Recommending Refactorings based on Team Co-Maintenance Patterns

Gabriele Bavota[1], Sebastiano Panichella[1], Nikolaos Tsantalis[2], Massimiliano Di Penta[1], Rocco Oliveto[3], Gerardo Canfora[1]
[1]Department of Engineering, University of Sannio, Benevento, Italy
[2]Department of Computer Science & Software Engineering, Concordia University, Canada
[3]Department of Bioscience and Territory, University of Molise, Pesche (IS), Italy
gbavota@unisannio.it, spanichella@unisannio.it, tsantalis@cse.concordia.ca, dipenta@unisannio.it, rocco.oliveto@unimol.it, canfora@unisannio.it

ABSTRACT

Refactoring aims at restructuring existing source code when undisciplined development activities have deteriorated its comprehensibility and maintainability. There exist various approaches for suggesting refactoring opportunities, based on different sources of information, e.g., structural, semantic, and historical. In this paper we claim that an additional source of information for identifying refactoring opportunities, sometimes orthogonal to the ones mentioned above, is team development activity. When the activity of a team working on common modules is not aligned with the current design structure of a system, it would be possible to recommend appropriate refactoring operations—e.g., extract class/method/package—to adjust the design according to the teams' activity patterns. Results of a preliminary study—conducted in the context of extract class refactoring—show the feasibility of the approach, and also suggest that this new refactoring dimension can be complemented with others to build better refactoring recommendation tools.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Documentation, Enhancement, Restructuring, Reverse Engineering, and Reengineering*

Keywords

Refactoring; Developers; Teams

1. INTRODUCTION

Software refactoring is *"a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior"* [7]. Refactoring is usually a direct consequence of an undisciplined evolution, due to the lack of a rigorous and documented development process, poor design decisions, or simply to the need for applying urgent patches to the software.

ASE'14, September 15-19, 2014, Vasteras, Sweden.

http://dx.doi.org/10.1145/2642937.2642948.

In recent and past years, different approaches and tools have been developed to identify refactoring opportunities. Some of them [14] use structural information to identify the need for refactoring, and then suggest a suitable refactoring action. Others also exploit semantic information (i.e., textual information extracted from source code indicating the implementation of similar responsibilities) [2], or historical data [11] to identify refactoring solutions.

We conjecture that a fourth dimension can be highly beneficial when recommending refactoring operations. Such a dimension relates to *changes performed by teams* and it is based on the assumption of congruence between social and technical activities [4]. We define a team as a group of developers that, during the project history, worked on common sets of source code entities (note that in this definition we do not necessarily assume that such developers communicate with each other). The basic idea is that *code entities frequently modified by the same team should be grouped together in a separate module.* A straight-forward consequence of that refactoring would be easing the release of individual components of the project and/or integration activities, without waiting for other teams to complete their tasks.

Suppose that two different teams T_i and T_j usually work on different code entities, and thus they are responsible for maintaining and evolving different features of the system. However, both T_i and T_j work on class C, but on different parts of it, e.g., T_i works on members C_{m_i}, while T_j on members C_{m_j}, where $C_{m_i} \cap C_{m_j} = \varnothing$. This could be a symptom of heterogeneous responsibilities implemented in C, and thus of an opportunity to perform Extract Class refactoring. We claim that a better source code organization can be obtained by separating C_{m_i} from C_{m_j}, because these distinct member groups of C can be likely related to the features being maintained by T_i and T_j, respectively. Note that, as it will be clearer later in the paper, relying solely in change history is insufficient. This is because history would just identify co-changes, while in this context we need to identify code entities co-maintained by the same team members.

In this paper we (i) introduce the approach named **T**eam **B**ased **R**efactoring (TBR), to identify refactoring opportunities based on team co-maintenance patterns, (ii) instantiate TBR to support Extract Class refactoring, and (iii) provide preliminary results showing that TBR is able to identify meaningful refactoring solutions and complementary in many cases to other sources of information. This opens the road towards better refactoring recommendation tools being able to provide more accurate and/or more complete suggestions by combining multiple sources of information.

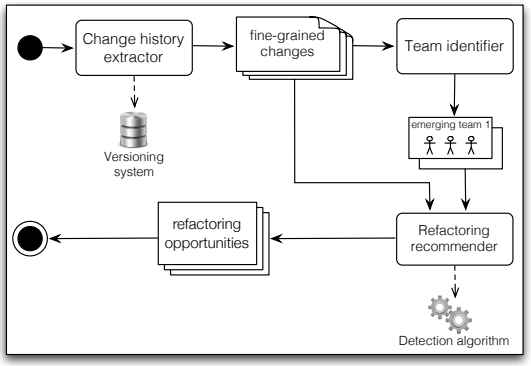

Figure 1: TBR: Team Based Refactoring

2. APPROACH

Figure 1 overviews the main steps of TBR. First, the entire change history of a system under analysis is extracted by mining the versioning system through a component called *Change history extractor*. This information is then provided as an input to the *Team identifier* component, in charge of detecting teams as groups of developers usually working on the same code entities. Finally, the change history information and the identified teams—together with a specific *detection algorithm* for a particular refactoring operation—are provided to the *Refactoring recommender*, producing a list of refactoring recommendations for the operation of interest. In the following we detail each of the main steps described above.

2.1 Extracting Change History Information

The *Change history extractor* mines the versioning system log, identifying changes occurred on the system files over time, as well as the authors of each change. Note that, when the mining is performed from git (as in our study), it allows to distinguish, when this information is available, actual authors of a change from committers. The logs extracted through this operation report code changes at file level of granularity. While this information is sufficient to identify teams as groups of developers usually working on the same code files, it is not enough for the *Refactoring recommender*. In fact, to detect solutions for several refactoring operations the *Refactoring recommender* needs to know the changes performed by developers in the code at a finer granularity level. To extract this information, a code analyzer developed in the context of the *Markos* European project[1] is used. The code analyzer parses the source code by relying on the *srcML* toolkit [5] and extracts a set of facts concerning the files, classes, methods, and attributes that have been added, removed, and changed in each commit. As it will be clear soon, this information can be used by the *Refactoring recommender* to support different refactoring operations.

2.2 Identifying Teams

TBR identifies teams as groups of developers usually working on the same set of code files. Once again, we remark that this may not correspond to actual teams of people structured by an organization/company; rather, it is a mean to identify groups of people co-changing sets of code elements. In accordance with this definition, there is a link between two developers if they modify the same file during a specific time interval. The strength of a link between two developers depends on the number of times the developers modify the same file(s) in the considered time period. To identify teams during the history of a project, the *Team identifier* splits the history of the system into time windows, identifying teams in each of them. This step is needed for two reasons:

1. Teams change during time. Thus, computing the teams by considering the entire change history would not make sense. A software engineer interested in identifying refactoring solutions on the current version of the system will focus her attention just on the last year(s) of development, since very old teams are unlikely to exist anymore.
2. To consider two developers as part of the same team, we need to ensure that they work on the same files in a specific (and limited) period of time. Otherwise, it could happen to consider two developers as part of the same team even if they modified the same file in two distant time periods.

Choosing an appropriate time window length is important. A longer time window captures more commits than a shorten one, hence more information to cluster developers into teams. However, a long time window also implies the risk of grouping people not really working on the file in the same time period, but rather in two subsequent time periods. The choice of the time window length depends on factors that are very project specific, like, for instance, the *frequency of commits* (the higher the commit frequency, the shorter the time window could be), the *average release interval* (for projects issuing releases very quickly a shorter time window could be appropriate), and the *stability of the developers' base* (the more stable is the developers' base, the longer the time window could be).

For each time window, the Ward's hierarchical clustering algorithm [10] is applied to cluster developers into teams. As a distance between the entities to cluster (i.e., developers), we adopt an Euclidean distance based on a dissimilarity measure taking into account the common files two developers modify in the considered time window. In particular, given the generic pair of developers (D_i,D_j), their dissimilarity d is computed as:

$$d(D_i, D_j) = 1 - \frac{|commonlyModifiedFiles| - min}{max - min}$$

where $|commonlyModifiedFiles|$ is the number of common files modified by D_i and D_j in the considered time window, and min (max) is the minimum (maximum) value of $|commonlyModifiedFiles|$ measured among all pairs of developers. As it can be noticed (i) d is normalized between zero and one and (ii) in d, the higher the number of common files two developers modify in the considered time window, the lower their dissimilarity.

The output of the Ward's algorithm is a dendrogram, a tree diagram where the leafs of the tree represent the entities to cluster (developers) while the remaining nodes represent possible clusters (teams) the entities belong to, up to the root representing a cluster containing all the entities. The distance between merged entities increases with the level of the merge (starting from the leaves towards the root). This means that nodes (i.e., clusters) at a higher level group together entities having higher distance (lower similarity) between them. To find the dendrogram cut-point

[1] `http://www.markosproject.eu`

we need to determine an appropriate number of clusters. To this aim, we relied on a widely adopted approach based on the Silhouette coefficient [8], a measure of the quality of the obtained clustering defined as the ratio between the average intra-clustering and the average inter-clustering distance. We compute the Silhouette coefficient for each possible partition obtainable from the dendrogram produced by the Ward's algorithm, selecting the one exhibiting the highest Silhouette value to cluster developers into teams.

2.3 Detecting Refactoring Opportunities

2.3.1 General Approach

When the *Refactoring recommender* is invoked, the following information is available for each time window (i) the teams existing in the time window, and (ii) the fine-grained changes performed by each team in the time window. By exploiting this information it is possible to support different refactoring operations. Indeed, as said before, the general idea behind TBR is that *code entities frequently modified by the same team should be grouped together in a separate module*. This idea can be instantiated to different refactoring operations by considering different granularity levels for *code entities* and *modules*.

Table 1 illustrates how TBR can be applied to different types of refactoring opportunities. For instance, by considering a group of classes maintained by a specific team as the *code entities* it is possible to improve the system package modularization to better reflect the way developers work; a typical refactoring in such a context is the Extract Package, aimed at isolating a specific responsibility into a package, representing our *module granularity*. Going at a finer level of granularity and considering methods as the *modules* of interest, Extract Method refactoring could be supported by two means: 1) a code fragment (i.e., a subset of statements) within a method maintained by a specific team of developers could indicate the existence of a distinct functionality that can be extracted in a separate method, 2) similar code fragments within different methods frequently modified by a specific team could indicate the existence of change-prone clones that can be merged into a single method. Additionally, the modification of the code fragments by the same team indicates that the developers are aware of the existence of the clones and dedicate effort to update them consistently.

According to the particular characteristics and relationships of the involved code entities, we can develop algorithms for extracting different types of refactoring opportunities. In the next subsection, we will describe a prototype approach for the detection of Extract Class refactoring opportunities using TBR.

2.3.2 TBR for Extract Class Refactoring

In this paper we instantiate TBR to Extract Class refactoring as described in Algorithm 1. Extract Class refactoring is a technique for splitting classes with many responsibilities into different classes. TBR identifies Extract Class solutions in a given time window TW. Note that, a software engineer interested in identifying refactoring solutions in a system S today, will look for teams and changes performed by them in the recent software history (of length TW_{length}).

TBR analyzes each class $C_i \in S$ to verify, for each team T_j existing in TW, if there exists a set of methods M_{T_j} owned by T_j (lines from 6 to 13 in Algorithm 1). M_{T_j} is

Algorithm 1 TBR applied to Extract Class refactoring

```
1: TW_length ←1 year
2: TW ← from today to TW_length before
3: Teams ← teams detected by the Teams identifier in TW
4: S ← the set of classes in the system under analysis
5: Refactorings ← ∅
6: for each class C_i ∈ S do
7:     for each team T_j ∈ Teams do
8:         M_Tj ← ∅                                  ▷ methods owned by T_j
9:         for each method m_k ∈ C_i do
10:            if T_j owns m_k then
11:                M_Tj ← m_k
12:            end if
13:        end for
14:        if |M_Tj| > λ and |C_i − M_Tj| > λ then
15:            A_Tj ← C_i attributes used more by M_Tj than by
   {C_i − M_Tj}
16:            RefToAdd ← extract M_Tj + A_Tj from C_i
17:            if satisfiesPreconditions(RefToAdd) then
18:                Refactorings ← RefToAdd
19:            end if
20:        end if
21:    end for
22: end for
23: return Refactorings
```

owned by T_j during TW if, for each method m_k in M_{T_j}, T_j is responsible of at least 75% of all changes performed on m_k during TW [3]. Our conjecture is that the set of methods M_{T_j} represents a precise responsibility, managed by T_j, that can be extracted from C_i to facilitate its maintenance.

In order to consider the extraction of M_{T_j} from C_i as a valid extract class refactoring opportunity we check that: (i) the cardinality of M_{T_j} is at least λ, where λ is fixed in our current implementation to three, otherwise it is unlikely that M_{T_j} represents a well-defined set of responsibilities, and (ii) the methods left in C_i when extracting M_{T_j} are more than λ (for the same reason explained above)—see lines from 14 to 16 in Algorithm 1. Note that our choice of fixing $\lambda = 3$ is not random, but based on previous work [2]. However, a deep investigation of such parameter is part of our future research agenda.

Attributes of C_i that are used by a greater proportion of methods in M_{T_j} than by the methods left in C_i are grouped in A_{T_j} to complete the extract class refactoring solution (lines from 15 to 16 in Algorithm 1). At the end of the process we examine the recommendations against the preconditions proposed by [6] to filter out those that could change program behavior or are not applicable in practice (lines from 17 to 19 in Algorithm 1).

Note that by using the above described algorithm it is also possible to identify different teams owning different subsets of C_i's methods. This would result in a whole new organization of the responsibilities implemented in C_i.

3. PRELIMINARY EVALUATION

The *goal* of our study is to evaluate the quality of the refactoring solutions identified by TBR and the complementarity of the information it exploits as compared to other sources of information typically used to support refactoring, as *structural*, *semantic*, and *historical* information. The *context* of the study consists of five software projects belonging to the Android APIs, chosen for being very active projects with a relatively large number of developers (see Table 2).

Table 1: Instantiation of TBR to Different Refactoring Operations

Code Entities	Module Granularity	Code Smell	Refactoring Operation
A subset of methods and attributes within the same class frequently modified by the same team	Class	Single Responsibility Principle violation	Extract Class
A method from one class (*source*) frequently co-modified with members of another class (*target*) by the same team	Class	Feature Envy	Move Method
A subset of statements within a method frequently modified by the same team	Method	Non-Cohesive Method	Extract Method
A set of similar code fragments within different methods frequently modified by the same team	Method	Duplicated Code	Extract Method
A subset of classes under the same package or different packages frequently modified by the same team	Package	Poor Package Organization	Extract Package

Table 2: Software systems used in the study.

Project from Andr. API	Period	KLOC
framework-opt-telephony	Aug 2011-Jan 2013	73-78
frameworks-base	Oct 2008-Jan 2013	534-1,043
frameworks-support	Feb 2011-Nov 2012	58-61
sdk	Oct 2008-Jan 2013	14-82
tool-base	Nov 2012-Jan 2013	80-134

Our study aims at addressing two research questions:

- **RQ$_1$:** *Is the information derived from teams useful to identify refactoring opportunities?*
- **RQ$_2$:** *Is the information derived from teams complementary to the sources of information typically exploited to identify refactoring opportunities?*

To answer **RQ$_1$** we simulate the use of TBR to detect refactoring opportunities. That is, given S_t the system snapshot at time t on which we are interested to perform refactoring, TBR identifies refactoring solutions by looking at changes occurred in the period $[t - TW_{length}, t]$. Releases of the Android APIs are generally issued every four/six months[2]. However, since Android APIs are divided into sub-projects (see Table 2), we observed that some of them did not change (or rarely changed) in a four-months time interval. Because of that, we considered possible values of TW_{length} between six months and one year, finding one year to be sufficient to capture enough changes for all subsystems, without risking to merge teams working in very different time periods. We plan to perform a thorough assessment of this parameter in the future with the aim of defining heuristics to automatically set the time window length based on specific characteristics of the project on which TBR is applied.

Then, to assess the quality of the recommended refactorings, two PhD students and one industrial developer (none of them are authors, nor they know how the approach works) evaluated each of them by answering the following questions:

- **Q1:** *How do you perceive the cohesiveness/decoupling of the classes involved in the refactoring?* Assign a score on a five points Likert scale: 1 (definitely worse), 2 (slightly worse), 3 (the same), 4 (slightly better), 5 (definitely better than before).
- **Q2:** *Evaluate the effort of implementing the refactoring operation by considering the impact on the source code (e.g., addition of getter/setter methods, update of references, etc).* Assign a score on a five points Likert scale: 1 (very low), 2 (low), 3 (medium), 4 (high), 5 (very high).

[2] `http://tinyurl.com/4an7xgg`

We also assess the evaluation agreement between the three participants by computing the Kendall's W coefficient of concordance [9], which ranges between 0 (no agreement) and 1 (complete agreement).

To answer **RQ$_2$**, we take each of the classes recommended by TBR in **RQ$_1$** and refactor them by applying approaches exploiting different sources of information. In particular, we apply the Extract Class refactoring approach by Bavota *et al.* [2]. This approach exploits a combination of structural (i.e., method calls and shared attributes) and semantic (i.e., textual similarity between methods) information to group together cohesive groups of methods and attributes to extract from a given class. We apply the approach by Bavota *et al.* [2] by forcing it to exploit only *structural* or only *semantic* information. This can be easily done by tuning the weight (importance) assigned to each source of information when identifying refactoring solutions (e.g., setting the weight of the semantic information to zero results in refactoring solutions identified only by exploiting structural information). From now on we refer to these two techniques simply as *structural* and *semantic*. Also, we refactor each class using *historical* information based on the approach proposed by Palomba *et al.* [12]. Specifically, we use association rule discovery [1] to detect subsets of methods in the same class that often change together (and thus can be extracted to a new class). From now on we refer to this approach simply as *historical*. Note that, to distribute attributes among classes, we use the same heuristics of TBR, i.e., we place each attribute in the class with the largest proportion of methods using it.

Then, we compute the percentage of classes recommended by TBR that have been also recommended by at least one of the other three techniques to assess the complementarity of the alternative techniques. For each class decomposed by both TBR and the competitive approaches, we compute the MoJo eFfectiveness Measure (*MoJoFM*) [15] between the decomposition suggested by TBR and the one suggested by the alternatives to evaluate their similarity. The MoJoFM is a normalized variant of the MoJo distance and is computed as follows:

$$MoJoFM(A, B) = 1 - \frac{mno(A, B)}{max(mno(\forall A, B))}$$

where $mno(A, B)$ is the minimum number of *Move* or *Join* operations to perform in order to transform the partition A into B, and $max(mno(\forall\ A, B)$ is the maximum possible distance of any partition A from partition B. Note that in our case, a partition represents the sets of members (methods and attributes) in which the class under investigation is decomposed applying a given technique. Thus, a

Table 3: Answers Provided by the Three Participants.

Q1: How do you perceive the cohesiveness/decoupling of the classes involved in the refactoring?					
Participant	*definitely worse*	*slightly worse*	*the same*	*slightly better*	*definitely better*
PhD_1	3 (13%)	3 (13%)	4 (17%)	8 (35%)	5 (22%)
PhD_2	4 (17%)	2 (9%)	6 (26%)	6 (26%)	5 (22%)
Industrial	4 (17%)	3 (13%)	5 (22%)	7 (30%)	4 (17%)
Q2: Evaluate the effort of implementing the refactoring by considering the impact on the source code					
Participant	*very low*	*low*	*medium*	*high*	*very high*
PhD_1	4 (17%)	6 (26%)	8 (35%)	3 (13%)	2 (9%)
PhD_2	4 (17%)	8 (35%)	6 (26%)	2 (9%)	3 (13%)
Industrial	2 (9%)	6 (26%)	9 (39%)	6 (26%)	2 (9%)

$MoJoFM$ value of 0 means that the decomposition solutions derived from two techniques have nothing in common, while a $MoJoFM$ value of 1 indicates an identical decomposition.

3.1 Results

The application of TBR to the five projects resulted in 23 Extract Class refactoring recommendations.

RQ1: ***Is the information derived from teams useful to identify refactoring opportunities?*** The answers provided by participants (from now on PhD_1, PhD_2, and *industrial*) to the two questions assessing the quality of the refactoring recommendations by TBR (i.e., **Q1** and **Q2**) are reported in Table 3. In particular, when answering **Q1**, participants found meaningful from 11 (47%, *industrial* and PhD_2) up to 13 (57%, PhD_1) of the 23 refactoring recommendations, assigning them a 5-score or a 4-score. Interestingly, the 11 recommendations evaluated as meaningful by *industrial* and PhD_2 are the same, and represent a subset of the 13 highly scored by PhD_1. Additional recommendations (on average 5, 22%) were assigned a 3-score, showing how the proposed refactoring represents an alternative to the original design but, from the participants' point of view, does not justify the need for a change. On average, only six out of the 23 suggestions (26%) were considered as bad refactoring recommendations, receiving a 2-score or a 1-score (see Table 3). When computing the Kendall's W coefficient to evaluate the level of agreement between the three participants in judging the 23 refactorings, we obtained a value equal to 0.897, indicating a very high level of concordance in the participants' evaluations (W is very close to 1).

Regarding the effort needed to implement the 23 recommended refactorings (**Q2**), the median rating assigned by the participants was 2 (low) for PhD_2, and 3 (medium) for PhD_1 and the *industrial* developer. On average, for six recommendations (26%) the evaluators felt that a *high* or a *very high* effort would be required to implement them. Also in this case the Kendall's coefficient showed a high level of agreement between participants, with W=0.791.

In summary, almost 50% of the refactoring recommendations by TBR have been highly appreciated by participants with only 26% of them, on average, negatively evaluated. Also, the effort needed to apply the suggested refactorings seems to be reasonable in most cases.

RQ2: ***Is the information derived from teams complementary to the sources of information typically exploited to identify refactoring opportunities?*** Table 4 shows (i) the percentage of the 23 classes recommended by TBR that have been also recommended by each of the three alternative techniques (i.e., *structural*, *semantic*, and *historical*) and (ii) the $MoJoFM$ between the refactorings suggested by TBR and those proposed by each of the other techniques. Also, the last row of Table 4 shows the percentage of the 23 classes that is recommended by at least one of the three competitive techniques.

Table 4: Overlap between the recommendations made by TBR and the other techniques.

Technique	Overlap	MoJoFM
structural	30%	0.77
semantic	43%	0.67
historical	35%	0.73
structural ∪ *semantic* ∪ *historical*	70%	-

As it can be easily noticed, the overlap between TBR and the other three techniques is present, but is not particularly high. In particular, the technique based on *structural* information recommends 30% of the classes for which TBR suggests a decomposition, *semantic* information 43%, and *historical* information 35%. Also, for 30% of the classes (i.e., seven classes) recommended by TBR, none of the three competitive techniques proposes an Extract Class refactoring solution. This means that the information extracted from teams is to some extent complementary to the *structural*, *semantic*, and *historical* information. In the evaluation conducted in the context of **RQ1**, the refactorings proposed by TBR on these seven classes were positively evaluated by all three participants in four cases (achieving a 4- or a 5-score), and with a median 2-score for what concerns the difficulty to apply them (**Q2**). Thus, even if only TBR recommends to refactor these classes, at least four out of the seven recommendations have been positively evaluated by participants.

When measuring the $MoJoFM$ between the refactorings suggested by TBR and the refactorings proposed by each of the other techniques (see Table 4) we obtained, an average value of 0.77 for the classes also recommended by the *structural*, 0.67 for the *semantic*, and 0.73 for the *historical* technique. Thus, the decompositions proposed by TBR have commonalities to those obtained by exploiting other sources of information, but also differences.

For instance, TBR, *structural*, and *semantic* techniques, consistently recommend an Extract Class refactoring for the class `BluethoothAdapter`, composed of 43 methods. This class, as stated in its Javadoc, represents the local device bluetooth adapter. It allows to perform fundamental bluetooth tasks, such as device discovery, query a list of paired devices, instantiate a `BluetoothDevice` using a known MAC address, and create a `BluetoothServerSocket` to listen for connection requests from other devices. TBR identified on this class a set of 14 methods owned by a team composed of three developers that should be extracted in a separate class. In particular, all 14 methods managed by this team are related to a precise responsibility implemented in the `BluethoothAdapter` class, i.e., creating a `BluetoothServerSocket` to listen for connection requests from other devices. Examples of these methods are `listenUsingRfcommOn`, `listenUsingRfcommWithServiceRecord`, and `createNewRfcommSocketAndRecord`. Also, TBR identified in `BluethoothAdapter` a set of 18 methods (owned by a team of 14 developers) mostly related to another responsibility of the class, i.e., de-

vice discovering. Examples of methods contained in it are `setScanMode`, `startDiscovery`, and `cancelDiscovery`.

The *structural* technique recommends, instead, the extraction of a single class that has a subset of the members of the first responsibility identified by TBR (i.e., the one managing the creation of a `BluetoothServerSocket`). In particular, it contains 5 of the 14 methods suggested to be extracted by TBR. While the remaining 9 methods are not structurally related to the 5 clustered together, they are still conceptually related to the management of a `BluetoothServerSocket`. For instance, the *structural* approach does not group together the methods `listenUsingRfcommWithServiceRecord` and `listenUsingEncryptedRfcommOn`.

Concerning the *semantic* technique, it recommends the extraction of two classes from `BluetoothServerSocket`: one is also in this case a subset of the first responsibility identified by TBR. In this case it contains 8 out of the 14 methods grouped together by TBR. The other class suggested to be extracted is composed of 10 methods, but from our manual inspection, it is very difficult to relate them to a precise responsibility implemented in the `BluethoothAdapter` class.

This example highlights that the four different sources of information, applied on the same class, can produce quite different and orthogonal results. Thus, refactoring recommendation tools could benefit from the combination of different sources of information. For instance, a "hybrid" Extract Class refactoring recommendation approach could be obtained in a similar fashion to what has been done in [2] to combine *structural* and *semantic* information. First, for each source of information a similarity measure between pairs of methods should be defined. Such a measure already exists for *structural* and *semantic* information [2] and can be easily defined for *historical* and *team-based* information. The underlying idea is that two methods are similar from the *historical* point of view if they often co-change during time, while from the *team* point of view they are similar if they are frequently maintained by the same team(s). Once defined, these four similarity measures can be combined in order to obtain an overall similarity between pairs of methods helping in identifying clusters of cohesive methods inside a class of interest. Of course, the weight of each of the four sources of information (i.e., of each of the four similarity methods) must be carefully assessed by (i) experimenting different configurations, (ii) exploiting machine learning techniques, or (iii) using search-based techniques to find the best configuration [13].

4. CONCLUSION AND FUTURE WORK

We proposed a novel refactoring approach named TBR (**T**eam **B**ased **R**efactoring), that exploits team co-maintenance patterns to recommend refactorings. It exploits the assumption that *code entities frequently modified by the same team should be grouped together in a separate module.*

In a preliminary evaluation conducted in the context of Extract Class refactoring we achieved results indicating that, overall, the recommended refactoring operations are meaningful and in 30% of the cases complementary to those suggested by alternative techniques. This is promising enough to conduct future work, focusing on (i) the instantiation of TBR to other refactoring operations, and (ii) the combination of information derived by teams with other sources of information typically used in the refactoring field.

Acknowledgment

Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, and Sebastiano Panichella are partially funded by the EU FP7-ICT-2011-8 project Markos, contract no. 317743. Nikolaos Tsantalis is partially funded by NSERC. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

5. REFERENCES

[1] R. Agrawal, T. Imielinski, and A. N. Swami. Mining association rules between sets of items in large databases. In *ACM DATA*, pages 207–216, 1993.

[2] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto. Automating extract class refactoring: an improved method and its evaluation. *Empirical Software Engineering*, Accepted on Apr 2013.

[3] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. T. Devanbu. Don't touch my code!: examining the effects of ownership on software quality. In *SIGSOFT/FSE'11*, pages 4–14, 2011.

[4] M. Cataldo, J. D. Herbsleb, and K. M. Carley. Socio-technical congruence: A framework for assessing the impact of technical and work dependencies on software development productivity. In *ESEM 2008*, pages 2–11. ACM.

[5] M. L. Collard, H. H. Kagdi, and J. I. Maletic. An XML-based lightweight C++ fact extractor. In *IWPC 2003*, pages 134–143. IEEE Computer Society.

[6] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou. Identification and application of extract class refactorings in object-oriented systems. *J. Syst. Softw.*, 85(10):2241–2260, Oct. 2012.

[7] M. Fowler. *Refactoring: improving the design of existing code.* Addison-Wesley, 1999.

[8] L. Kaufman and P. J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis.* Wiley-Interscience, 2005.

[9] M. G. Kendall and S. Babington. The problem of m rankings. *Annals of Mathematical Statistics*, pages 275–287, 1939.

[10] F. Murtagh and P. Legendre. Ward's hierarchical clustering method: Clustering criterion and agglomerative algorithm. *CoRR*, abs/1111.6285, 2011.

[11] A. Ouni, M. Kessentini, H. A. Sahraoui, and M. S. Hamdi. The use of development history in software refactoring using a multi-objective evolutionary algorithm. In *GECCO*, pages 1461–1468. ACM, 2013.

[12] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk. Detecting bad smells in source code using change history information. In *ASE*, 2013.

[13] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia. How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In *ICSE 2013*, pages 522–531, 2013.

[14] N. Tsantalis and A. Chatzigeorgiou. Identification of move method refactoring opportunities. *IEEE TSE*, 35(3):347–367, 2009.

[15] Z. Wen and V. Tzerpos. An effectiveness measure for software clustering algorithms. In *IWPC 2004*, pages 194–203.