Identifying Method Friendships to Remove the Feature Envy Bad Smell (NIER Track)

Rocco Oliveto
University of Molise, Pesche (IS), Italy
rocco.oliveto@unimol.it

Malcom Gethers
The College of William and Mary, Williamsburg, USA
mgethers@cs.wm.edu

Gabriele Bavota
University of Salerno, Fisciano (SA), Italy
gbavota@unisa.it

Denys Poshyvanyk
The College of William and Mary, Williamsburg, USA
denys@cs.wm.edu

Andrea De Lucia
University of Salerno, Fisciano (SA), Italy
adelucia@unisa.it

ABSTRACT

We propose a novel approach to identify *Move Method* refactoring opportunities and remove the *Feature Envy* bad smell from source code. The proposed approach analyzes both structural and conceptual relationships between methods and uses Relational Topic Models to identify sets of methods that share several responsabilities, i.e., "friend methods". The analysis of method friendships of a given method can be used to pinpoint the target class (envied class) where the method should be moved in. The results of a preliminary empirical evaluation indicate that the proposed approach provides meaningful refactoring opportunities.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*

General Terms

Design, Experimentation.

Keywords

Source Code Quality, Refactoring, Relational Topic Model.

1. INTRODUCTION

High levels of coupling and lack of cohesion are generally associated with lower productivity, greater rework, more significant design efforts by developers, and higher defect rates [11]. Consequently, low coupling and high cohesion can be regarded as indicators of good design quality in terms of maintenance. Monitoring coupling and cohesion of classes is particularly valuable during software evolution, where the classes of a system undergo continuous modifications making the source code more complex and drifting away from its original design. Indeed, due to strict deadlines, programmers do not always have sufficient time to make sure that the resulting source code conforms to OOP guidelines [9].

Feature Envy bad smell can be considered as the most common symptom related to problems with class coupling and cohesion. Such a bad code smell arises when a software engineer violates the principle of grouping behavior with related data and occurs when *"a method is more interested in a class other than the one it actually is in"* [9]. Move method refactoring (a method is moved to the class that it envies) represents a way to remove the Feature Envy bad smell and improve the overall system quality. Unfortunately, the identification of both the envied class and the method that has to be moved to the target class is not always trivial [20].

In this paper, we propose a novel approach for identifying move method refactoring opportunities. The proposed approach, called MethodBook, follows the Facebook[1] metaphor. Facebook is a well-known social networking portal, where users can add people as friends, send them messages, and update their personal profiles to notify friends about themselves. The personal profile plays a crucial role. In particular, Facebook uses Relational Topic Model (RTM) [5] to analyze users' profiles and suggest new friends or groups of people sharing similar interests. In our implementation of MethodBook, methods and classes play the same role as people and groups of people, respectively, in Facebook; methods' bodies, that is profiles, contain information about structural (e.g., method calls) and conceptual relationships (e.g., similar comments) with other methods in the same class and in the other classes. Having such a metaphorical representation, we can employ the same technique successfully utilized in Facebook, i.e., RTM, to identify "friends" of a method in order to suggest move method refactoring opportunities in software. In particular, given a method, we can apply RTM to suggest as a target class (i.e., group of methods), the class that contains the higher number of "friends" of the method under analysis.

[1]http://www.facebook.com/

The usefulness of MethodBook has been empirically evaluated in a preliminary case study conducted on ArgoUML[2]. The results are encouraging and highlighted that MethodBook is able to suggest meaningful move method refactorings.

The paper is organized as follows. Section 2 introduces the proposed approach, while Section 3 reports and discusses results from a preliminary empirical evaluation. After an overview of existing approaches to support refactoring (Section 4), Section 5 concludes the paper and overviews challenges and potential applications of the proposed approach.

2. METHODBOOK IN A NUTSHELL

The concept of *method friendships*[3] can be used as a metaphor to explain our idea of identifying envied classes. In the context of our approach two methods are considered to be friends if they share responsibilities, i.e., they operate on the same data structures or are related to the same features or concepts in the program. Such a definition suggests that methods that are good friends should be in the same class, since *"a class should be a crisp abstraction, handle a few clear responsibilities, or some similar guideline"* [9]. Based on this definition, if the "best" friends of a method m implemented in C_m are in a class C_f, then m shares more responsibilities with the methods of class C_f than with those in C_m. We conjecture that such a scenario implies the presence of a Feature Envy bad smell with the class C_f as an envied class.

2.1 Identify Method Friendships

The *method friendships* are identified using RTM through the analysis of structural and conceptual relationships among methods as well as the original structure of the classes. In particular, RTM analyzes the content of methods[4] and defines a model capable of comparing methods on the basis of conceptual topics that they share. Semantic overlap between methods serves as a good indication of method friendships. However, we also supply two different types of structural information to RTM, i.e., class composition data and structural relationships between methods derived using similarity measures. The former is represented by class composition, where methods implemented in the same class are considered to be friends. The latter type of structural information is derived by combining two different structural measures, namely, Structural Similarity between Methods (SSM) [10] and Call-based Interaction between Methods (CIM) [2], previously used to compute similarities between methods for identifying Extract Class refactoring opportunities [2]. These measures do not correlate and capture two distinct aspects of method relationships [2]. SSM captures attribute references in methods, i.e., the higher the number of instance variables that two methods share, the higher the similarity between the two methods. Instead, CIM takes into account calls performed by the methods. It is important to note that using such measures it is also possible to identify spurious (light) structural relationships between methods [2]. In order to identify the most important relationships (actual friendships) between methods, we filter all the relationships having near zero values, i.e., values lower than 0.1 [2].

The set of friendships derived by analyzing structural similarity between methods are supplied as existing links to RTM. It should be noted that when the method provided as input to MethodBook is implemented in the "wrong" class, there is a risk of supplying "inconsistent" information to RTM. In particular, by analyzing class composition we can supply RTM with a list of friends of the method in question that are actually not friends. Since we do not know *a priori* if the input method is in the correct class or not, we opted for leaving out relationships of the method under evaluation with other methods from the same class identified via class composition, as supplied to RTM. However, we opted for keeping the links between the method under evaluation and other methods in the same class as identified by structural similarity metrics.

The set of friendships provided to RTM are used to enrich the model for identifying friendships between methods that are not only based on conceptual overlap, but also on structural relationships derived from structural similarity measures and class composition. In addition, the structural and conceptual relationships between methods are based on factors that might affect class cohesion and coupling. Thus, implementing methods with high structural and conceptual similarity, i.e., friend methods, in the same class should improve overall quality of a software system.

2.2 Identify the Envied Class

The model built by RTM is used to determine degree of similarity among methods in the system and rank friendships among these methods. Thus, a cut point is then used to identify the μ best friends of (the methods having the highest similarity with) the method under analysis (in our evaluation we set $\mu = 5$). Once the "best" friends of a given method are identified, MethodBook analyzes the classes where these methods are implemented aiming at identifying the envied class. The suggested envied class is the class containing the highest number of identified friend methods. This choice is justified by our conjecture that the higher the number of friends in a class, the higher the quality of the class in terms of cohesion and/or coupling. Note that if two or more classes contain identical number of friend methods, the envied class is the class that contains the highest ranked best friend methods.

It is worth noting cases where the identification of the envied class is trivial, i.e., there is a class containing a sensibly higher number of friend methods as compared to other classes. However, there might also be cases where the envied class is difficult to identify, i.e., there are two or more classes that contain a comparable number of friend methods. To provide further support to software engineers, the suggestion of the envied class is supplied with a confidence level that indicates the reliability of the proposed refactoring. The confidence level uses the concept of information entropy which measures the amount of uncertainty of a discrete random variable [7]. In particular, we consider the suggestion of the envied class as a random variable, where the probability of its states is given by the distribution of the friend methods over suggested classes. We compute the confidence level as the entropy of the suggestion of the envied class. That is, the more scattered the friend methods among the classes are, the higher the entropy of the sugges-

[2]http://argouml.tigris.org/.

[3]Here the concept of method friendship is different from the concept of friend classes/methods of C++.

[4]A content of a method is represented by identifiers (e.g., variable and parameters names) and comments.

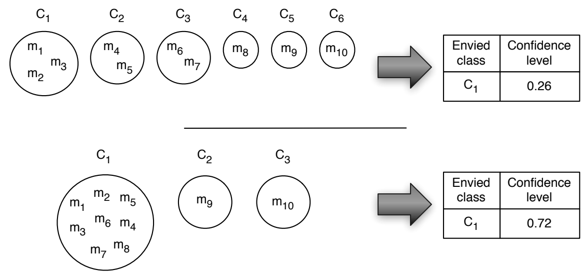

Figure 1: Two examples of envied class identification with different confidence levels.

tion of the envied class, i.e., it is more difficult to identify the envied class. On the contrary, if nearly all the friend methods are implemented in a single class, the entropy of the suggestion is low.

Figure 1 shows two examples of identifying envied class with different confidence levels. In both scenarios the number of best friends identified is ten. In the first case, the friend methods are scattered across several classes. In particular, the highest number of friend methods (three) belong to class C_1, while C_2 and C_3 contain two friend methods and the other three methods are distributed across classes C_4, C_5, and C_6. In this case the envied class is C_1 with a confidence level being relatively low, i.e., 0.26. The situation is quite different in the second example, where there is a class C_1 that contains a higher number of friend methods as compared to the other classes. In this case the confidence level is higher (0.72) indicating a better recommendation reliability as compared to the prior scenario.

3. PRELIMINARY EVALUATION

A preliminary evaluation of our approach was performed on a well-designed open-source system, namely ArgoUML version 0.16. This system is characterized by 1,071 classes and an overall number of methods equal to 9,926. The classes of the system have generally a good average quality in terms of cohesion (LCOM2[5] = 23.40 and C3[6] = 0.58) and coupling (CBO[7] = 28.49).

The evaluation aimed at investigating whether MethodBook is able to identify meaningful move method refactoring operations analyzing a given input method. Moreover, we were also interested in verifying whether the proposed approach for computing the confidence level serves as a good estimation of the recommendation reliability provided by MethodBook. The evaluation planning is inspired by mutation testing. In particular, we randomly extracted 1,000 methods from the classes of ArgoUML. The extracted methods were removed from the original classes. Then, we applied MethodBook to identify the envied class for each of the extracted methods. The original classes, where the extracted methods are implemented, were used as the oracle to evaluate MethodBook's recommendations for move method refactoring. Since the system used in our experimentation has a good design, we assume that the ideal outcome is when the envied class coincides with the original class. Thus, the suggested envied classes are compared against original classes to evaluate the accuracy of MethodBook, in terms of a number of envied classes correctly identified.

[5]Lack of Cohesion of Methods [6].
[6]Conceptual Cohesion of Classes [11].
[7]Coupling Between Object classes [6].

Table 1: MethodBook performance on ArgoUML.

Confidence level	Recall	Precision	Confidence level	Recall	Precision
0.0	0.71	0.71	0.6	0.49	0.89
0.1	0.71	0.75	0.7	0.39	0.98
0.2	0.66	0.79	0.8	0.39	0.98
0.3	0.66	0.79	0.9	0.39	0.98
0.4	0.64	0.82	1.0	0.39	0.98
0.5	0.58	0.85	-	-	-

The accuracy of MethodBook is analyzed considering different confidence levels (from 0 to 1 with a step of 0.1). Indeed, we fixed a threshold for the confidence level and analyzed the accuracy of the proposed envied classes having a confidence level higher than the fixed threshold. The accuracy of the MethodBook can be evaluated using two well-known Information Retrieval (IR) metrics, namely recall and precision [1].

Table 1 reports the results achieved using various thresholds for the confidence level. The analysis of the results shows quite encouraging results. In particular, MethodBook is capable of correctly identifying 40% of envied classes with precision of 95%. Additionally, 75% recall is achieved while precision is at 70%. In addition, considering different confidence levels also allowed us to verify whether confidence level correlates with the quality of MethodBook recommendations. In particular, we computed correlation between confidence level and MethodBook's precision and obtained a very high positive correlation, i.e., 0.97. Such a result suggests that the confidence level provides a good indication of the reliability of the recommendations generated by MethodBook, i.e., the higher the confidence level the higher the accuracy of MethodBook.

4. RELATED WORK

A lot of effort has been devoted to the definition of approaches to support the software engineer during refactoring. Existing approaches can be roughly classified into two different categories. Approaches of the first category aim at identifying source code components that might need to be refactored. Object-oriented metrics have been widely used to support the software engineer in the identification of source code components that needs refactoring [8, 16, 18]. In addition, correlation between structural anomalies, i.e., different type of code smells that often occur together, and other structural and semantic information has also been exploited to build a pattern-like mapping of design problems to the adequate treatments [19].

The approaches that fall into the second category automatically or semi-automatically perform refactoring operations to improve the overall quality of a system. Maruyama *et al.* [12] present a mechanism that automatically refactors methods in object-oriented frameworks to improve their reusability. In [4, 17] the authors propose algorithms to restructure class hierarchies to maximize abstraction, while Moore [13] proposes a method where existing classes with a low quality are replaced with a new set of classes where their methods are optimally factored to minimize meaningful code duplication. In [2, 3] the authors proposed two approaches for Extract Class refactoring that exploit graph theory to identify sets of (structural and conceptual) strongly related methods in a class to be refactored. The refactoring of a software system has also been formulated as a search problem in the space of alternative designs [14, 15].

Recently, Tsantalis *et. al.* [20] propose a technique to identify move method refactoring opportunities. In particular, for each method of the system, their approach forms a set of candidate target classes where the method should be moved. This set is obtained examining the entities that a method accesses from the system classes. This approach and the search-based approach proposed in [15] are, in our knowledge, the closest to our work. The main difference is that in previous works the class where a method should be moved is based only on structural relationships between methods. In our approach, we use also conceptual relationships between methods. In addition, our approach is based on an emerging learning technique, RTM, that also analyzes the original system design in order to capture other kind of relationships between methods that might be missed considering only structural, e.g., attribute references, and conceptual relationships.

5. CONCLUSION AND FUTURE WORK

We proposed to exploit method friendships to build a recommendation system supporting the software engineer in the identification of move method refactoring opportunities. The results achieved in a preliminary evaluation supported the applicability of such a metaphor and highlighted the valuable support given by RTM in the identification of refactoring opportunities.

Replicating our case studies in different contexts and using different experimental designs is the only way to corroborate the results presented in this paper and ensure generalization. We plan such empirical studies in the future. In addition, we have other items on our research agenda. In particular, currently we focus on the specific problem of identifying the *envied class* rather than on identifying all the methods that need to be moved. MethodBook takes a method as an input and suggests a class where the method should be placed or implemented in. Such an approach finds a natural collocation in Integrated Development Environments (IDE) to suggest whether a particular method under development is implemented in the right class or if it should be moved to some other location in order to increase the overall quality of the software. This type of recommendation provided during development and maintenance should proactively help software developers avoid the Feature Envy bad code smell. Future work will be devoted to (i) empirically verify such a conjecture and (ii) extend the proposed approach to discover candidate methods that need to be moved. In particular, MethodBook can take a set of classes as an input, e.g., the classes of some package, and identify the envied classes for each method. A mismatch between an identified envied class and the class where a method is actually implemented would indicate alternative design decisions for Move Method refactoring operations.

We also plan to improve MethodBook's accuracy for identifying envied classes. In particular, the process adopted in this paper does not take into account the number of friends and other methods contained in a suggested class. Future work will be devoted to analyze such scenarios and defining new heuristics for MethodBook to overcome such problems. Last but not least, we plan to apply the concept of friendships to other problems, such as software re-modularization.

Acknowledgement

This work is supported by NSF CCF-1016868. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

6. REFERENCES

[1] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

[2] G. Bavota, A. D. Lucia, A. Marcus, and R. Oliveto. A two-step technique for extract class refactoring. *ASE'10*, pages 151–154, 2010.

[3] G. Bavota, A. De Lucia, and R. Oliveto. Identifying extract class refactoring opportunities using structural and semantic measures. *JSS*, 84:397–414, 2011.

[4] E. Casais. An incremental class reorganization approach. *ECOOP'92*, pages 114–132, 1992.

[5] J. Chang and D. M. Blei. Hierarchical relational models for document networks. *Annals of Applied Statistics*, 2010.

[6] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *TSE*, 20(6):476–493, 1994.

[7] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Wiley-Interscience, 1991.

[8] B. Du Bois, S. Demeyer, and J. Verelst. Refactoring - improving coupling and cohesion of existing code. *WCRE'04*, pages 144–151, 2004.

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

[10] G. Gui and P. D. Scott. Coupling and cohesion measures for evaluation of component reusability. *MSR'06*, pages 18–21, 2006.

[11] A. Marcus, D. Poshyvanyk, and R. Ferenc. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *TSE*, 34(2):287–300, 2008.

[12] K. Maruyama and K. Shima. Automatic method refactoring using weighted dependence graphs. *ICSE'99*, pages 236–245, 1999.

[13] I. Moore. Automatic inheritance hierarchy restructuring and method refactoring. *OOPSLA'96*, pages 235–250, 1996.

[14] M. O'Keeffe and M. O'Cinneide. Search-based software maintenance. *CSMR'06*, pages 249–260, 2006.

[15] O. Seng, J. Stammel, and D. Burkhart. Search-based determination of refactorings for improving the class structure of object-oriented systems. *GECCO'06*, pages 1909–1916, 2006.

[16] F. Simon, F. Steinbr, and C. Lewerentz. Metrics based refactoring. *CSMR'01*, pages 30–38, 2001.

[17] M. Streckenbach and G. Snelting. Refactoring class hierarchies with kaba. *OOPSLA'04*, pages 315–330, 2004.

[18] L. Tahvildari and K. Kontogiannis. A metric-based approach to enhance design quality through meta-pattern transformation. *CSMR'03*, pages 183–192, 2003.

[19] A. Trifu and R. Marinescu. Diagnosing design problems in object oriented systems. *WCRE'05*, pages 155–164, 2005.

[20] N. Tsantalis and A. Chatzigeorgiou. Identification of move method refactoring opportunities. *TSE*, pages 347–367, 2009.