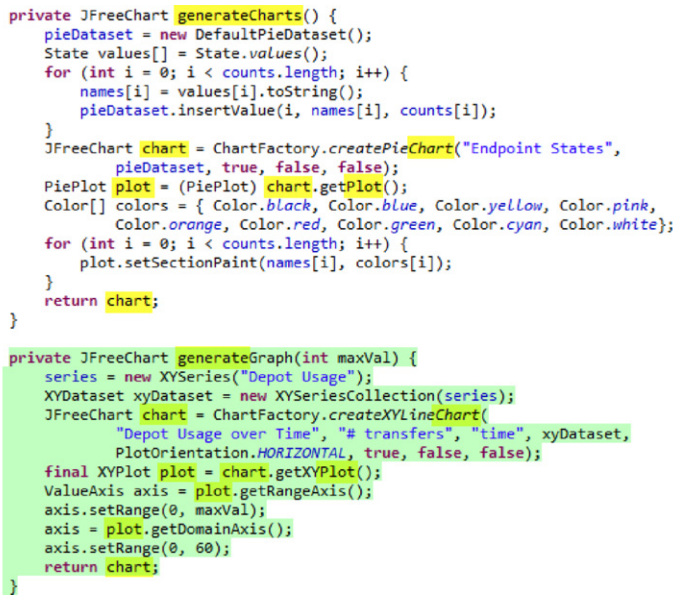
```
public class Database {
    private final boolean NODATABASE = false;
    private Connection connection = null;

    public Database(String driver, String url, String user, String password)
     * Closes the connection to the database.
    public void closeConnection(){

        try {

            this.connection.close();
            System.out.println("Database connection closed.");

        } catch (SQLException sqle) {
            System.err.println("Can not close the database connection");
            sqle.printStackTrace();
        }
    }
     * Creates a Statement in such a way so that the result set
    private Statement createStatement() {

        Statement statement = null;

        try {

            if (connection == null){
                System.err.println(this.getClass().getName() +
                        "Error: unable to create Statement object.");
                System.exit(-1);
            }

            statement = connection.createStatement(
                    ResultSet.TYPE_SCROLL_INSENSITIVE,
                    ResultSet.CONCUR_READ_ONLY);

        } catch (SQLException sqle) {
            System.err.println("Could not create statement");
            sqle.printStackTrace();
        }

        return statement;
    }
```

Fig. A.3. Separating the database connection task from the query construction task as identified by the tool.



```
/**
package org.ualberta.serl.sim.simplayer;

import java.awt.BorderLayout;

/**
 * @author msmit
 *
 */
public class PlaybackControl extends JFrame implements ChangeListener {
    JSlider speedSlider = null;
    JSlider currentSlider = null;
    private void initComponents(int max) {
        /**
         * The number of milliseconds of simulation time that pass
         * per second of real time.  For example, if you want to take 40 ms
         * to simulate every real second, set this to 40.
         */
        this. setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        speedSlider = new JSlider(SwingConstants.HORIZONTAL, 0, 1000, 40);
        speedSlider.setLabelTable(speedSlider.createStandardLabels(100));
        speedSlider.setMajorTickSpacing(25);
        speedSlider.setPaintLabels(true);
        speedSlider.setPaintTicks(true);
        speedSlider.setMinimumSize(new Dimension(500,100));
        speedSlider.addChangeListener(this);


        // this shows the current point in the simulation.  it always starts at 0,
        // with a min of 0 and a max of the maximum time achieved during the
        // original simulation.  now we caclulate interval:
        int spacing = (max / 100) * 10;
        currentSlider = new JSlider(SwingConstants.HORIZONTAL, 0, max, 0);
        currentSlider.setLabelTable(currentSlider.createStandardLabels(spacing));
        currentSlider.setMajorTickSpacing(spacing / 4);
        currentSlider.setPaintLabels(true);
        currentSlider.setPaintTicks(true);
        currentSlider.setMinimumSize(new Dimension(500,100));
        currentSlider.addChangeListener(this);

        JPanel sliderPanel = new JPanel();
        sliderPanel.setLayout(new BorderLayout());
        sliderPanel.add(new JLabel("Current point in simulation"), BorderLayout.NOR
        sliderPanel.add(currentSlider, BorderLayout.SOUTH);


        JPanel sliderPanel2 = new JPanel();
        sliderPanel2.setLayout(new BorderLayout());
        sliderPanel2.add(new JLabel("Speed of simulation"), BorderLayout.NORTH);
        sliderPanel2.add(speedSlider, BorderLayout.SOUTH);



        this.setLayout(new BorderLayout());
        this.getContentPane().add(sliderPanel, BorderLayout.NORTH);
        this.getContentPane().add(sliderPanel2, BorderLayout.SOUTH);


        //this.pack();
        this.setSize(500,160);
        this.setVisible(true);


    }
```

Fig. A.4. Separating view from control as suggested by the tool.



```
private JFreeChart generateCharts() {
    pieDataset = new DefaultPieDataset();
    State values[] = State.values();
    for (int i = 0; i < counts.length; i++) {
        names[i] = values[i].toString();
        pieDataset.insertValue(i, names[i], counts[i]);
    }
    JFreeChart chart = ChartFactory.createPieChart("Endpoint States",
            pieDataset, true, false, false);
    PiePlot plot = (PiePlot) chart.getPlot();
    Color[] colors = { Color.black, Color.blue, Color.yellow, Color.pink,
            Color.orange, Color.red, Color.green, Color.cyan, Color.white};
    for (int i = 0; i < counts.length; i++) {
        plot.setSectionPaint(names[i], colors[i]);
    }
    return chart;
}

private JFreeChart generateGraph(int maxVal) {
    series = new XYSeries("Depot Usage");
    XYDataset xyDataset = new XYSeriesCollection(series);
    JFreeChart chart = ChartFactory.createXYLineChart(
            "Depot Usage over Time", "# transfers", "time", xyDataset,
            PlotOrientation.HORIZONTAL, true, false, false);
    final XYPlot plot = chart.getXYPlot();
    ValueAxis axis = plot.getRangeAxis();
    axis.setRange(0, maxVal);
    axis = plot.getDomainAxis();
    axis.setRange(0, 60);
    return chart;
}
```

Fig. A.5. Semantic measures erroneously grouping together two different tasks due to the common use of terms *chart*, *plot* and *generate*.

References

Anquetil, N., Lethbridge, T., 1999. Experiments with clustering as a software remodularization method. In: 6th Working Conference on Reverse Engineering.

Bavota, G., De Lucia, A., Marcus, A., Oliveto, R.,2010. A two-step technique for extract class refactoring. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering. ACM, New York, NY, USA, pp. 151–154.

Bavota, G., Oliveto, R., De Lucia, A., Antoniol, G., Guéhéneuc, Y.G., 2010. Playing with refactoring: identifying extract class opportunities through game theory. In: Early Research Achievement Track of the 26th IEEE International Conference on Software Maintenance (ICSM'2010), Timisoara, Romania.

Bavota, G., De Lucia, A., Oliveto, R., 2011. Identifying extract class refactoring opportunities using structural and semantic cohesion measures. Journal of Systems and Software 84, 397–414.

Briand, L.C., Daly, J.W., Wüster, J., 1998. A unified framework for cohesion measurement in object-oriented systems. Empirical Software Engineering 3, 65–117.

Chatzigeorgiou, A., 2003. Mathematical assessment of object-oriented design quality. IEEE Transactions on Software Engineering 29, 1050–1053.

Chatzigeorgiou, A., Xanthos, S., Stephanides, G.,2004. Evaluating object-oriented designs with link analysis. In: Proceedings of the 26th International Conference on Software Engineering. IEEE Computer Society, Washington, DC, USA, pp. 656–665.

Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C., 2001. Introduction to Algorithms. The MIT Press, New York.

Daskalakis, C., Goldberg, P.W., Papadimitriou, C.H., 2009. The complexity of computing a Nash equilibrium. Communications of the ACM 52, 89–97.

De Lucia, A., Oliveto, R., Vorraro, L., 2008. Using structural and semantic metrics to improve class cohesion. In: 24th IEEE International Conference on Software Maintenance, Beijing, China.

Demeyer, S., Ducasse, S., Nierstrasz, O.M., 2002. Object-oriented Reengineering Patterns. Morgan Kaufman Publishers.

van Deursen, A., Kuipers, T., 1999. Identifying objects using cluster and concept analysis. In: 21st International Conference Software Engineering, pp. 246–255.

Deursen, A.v., Kuipers, T.,2003. Source-based software risk assessment. In: ICSM'03: Proceedings of the International Conference on Software Maintenance. IEEE Computer Society.

Doval, D., Mancoridis, S., Mitchell, B.S., 1999. Automatic clustering of software systems using a genetic algorithm. In: 5th International Conference on Software Tools and Engineering Practice, Pittsburgh, PA.

DuBois, B., Demeyer, S., Verelst, J.,2004. Refactoring – improving coupling and cohesion of existing code. In: 11th Working Conference on Reverse Engineering. Delft University of Technology, The Netherlands, pp. 144–151.

Ester, M., Kriegel, H.P., Sander, J., Xu, X., 1996. A density-based algorithm for discovering clusters in large spatial database with noise. In: International Conference on Knowledge Discovery in Databases and Data Mining, Portland, OR.

Fokaefs, M., Tsantalis, N., Stroulia, E., Chatzigeorgiou, A., 2009. Decomposing object-oriented class modules using an agglomerative clustering technique. In: 25th IEEE International Conference on Software Maintenance (ICSM'2009), Edmonton, AB, Canada.

Fokaefs, M., Tsantalis, N., Stroulia, E., Chatzigeorgiou, A., 2011. JDeodorant: identification and application of extract class refactorings. In: Proceedings of the 33rd International Conference on Software Engineering.

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

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley, Boston, MA.

Holzrichter, M., Oliveira, S., 1999. A graph based method for generating the Fiedler vector of irregular problems. In: IPPS/SPDP Workshops.

Jolliffe, I., 1986. Principal Component Analysis. Springer Verlag.

Joshi, P., Joshi, R.K., 2009. Concept analysis for class cohesion. In: 13th European Conference on Software Maintenance and Reengineering, Kaiserslautern, Germany, pp. 237–240.

Khomh, F., Vaucher, S., Guéhéneuc, Y.G., Sahraoui, H., 2009. A Bayesian approach for the detection of code and design smells. In: Proceedings of the 2009 Ninth International Conference on Quality Software, Washington, DC, USA. IEEE Computer Society, pp. 305–314.

Kuipers, T., Visser, J.,2004. A tool-based methodology for software portfolio monitoring. In: Software Audit and Metrics, Proceedings of the 1st International Workshop on Software Audit and Metrics. INSTICC Press, pp. 118–128.

Li, W., Henry, S., 1993. Object-oriented metrics that predict maintainability. Journal of Systems and Software 23, 111–122.

Maletic, J.I., Marcus, A.,2001. Supporting program comprehension using semantic and structural information. In: Proceedings of the 23rd International Conference on Software Engineering. IEEE Computer Society, Washington, DC, USA, pp. 103–112.

Mancoridis, S., Mitchell, B.S., Rorres, C., Chen, Y., Gansner, E.R.,1998. Using automatic clustering to produce high-level system organizations of source code. In: 6th International Workshop on Program Comprehension. IEEE Computer Society Press, pp. 45–52.

Marcus, A., Poshyvanyk, D.,2005. The conceptual cohesion of classes. In: Proceedings of the 21st IEEE International Conference on Software Maintenance. IEEE Computer Society, Washington, DC, USA, pp. 133–142.

Marinescu, R.,2004. Detection strategies: metrics-based rules for detecting design flaws. In: Proceedings of the 20th IEEE International Conference on Software Maintenance. IEEE Computer Society, Washington, DC, USA, pp. 350–359.

Martin, R.C., 2003. Agile Software Development: Principles, Patterns and Practices. Prentice Hall, Upper Saddle River, NJ.

Moha, N., Gueheneuc, Y.G., Duchien, L., Meur, A.F.L., 2010. Decor: a method for the specification and detection of code and design smells. IEEE Transactions on Software Engineering 36, 20–36.

Munro, M.J.,2005. Product metrics for automatic identification of "bad smell" design problems in java source-code. In: Proceedings of the 11th IEEE International Software Metrics Symposium. IEEE Computer Society, Washington, DC, USA, p. 15.

Opdyke, W.F., 1992. Refactoring object-oriented frameworks. Ph.D. Dissertation.

Pontryagin, L., Arkhangel'skii, A., 1990. General Topology I: Basic Concepts and Constructions, Dimension Theory. Springer, Heidelberg.

Sartipi, K., Kontogiannis, K., 2001. Component clustering based on maximal association. In: Proceedings of the IEEE Working Conference on Reverse Engineering, Stuttgart, Germany.

Schäfer, M., Dolby, J., Sridharan, M., Torlak, E., Tip, F.,2010. Correct refactoring of concurrent java code. In: Proceedings of the 24th European Conference on Object-oriented Programming. Springer-Verlag, Berlin, Heidelberg, pp. 225–249.

Shokoufandeh, A., Mancoridis, S., Denton, T., Maycock, M., 2005. Spectral and meta-heuristic algorithms for software clustering. Journal of Systems and Software 77, 213–223.

Simon, F., Steinbruckner, F., Lewrentz, C., 2001. Metrics based refactoring. In: 5th European Conference on Software Maintenance and Reengineering, Lisbon, Portugal, pp. 30–38.

Tahvildari, L., Kontogiannis, K., 2003. A metric-based approach to enhance design quality through meta-pattern transformations. In: 7th European Conference on Software Maintenance and Reengineering, Benevento, Italy, pp. 183–192.

Tan, P.N., Steinbach, M., Kumar, V., 2005. Introduction to Data Mining. Addison-Wesley.

Tarjan, R.E., 1972. Depth-first search and linear graph algorithms. SIAM Journal on Computing, 146–160.

Trifu, A., Marinescu, R., 2005. Diagnosing design problems in object oriented systems. In: 12th Working Conference on Reverse Engineering.

Tsantalis, N., Chatzigeorgiou, A., 2009. Identification of move method refactoring opportunities. IEEE Transactions on Software Engineering 35, 347–367.

Tzerpos, V., Holt, R.C., 1998. Software botryology: automatic clustering of software systems. In: International Workshop on Large-scale Software Composition.

Van Emden, E., Moonen, L.,2002. Java quality assurance by detecting code smells. In: Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02). IEEE Computer Society, Washington, DC, USA, pp. 97–106.

Vaucher, S., Khomh, F., Moha, N., Gueheneuc, Y.G., 2009. Tracking design smells: lessons from a study of god classes. In: Working Conference on Reverse Engineering, pp. 145–154.

Wiggerts, T.A., 1997. Using clustering algorithms in legacy systems remodularization. In: 4th Working Conference on Reverse Engineering.

Xanthos, S., 2006. Clustering Object-oriented Software Systems using Spectral Graph Partitioning. ACM Student Research Competition.

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

Marios Fokaefs is a PhD candidate in the Department of Computing Science at the University of Alberta, Greece. He received his BSc from the Department of Applied Informatics at the University of Macedonia, Greece in 2008 and his MSc from the Department of Computing Science at the University of Alberta, Canada in 2010. His research interests include object-oriented and service-oriented design and maintenance. Hi is a member of the IEEE.

Nikolaos Tsantalis received the BS, MS and PhD degrees in applied informatics from the University of Macedonia, Greece, in 2004, 2006 and 2010, respectively. He is currently a Postdoctoral Fellow at the Department of Computing Science, University of Alberta, Canada. His research interests include design pattern detection, identification of refactoring opportunities, and design evolution analysis. He is a member of the IEEE and the IEEE Computer Society.

Eleni Stroulia is a Professor and NSERC/iCORE Industrial Research Chair on Service Systems Management (w. support from IBM) with the Department of Computing Science at the University of Alberta, Canada. She holds M.Sc. and Ph.D. degrees from Georgia Institute of Technology. Her research addresses industrially relevant software-engineering problems with automated methods, based on artificial-intelligence techniques. She is a member of ACM, and IEEE.

Alexander Chatzigeorgiou is an assistant professor of software engineering in the Department of Applied Informatics at the University of Macedonia, Thessaloniki, Greece. He received the Diploma in electrical engineering and the PhD degree in computer science from the Aristotle University of Thessaloniki, Greece, in 1996 and 2000, respectively. From 1997 to 1999, he was with Intracom, Greece, as a telecommunications software designer. His research interests include object-oriented design, software maintenance and evolution. He is a member of the IEEE.