Playing with Refactoring: Identifying Extract Class Opportunities through Game Theory

Gabriele Bavota, Rocco Oliveto, Andrea De Lucia
DMI, University of Salerno, Fisciano (SA), Italy
Emails: {gbavota, roliveto, adelucia}@unisa.it

Giuliano Antoniol, Yann-Gaël Guéhéneuc
DGIGL, École Polytechnique de Montréal, Québec, Canada
Emails: antoniol@ieee.org, yann-gael.gueheneuc@polymtl.ca

***Abstract*—In software engineering, developers must often find solutions to problems balancing competing goals, e.g., quality versus cost, time to market versus resources, or cohesion versus coupling. Finding a suitable balance between contrasting goals is often complex and recommendation systems are useful to support developers and managers in performing such a complex task. We believe that contrasting goals can be often dealt with game theory techniques. Indeed, game theory is successfully used in other fields, especially in economics, to mathematically propose solutions to strategic situation, in which an individual's success in making choices depends on the choices of others. To demonstrate the applicability of game theory to software engineering and to understand its pros and cons, we propose an approach based on game theory that recommend extract-class refactoring opportunities. A preliminary evaluation inspired by mutation testing demonstrates the applicability and the benefits of the proposed approach.**

***Index Terms*—Game Theory; Quality metrics; Refactoring.**

I. Introduction

Game theory is a branch of mathematics widely applied in the social sciences, especially in economics. It attempts to mathematically capture the behavior of individuals in strategic situations in which an individual's success in making choices depends on the choices of others [1]. The games studied in game theory are well-defined mathematical objects involving two or more, usually non-cooperating, players; each of which plays a set of strategies. More precisely, a game consists of a set of players, a set of moves (or strategies) available to those players, and a specification of payoffs for each combination of moves. For a given game, it is of particular interest to determine the game solutions in which no player gains anything by changing only her own strategy unilaterally (situation of equilibrium).

Game theory still represents an unexplored territory in software engineering, even if some preliminary studies [2], [3] have highlighted the applicability of game theory techniques to software development. Yet, game theory has natural applications in software engineering where an optimal solution to many problems involves finding a balance between contrasting goals. For example, during the development of an object-oriented software system, developers and their managers must address design trade-offs associated with the balancing of competing constraints, such as cohesion and coupling. Indeed, developers strive to define classes having (i) strongly related and focused responsibilities, i.e., high cohesion and (ii) limited numbers of dependencies with other classes, i.e., low coupling. However, increasing cohesion might also result in increasing coupling and *vice-versa*. Thus, a trade-off between cohesion and coupling must be ideally attained. Similar situations arise during software maintenance. Classes structures undergo continuous modifications often drifting away from their original design and intent. In the long run, a system may become too complex, difficult to understand, modify, and evolve [4]. Thus, refactoring operations are often advocated to improve classes with deteriorated quality and, among other goals, re-establishing a balance between cohesion and coupling.

An optimal balance between contrasting goals, such as cohesion and coupling, is not easy to find. Closed analytical forms of an optimal (or near optimal) solution may be difficult to define and recently, search-based approaches have been advocated to find optimal or sub-optimal solutions [5]. In previous work, search-based approaches tried to find refactoring opportunities that optimize the overall quality of a system measured by a linear combination of cohesion and coupling that captures implicitly the trade-off between cohesion and coupling [5]. Alternatively, search-based approaches can provides the developers and managers with a Pareto front [7] of solutions optimizing cohesion and coupling [6]. However, there may be more than one optimal solution, each optimum can be achieved with different refactoring operations and it may be far from obvious to define the best sequence of refactoring operations. For example, the manager may need to iteratively inspect several Pareto fronts and decide which compromise she prefers. Refactoring operations are then applied to the system automatically or semi-automatically. Moreover, the reason why a particular refactoring operation must be performed at a certain stage is difficult to know as search-based approaches only report the solution, the operation to be performed, and not the reason why that particular refactoring operation must be preferred.

The above observations motivate our work. In particular, we believe that the perfect balance between different design goals is not always achieved by finding the optimal value of a function describing the overall quality of a system [1]. Instead, we advocate an approach that is able to capture the behavior of players in strategic situations (e.g., design trade-offs), in which the optimization of a goal depends on the choices performed to optimize other competing goals. Furthermore, we believe that it is important to know the reasons for a sequence of

TABLE I
PAYOFF MATRIX FOR THE PRISONER'S DILEMMA.

		Tom	
		confess	not confess
Sally	confess	**(6, 6)**	(0, 7)
	not confess	(7, 0)	(4, 4)

In bold the Nash equilibrium

refactoring operations to be preferred over others.

Consequently, we present a recommendation system that exploits game theory techniques. In particular, we use game theory to recommend *Extract Class* refactoring opportunities, finding a balance between class cohesion and coupling when splitting a class with different responsibilities into several classes. The sequence of refactoring operations is computed using a refactoring game, in which the Nash equilibrium (see Section II) defines the compromise between coupling and cohesion. A preliminary evaluation of the proposed approach on JHotDraw and ArgoUML classes was performed in a mutation-test like environment. In essence, we merged pairs or classes randomly-extracted from JHotDraw and ArgoUML. Merged classes plays the role of mutated programs; non-merged classes correspond to original, non-mutated programs, they are the oracle against which refactoring strategies can be compared. Reported results support the applicability and superiority of game theory over previous approaches.

The rest of the paper is organized as follows. Section II introduces the reader to game theory. Sections III and IV present the proposed approach and its preliminary evaluation, respectively. Finally, Section V gives concluding remarks.

II. BACKGROUND: THE PRISONER'S DILEMMA

The purpose of this section is to give a general introduction to game theory focusing on non-cooperative games. In particular, we report the prisoner's dilemma [8], a famous example of game theory. Suppose that there are two brokers accused of fraudulent trading activities: Sally and Tom. Both Sally and Tom are being interrogated separately and do not know what the other is saying. Both brokers want to minimize the amount of time spent in jail and here lies their dilemma: Table I represents the payoff matrix for their different possible choices, reporting the sentences for each player in consequence of their selected strategy. In particular,

- If *Sally (Tom) confesses and Tom (Sally) does not confess*: Sally (Tom) will not receive any sentence, while Tom (Sally) will have to stay in jail for the maximum sentence (seven years).
- If *both decide to confess*: both Sally and Tom will receive a sentence of six years.
- If *both decide not to confess*: both Sally and Tom will receive a sentence of four years.

Traditional applications of game theory attempt to find equilibria in such games. The most famous equilibrium is the Nash equilibrium [9]. Nash equilibrium is a solution to any game involving two or more players, in which each player is assumed to know the equilibrium strategies of the other players, and no player has anything to gain by changing only her own strategy unilaterally. Each finite game, i.e., a game with a finite number of players and actions, have at least one Nash equilibrium in mixed strategies [9].

In the example shown in Table I, there is only one Nash equilibrium, i.e., *(confess, confess)*. Indeed, if one of the players decides not to confess the other player can confess to minimize her own sentence. However, she would also sentence the other to the maximum punishment. Therefore, given the non-cooperative nature of the game, the minimum sentence for both players can be obtained only if both the players confess.

It is important to understand that the best solution for the two players is *(non confess, non confess)*, i.e., the Pareto optimum. The exemplified game presents two other Pareto optima, i.e., *(non confess, confess)*, and *(confess, non confess)* but these optima give the maximum payoff to only one of the two players i.e., minimize only one of the goal while the other is actually maximize.

Establishing a criterion to choose the best solution for such a non-cooperative game is usually hard. In fact, if we choose the solution with the minimum total sentence, i.e., maximum total payoff, we should choose between *(non confess, confess)* and *(confess, non confess)*, which heavily penalizes one of the two players. In this situation, the Nash equilibrium gives us a solution that not necessarily is the Pareto optimal solution but may represent a fair compromise between two competing goals while also providing an explanation of the equilibrium found.

III. GAME-BASED EXTRACT CLASS REFACTORING

The extract-class refactoring problem can be modelled as a non-cooperative game involving two players. Given a class to be refactored, the two players contend for the methods of the original class to build two new classes with higher cohesion and lower coupling than the original class. Our approach assumes that the original class must be split only in two classes, while a class with low cohesion could be split in more than two classes. Nevertheless, the proposed approach can be easily extended to extract more than two classes. In particular, once obtained the two classes, the developers and managers could analyze their cohesion and coupling and, if not satisfactory, apply again the approach on the newly extracted classes.

A. Playing with Methods

We now detail the modelling of the extract-class refactoring problem using game theory as a game involving two players S (Sally) and T (Tom). Each player is in charge to build a new class selecting methods from the original class to be refactored. The construction of the two classes is iterative. At each iteration of the process, a player (S or T) can select at most one method of the class to be refactored. Thus, at each iteration a player can add at most one method to her class under construction. A player selects the method to extract by considering the impact of adding the method on the cohesion and coupling of her class.

Specifically, the game starts by assigning to S and T the two methods having the lowest (structural and semantic) similarity. Thus, the refactoring process is guided by the two less cohesive methods. The two players will then iteratively contend for the remaining $n-2$ methods of the class to be refactored. In particular, during an iteration of the process, a player can perform one of the following moves:

- Selects the method m_i and yield the method m_j to her opponent $(i, j$ move);
- Selects the method m_i while her opponent does not select any method $(i, null$ move);
- Does not select any method while her opponent selects the method m_j $(null, j$ move).

The purpose of $null$ move is to increase the rationality of a player, i.e., a player selects a method only if there is a clear advantage in selecting it. Indeed, without the $null$ move, a player must select a method at each iteration of the refactoring process. Also, without the $null$ move, a trivial splitting of the class to be refactored could happen: the original class is split in two new classes having the same number of methods.

The move to be performed during an iteration of the process is chosen by finding the Nash equilibrium in the payoff matrix. The Nash equilibria are obtained using the algorithm defined in [10], which enumerates all the equilibria in a two-player finite game. If there are more than one Nash equilibrium, then we select the one having the highest sum of the payoffs of both players.

Once identified the move to be performed, each player adds to its class the method (if any) obtained performing the selected move, i.e., the player incrementally builds its class. The process stops when each method of the original class is assigned to one of the two players.

B. Computing the Payoff Matrix

The payoff matrix is at the core of the approach. The generic entry of such matrix, $p_{i,j}$, is a couple of payoffs (p_T, p_S), representing the payoffs of each player corresponding to a generic move (i, j). These payoffs depend on the impact of the methods obtained by the two players, S and T, on the cohesion and coupling of their classes. In particular, p_k measures the impact on cohesion obtained by player k selecting the method m_i balanced with the impact on coupling caused by yielding the method m_j to the other player.

To quantify the gain obtained by a player k selecting a method m_i in terms of cohesion and coupling, we define a function f_{cc} computed as follows:

$$f_{cc}(C_k, m_j) = \frac{1}{N} \sum_{m_i \in C_k} sim(m_i, m_j)$$

where C_k represents the set of method assigned at a certain iteration to the player k, N is the number of methods in C_k, and $sim(m_i, m_j)$ captures the relationships among methods that affect cohesion and coupling. In particular, $sim(m_i, m_j)$ is obtained by combining three different (structural and semantic) measures, i.e., Structural Similarity between Methods (SSM) [11], Call-based Interaction between Methods (CIM) [12], and Conceptual Similarity between Methods (CSM) [13]. These measures capture three distinct ways in which methods relate to one another, each reflecting a different type of relationships among methods that impact class cohesion and coupling.

Structural Similarity between Methods (SSM) captures the 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. Call-based Interaction between Methods (CIM) [12] takes into account the calls performed by the methods; the higher the interactions between methods, e.g., a method m_i is only called by m_j, the higher the similarity between the two methods. Clearly, methods with high interactions should be in the same class to reduce coupling between classes. Finally, we also capture semantic information between methods trough the Conceptual Similarity between Methods (CSM) metric [14]: two methods are conceptually related if their (domain) semantics are similar, i.e. they perform conceptually similar actions.

All these similarity measures have values in $[0, 1]$. We compute the overall structural and semantic similarity between a couple of methods m_i and m_j as:

$$sim(m_i, m_j) = w_{SSM} \cdot SSM(m_i, m_j) +$$

$$w_{CIM} \cdot CIM(m_i, m_j) + w_{CSM} \cdot CSM(m_i, m_j)$$

where $w_{SSM}, w_{CIM}, w_{CSM} \in [0, 1]$ and $w_{SSM} + w_{CIM} + w_{CSM} = 1$. Their values express the confidence (i.e., weight) in each measure.

The generic entry $p_{i,j}$ of P is a couple of payoffs that depends on the f_{cc} function and is computed as follows:

$$p_{i,j} = \begin{cases} (f_{cc}(C_S, m_i) - f_{cc}(C_S, m_j), \\ f_{cc}(C_T, m_j) - f_{cc}(C_T, m_i)) & \text{if } i, j \neq null; \\ (f_{cc}(C_S, m_i), \mu - f_{cc}(C_T, m_i)) & \text{if } i \neq null, j = null; \\ (\mu - f_{cc}(C_S, m_j), f_{cc}(C_T, m_j)) & \text{if } j \neq null, i = null; \\ (-1, -1) & \text{if } i = j. \end{cases}$$

where $\mu > 0$ is a configuration parameter used to balance the payoff of the $null$ move with respect to the other payoffs. This parameter gives a chance to the player to play the $null$ move. In fact, without μ, the payoff of the $null$ move will always be negative and the player would hardly select such a move. The value of this parameter has been empirically defined; the best refactoring opportunities are achieved setting $\mu = 0.5$. The two indexes i and j are in $[1 \dots p+1]$ where p is the number of remaining methods to be assigned. Other than the possible methods to be assigned (p), a $null$ move must also be considered $(p+1)$.

The value of the payoff for each player is in $[-1, 1]$. Thus, the diagonal of the payoff matrix is set to $(-1, -1)$, the lowest payoff for both the players, to avoid that the two players play the same move, seek the same method, or play both the $null$ move.

This definition of the payoff matrix allows us to consider at the same time both the cohesion and coupling of the new

f_{cc} values - Iteration I

	m_1	m_2	m_3	m_4	m_5	m_6
m_1	1.00	0.70	0.21	*0.02*	*0.10*	**0.00**
m_2	0.70	1.00	0.30	*0.06*	*0.01*	*0.03*
m_3	0.21	0.30	1.00	0.50	0.40	0.22
m_4	*0.02*	*0.06*	0.50	1.00	0.60	0.30
m_5	*0.10*	*0.01*	0.40	0.60	1.00	0.80
m_6	**0.00**	*0.03*	0.22	0.30	0.80	1.00

Payoff Matrix For Class *c* - Iteration I

		C_T m_2	m_3	m_4	m_5	null
C_S	m_2	(-1.00, -1.00)	(0.49, 0.22)	(0.70, 0.30)	**(0.70, 0.80)**	(0.70, 0.50)
	m_3	(-0.49, -0.22)	(-1.00, -1.00)	(0.21, 0.08)	(0.21, 0.58)	(0.21, 0.28)
	m_4	(-0.70, -0.30)	(-0.21, -0.08)	(-1.00, -1.00)	(0.00, 0.50)	(0.00, 0.20)
	m_5	(-0.70, -0.80)	(-0.21, -0.58)	(0.00, -0.50)	(-1.00, -1.00)	(0.00, -0.30)
	null	(-0.20, 0.00)	(0.29, 0.22)	(0.50, 0.30)	(0.50, 0.80)	(-1.00, -1.00)

C_S = { m_1 } C_T ={ m_6 }

f_{cc} values - Iteration II

	m_3	m_4
C_S	0.26	0.00
C_T	0.31	0.45

Payoff Matrix For Class *c* - Iteration II

		C_T m_3	m_4	null
C_S	m_3	(-1.00, -1.00)	(0.26, 0.14)	**(0.26, 0.19)**
	m_4	(-0.26, -0.14)	(-1.00, -1.00)	(0.00, 0.05)
	null	(0.25, 0.31)	**(0.50, 0.45)**	(-1.00, -1.00)

C_S = { m_1, m_2 } C_T ={ m_6, m_5 }

f_{cc} values - Iteration III

	m_3
C_S	0.26
C_T	0.37

Payoff Matrix For Class *c* - Iteration III

		C_T m_3	null
C_S	m_3	(-1.00, -1.00)	**(0.26, 0.13)**
	null	**(0.25, 0.37)**	(-1.00, -1.00)

C_S = { m_1, m_2 } C_T ={ m_6, m_5, m_4 }

Fig. 1. An example application of our approach (μ = 0.5).

classes. If the player S chooses the move i, j selecting the method m_i and yielding the method m_j to T, the payoff for S considers the effect on the cohesion of the new class C_S as the similarity between the methods in C_S and m_i as it considers the coupling increasing with the other new class, i.e., C_T, as the similarity between the methods in C_S and m_j.

C. An Example Application of our Approach

To better understand our approach, let us assume that a class C contains six methods, $C = \{m_1, m_2, m_3, m_4, m_5, m_6\}$ and further suppose that the f_{cc} function and payoff matrices at each iteration of the approach are those reported in Figure 1. In the first iteration, we consider equal to zero all the values ≤ 0.2 of f_{cc} to remove spurious relationships between the methods of the original class. Such values are indicated in *italic* in Figure 1. In Figure 1, the payoff matrix entries in bold correspond to the Nash equilibrium. The value 0.2 has no particular reason to be chosen and was selected just for illustrating the complete process.

The two players, S and T, are in charge of defining two new classes (C_S and C_T) extracting methods from C. At initialization, the two methods having the lowest similarities are m_1 and m_6, so we assign m_1 to C_S and m_6 to C_T. We then build the payoff matrix for the first iteration (see Figure 1) and find the Nash equilibrium m_2, m_5, which means that player S takes method m_2 while player T takes method m_5. After the first iteration, we calculate the new f_{cc} values and recompute the payoff matrix using the new configuration of C_S and C_T (see Figure 1).

The process continues in a second iteration and we find two Nash equilibria. We select the Nash equilibrium with the higher total payoff: $null, m_4$. This equilibrium means that player S takes no method while player T takes method m_4.

The process goes into a third iteration. We compute the new f_{cc} values and recompute the payoff matrix. The equilibrium is now $null, m_3$. At the end of the process, we have two new classes, namely $C_S = \{m_1, m_2\}$ and $C_T = \{m_3, m_4, m_5, m_6\}$.

IV. Preliminary Evaluation

A preliminary evaluation of our approach was performed on two well-designed open-source systems, namely ArgoUML version 0.16 and JHotDraw version 6.0b1. The evaluation was carried out to answer the following research questions:

- **RQ**$_1$: *Are the refactoring opportunities identified through the Nash equilibrium better than those identified through the Pareto optimum?*
- **RQ**$_2$: *Is the proposed approach better than other approaches that support* Extract Class *refactoring?*

Regarding the first research question, we adapt our approach to find a Pareto optimum instead of a Nash equilibrium and then compare the solutions of the two approaches. For the second research question, we compare our approach with De Lucia *et al.*'s approach [15] because their goals are identical. In particular, both approaches take as input a class with a low cohesion and split it in two classes having a higher cohesion. In [15], a class to be refactored is represented as a weighted graph, where each edge is weighted by a measure that captures the structural and semantic relationships between methods, i.e., the same measures used in the proposed approach. Then, a MaxFlow-MinCut algorithm is applied to split the graph representing the class to be refactored in two sub-graphs representing two classes having higher cohesion than the original class.

The evaluation planning is inspired by mutation testing. In particular, we randomly select two classes of one of the object systems, merge them in a single class $\widehat{C}_m$ and then use the experimented approaches to split the merged class in two classes. Given a merged class $\widehat{C}_m$, an approach is expected to

TABLE II
F-MEASURE OF THE EXPERIMENTED APPROACHES. THE NUMBER OF MERGING OPERATIONS IS REPORTED BETWEEN BRACKETS.

	ArgoUML (150)			JHotDraw (50)		
	Mean	Median	Std.dev.	Mean	Median	Std.dev.
Game Theory	0.897	0.966	0.201	0.846	1.000	0.227
Pareto Optimum	0.876	0.918	0.149	0.822	0.907	0.211
De Lucia *et al.* [15]	0.769	0.693	0.241	0.761	0.735	0.214

generate two classes similar to the original ones. In general, we cannot be sure that generated classes will be exactly as the original for a variety of reasons, including the fact the coupling and cohesion of original classes may well point out the need of refactoring.

However, to simplify our evaluation in this preliminary case study, we conservatively quantify accuracy as a function of the total number of methods correctly and incorrectly moved in the split classes when compared with the original ones. Since the merged classes have a good quality in terms of cohesion and coupling, the ideal behavior (i.e., correct) is that the split classes are identical to (i.e., contain the same methods as) the original classes. This behavior means that the approach is able to identify meaningful sequences of refactoring operations, because starting from a class with low cohesion, the approach is able to split the class in two classes with a higher cohesion that that of the original classes.

The accuracy of the identified refactoring opportunities can be evaluated using two well-known Information Retrieval (IR) metrics, recall and precision [16]. Since the two metrics measure two different concepts, a balance between them can be obtained using an aggregate metric, the F-measure [16].

Table II reports the descriptive statistics of F-measure for both ArgoUML and JHotDraw obtained with the new approach, the one based on the identification of Pareto optima, and the approach proposed by De Lucia *et al.* [15], respectively. The reported results are obtained using two (similar) configurations of weights for the similarity measures. We obtained best results with the setting $w_{CIM} = 0.2$, $w_{SSM} = 0.1$, $w_{CSM} = 0.7$ on ArgoUML and with $w_{CIM} = 0.1$, $w_{SSM} = 0.2$, $w_{CSM} = 0.7$ on JHotDraw.

The results reported in Table II seem to support the superiority of the new approach. In particular, we can positively respond to both $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$. These results suggest the applicability of game theory to software engineering with encouraging results, as (i) the results achieved prove that, in some situations, the perfect balance between different design goals is not achieved by finding the optimal value of a function describing the overall quality of the system (Pareto optimum) but exploiting the Nash equilibrium and (ii) the game theory-inspired approach outperfoms a previous approach.

V. CONCLUSION AND FUTURE WORK

This paper proposed a novel approach based on game theory to support *Extract Class* refactoring. Given a class to be refactored, the approach models a non-cooperative game where two players contend for the methods of the original class to build two new classes with higher cohesion than the original class. The results achieved in a preliminary evaluation supported the applicability and superiority of game theory.

Other than refactoring, other software engineering problems characterized by competing goals are likely to be representable as games and resolved in a similar way. For example, software re-modularization, project scheduling, or regression test selection. Thus, this paper sheds light on a branch of mathematics that has received little attention from software engineers but that can be effectively applied to tackle multi-objective optimizations. We believe that the investigation of game theory in software engineering can contribute to the development of novel solutions in software engineering research and practice.

Several items are on our work agenda to pursue our research work. We plan to investigate additional game theory techniques, such as cooperative games, and determine whether the choices we made here are the best or can be improved. We also plan to (i) better understand the impact of different parameter settings and (ii) carry out more empirical studies, using other systems and different experimental settings. A natural extension of this work will be to adapt our technique to split a given class in more than two classes. Last but not least, we also plan to apply game theory techniques to other problems, such as software re-modularization and scheduling.

ACKNOWLEDGEMENTS

Giuliano Antoniol and Yann-Gaël Guéhéneuc are supported by the NSERC Research Chairs in Software Change and Evolution and in Software Patterns and Patterns of Software.

REFERENCES

[1] M. Dresher, *The Mathematics of Games of Strategy: Theory and Applications*. Prentice-Hall, 1961.
[2] M. Grechanik and D. E. Perry, "Analyzing software development as a noncooperative game," *EDSER*, 2004.
[3] V. Sazawal and N. Sudan, "Modeling software evolution with game theory," in *ICSP*, 2009, pp. 354–365.
[4] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley, 1999.
[5] O. Seng, J. Stammel, and D. Burkhart, "Search-based determination of refactorings for improving the class structure of object-oriented systems," *GECCO*, 2006, pp. 1909–1916.
[6] M. Harman and B. F. Jones, "Search based software engineering," *IST*, vol. 43, no. 14, pp. 833–839, 2001.
[7] K. Deb, *Multi-Objective Optimization using Evolutionary Algorithms*. JW, 2001.
[8] A. Rapoport and A. M. Chammah, *Prisoner's Dilemma*. University of Michigan Press, 1965.
[9] J. Nash, "Non-cooperative games," *Annals of Mathematics*, vol. 54, no. 2, pp. 286–295, 1951.
[10] O. Mangasarian, "Equilibrium points in bimatrix games," *Journal of the Society for Industrial and Applied Mathematics*, vol. 12, no. 4, pp. 778–780, 1964.
[11] G. Gui and P. D. Scott, "Coupling and cohesion measures for evaluation of component reusability," *MSR*, 2006, pp. 18–21.
[12] G. Bavota, A. D. Lucia, A. Marcus, and R. Oliveto, "A two-step technique for extract class refactoring," *ASE*, 2010.
[13] D. Poshyvanyk, A. Marcus, R. Ferenc, and T. Gyimóthy, "Using information retrieval based coupling measures for impact analysis," *EMSE Journal*, vol. 14, no. 1, pp. 5–32, 2009.
[14] A. Marcus and D. Poshyvanyk, "The conceptual cohesion of classes," in *ICSM*, 2005, pp. 133–142.
[15] A. De Lucia, R. Oliveto, and L. Vorraro, "Using structural and semantic metrics to improve class cohesion," *ICSM*, 2008, pp. 27–36.
[16] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. Addison-Wesley, 1999.