# Refactoring Recommendations Based on the Optimization of Socio-Technical Congruence

Manuel De Stefano*, Fabiano Pecorelli*, Damian Andrew Tamburri†, Fabio Palomba*, Andrea De Lucia*
*SeSa Lab
University of Salerno, Italy
m.destefano36@studenti.unisa.it
{fpecorelli, fpalomba, adelucia}@unisa.it
†Jheronimus Academy of Data Science, The Netherlands
d.a.tamburri@tue.nl

***Abstract*—Software development is known to be a social activity that involves developers, project managers, and stakeholders. Recent studies have proved a direct relation between social and technical aspects, *e.g.*, poor coordination among developers may lead to an increase of technical debt in source code. The so-called socio-technical congruence measures the level of coordination existing in an organization at their different levels. In this late-breaking idea paper, we propose a novel way to employ the socio-technical congruence in the context of source code quality improvement: we design a community-based refactoring recommendation approach that aims at optimizing socio-technical congruence while keeping into account the source code dependencies among the components of a software project. A search-based algorithm is employed to this purpose and we envision the novel approach to be suitable for providing Extract Class and Extract Package refactoring recommendations.**

***Index Terms*—Refactoring; Search-based Software Engineering; Recommendation Systems.**

## I. Introduction

Software needs continuously to change to keep being useful, as Lehman stated [1], so developers have to conduct maintenance activities to keep the software useful to the end-user. Sometimes, maintenance activities have no reflection on external behavior, but they are fundamental as well: these are called refactoring activities, that have the aim to improve the design and the quality of the existing source code under various aspects (e.g., readability, understandability, cohesion, coupling, ecc.) [2]. Unfortunately, refactoring it is not a trivial activity, in particular when it affects the software architecture. To support developers in refactoring activities, tools and techniques aimed to refactor the source code automatically have been proposed. However, only a few of them are able to address large scale re-modularization [3], which indeed cannot be performed as a big bang operation, but rather as small, commit-friendly steps [4]. These kinds of tools only consider the structural characteristics of source code and propose refactoring based only on this information.

Over the last years, researchers have been investigating the impact of developers' social networks on the sustainability of open- and closed-source communities as well as source code quality, finding them to be a highly relevant factor for the success of software systems [5], [6], [7]. Socio-Technical congruence [5], defined as a metric used to investigate how proper the alignment or fit is between the organizational structure and the software architecture, has proven to play a central role in quality issues. As an example, Kwan et al. [6] showed that this alignment affects the build success. At the same time, Palomba et al. [7] found that community-related factors can increase the criticality of source code quality issues. These social aspects, however, have rarely been considered when taking decisions about software architecture, although they influence each other.

So, as of a late-breaking idea, in this paper, we propose a technique that, exploiting socio-technical aspects can recommend both structural refactorings, *e.g.*, extract class and extract package refactorings, and *social* refactoring, intended as a community restructuring and teams rearrangement. By mining software repositories, we can construct three graphs, representing the communication among developers (dev-dev graph), the collaboration among developers (dev-file graph), and the static dependencies among files (file-file graph), and then combine them in one augmented weighted graph, representing the socio-technical congruence in the software's architecture. This graph si fed into a GA, to find the best set of relationships between developers and components.

## II. The Technique

The proposed technique is mainly made up of two steps, as depicted in Fig. 1. First, for a given project, its software repository is mined to acquire all the information required by our technique. Secondly, a genetic algorithm is executed to compute an improved version of the socio-technical structure of the project under analysis. In the following, we describe these two steps more deeply.

### A. Repository Mining

The first step consists of mining the software repository to obtain information about communication and collaboration among developers and dependencies among source files. We represent them by using three graphs, namely the dev-dev graph (for communication), the dev-file graph (for collaboration), and file-file graph (for code dependencies)—the obtained information span on a customizable time interval.

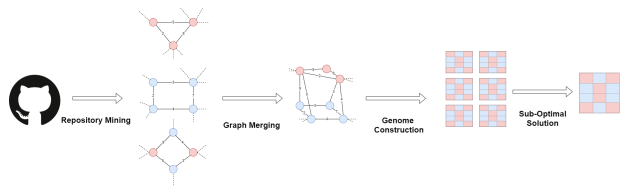

Fig. 1. Schema of the technique's steps

The first graph can be obtained mining the issue tracker conversations, resulting in an undirected weighted graph in which the nodes represent the developers and the edges the communications among them. If a developer participated in at least one issue thread with another developer, an edge is created to connect them. The weights are represented by the total number of replies given in a shared thread, without considering the criticality of the issue. The dev-file graph is obtained considering the commits made by developers on source files. The obtained graph is indeed bipartite, as we have only two types of nodes, developer and file, connected. A developer node is connected to a file node if the developer has committed at least once that file in the considered time interval. The weight on the edges consists of the developer's total number of commits on that file in the considered interval. Although there are some other social aspects that have been overlooked (*i.e.*, developers' role), these two are the only one that can actually be extracted directly mining the repository. The file-file graph, finally, is obtained considering structural code dependencies. A file node is connected with another if it contains a reference to a code component contained in another file or vice versa. The obtained graph is then undirected, and the weights on the edges represent the actual number of references present among the two source files.

At the end of phase one, as Fig. 1 depicts, these three graphs are merged to have all the communication, collaboration, and coupling information represented by one single structure. As the figure shows, this merge consists in connecting the dev-dev and the file-file graph using the dev-file as a sort of bridge. This construction could be imagined as if we lie down the dev-dev and the file-file package on two parallel plains and connect the nodes present on these plain using the dev-file edges.

## B. Graph Optimization and Refactoring Recommendation

The second phase of the technique involves executing a many-objective Genetic Algorithm (GA) to find an improved version of the starting graph, given as input by the previous phase. In order to execute the GA, the augmented graph needs to be converted into a representation suitable for the GA, as shown by the schema in Fig. 1. This is relatively easy as the adjacency matrix can be directly employed to represent the graph as a two-dimensional vector suitable for the GA.

Starting from an initial population, in which each individual is represented by an adjacency matrix that is structured in the same manner of the starting matrix and pseudo-randomly generated based on it, the GA iteratively generates higher-quality solutions based on three main objectives aiming at maximizing the communication and the collaboration between developers and minimizing the coupling among components. Besides these three objectives, it is also essential to maximize the similarity between the output graph and the starting one to prevent the GA from excessively altering the original structure while optimizing the objectives. The equation 1 reports the fitness function we define. For the sake of comprehensibility we refer to different parts of the graph with different names, *i.e.*, $G_{dev-dev}$, $G_{dev-file}$, and $G_{file-file}$.

$$f(G)=\begin{cases} max(\sum com_{d_i,d_j})\forall(d_i,d_j) \in G_{dev-dev} \\ max(\sum min(col_{d_i,f}, col_{d_j,f})\forall(u,v) \in G_{dev-file} \\ min(\sum cou_{f_i,f_j})\forall(u,v) \in G_{file-file} \\ min(dist(G_s,G)) \end{cases} \tag{1}$$

where $com_{d_i,d_j}$ is the degree of communication between developers $d_i$ and $d_j$ in the dev-dev graph, $max(col_{u_i,v}, col_{u_j,v})$ represents the number of co-commits of developers $d_i$ and $d_j$ on the file $f$, $cou_{f_i,f_j}$ is the degree of coupling between files $f_i$ and $f_j$, and $dist(G_s,G)$ is the minimum number of operations to transform graph $G_s$ to graph $G$.

At each iteration, an offspring population is generated selecting the best individuals of the current population (based on their fitness scores) and creating new ones using the reproduction, consisting of two operators: mutation and crossover. The mutation operator we define consists of adding/removing edges in a particular individual, while the crossover between two graphs is applied by exchanging entire sub-graphs with each other. We implemented these operations by applying Tsai et al. directives [8]. The GA runs until the budget expires, and then the fittest solution is chosen as a refactoring target. Please note that the provided solution is just a preliminary step toward a complete refactoring recommendation approach. It can be used to extract information about the main problems in the starting socio-technical structure and on how to refactor them.

## III. Conclusions and Future Work

In this paper, we have presented a novel search-based approach for refactoring recommendations of socio-technical structures. Our approach is suitable to optimize not only the coupling between software components, but also the way developers communicate and collaborate. As future work, we first plan to validate the socio-technical graph construction approach by exploiting a dataset of projects that have been already analyzed in the context of community-related and architectural aspects. Then we plan to conduct an empirical study aiming at assessing the architecture's quality improvement after applying the refactoring operations recommended by the technique, taking computational efficiency of such approach in this day-to-day activity as well. Finally, we plan to consider other social aspects (*e.g.*, developers' knowledge or role) in order to perform a better optimization and taking into account both social and structural history (*i.e.*, team changes and file changes) to get better and safer optimizations.

## References

[1] M. M. Lehman, “Programs, life cycles, and laws of software evolution,” *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, 1980.

[2] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[3] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner, “Bunch: A clustering tool for the recovery and maintenance of software system structures,” in *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM’99).’Software Maintenance for Business Change’(Cat. No. 99CB36360)*. IEEE, 1999, pp. 50–59.

[4] M. Hall, M. A. Khojaye, N. Walkinshaw, and P. McMinn, “Establishing the source code disruption caused by automated remodularisation tools,” in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 466–470.

[5] 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 *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, 2008, pp. 2–11.

[6] I. Kwan, A. Schroter, and D. Damian, “Does socio-technical congruence have an effect on software build success? a study of coordination in a software project,” *IEEE Transactions on Software Engineering*, vol. 37, no. 3, pp. 307–324, 2011.

[7] F. Palomba, D. A. A. Tamburri, F. A. Fontana, R. Oliveto, A. Zaidman, and A. Serebrenik, “Beyond technical aspects: How do community smells influence the intensity of code smells?” *IEEE transactions on software engineering*, 2018.

[8] M.-W. Tsai, T.-P. Hong, and W.-T. Lin, “A two-dimensional genetic algorithm and its application to aircraft scheduling problem,” *Mathematical Problems in Engineering*, vol. 2015, 2015.