# Recommendation System for Software Refactoring Using Innovization and Interactive Dynamic Optimization

Wiem Mkaouer, Marouane Kessentini, Slim Bechikh
University of Michigan, MI, USA
firstname@umich.edu

Kalyanmoy Deb
Michigan State University, MI, USA
kdeb@egr.msu.edu

Mel Ó Cinnéide
University College Dublin, Ireland
mel.ocinneide@ucd.ie

## ABSTRACT

We propose a novel recommendation tool for software refactoring that dynamically adapts and suggests refactorings to developers interactively based on their feedback and introduced code changes. Our approach starts by finding upfront a set of non-dominated refactoring solutions using NSGA-II to improve software quality, reduce the number of refactorings and increase semantic coherence. The generated non-dominated refactoring solutions are analyzed using our innovization component to extract some interesting common features between them. Based on this analysis, the suggested refactorings are ranked and suggested to the developer one by one. The developer can approve, modify or reject each suggested refactoring, and this feedback is used to update the ranking of the suggested refactorings. After a number of introduced code changes, a local search is performed to update and adapt the set of refactoring solutions suggested by NSGA-II. We evaluated this tool on four large open source systems and one industrial project provided by our partner. Statistical analysis of our experiments over 31 runs shows that the dynamic refactoring approach performed significantly better than three other search-based refactoring techniques, manual refactorings, and one refactoring tool not based on heuristic search.

## Categories and Subject Descriptors

D.2 [**Software Engineering**]

## General Terms

Algorithms, Reliability

## Keywords

Search-based software engineering, software quality, refactoring

## 1. INTRODUCTION

Software systems rapidly become complex and difficult to maintain. It has been reported that the cost of maintenance and evolution activities comprises more than 80% of total software costs. In addition, it has been shown that software maintainers spend around 60% of their time in understanding the code [10]. To facilitate maintenance tasks, one of the widely used techniques is refactoring, which improves design structure while preserving the overall functionality of the software [2].

There has been much work on different techniques and tools for software refactoring [2][3][4][5][6][17] that can be mainly classified into two categories: manual and fully-automated approaches. For the first category, several tools are proposed to provide support for the application of several types of refactoring manually [2][3]. The developers identify which refactoring type to apply and where. Thus, the manual refactoring process can be a tedious task for developers. In the second category of refactoring studies [4][5][6], most of the proposed approaches generate as output a long sequence of refactorings that can be applied by developers to improve the quality of systems by fixing, for example, code smells [2]. Here the developers have to accept the entire solution in spite of the fact that they prefer, in general, stepwise interactive approaches where they have total control of the refactorings being applied. Few studies consider the suggestion of refactoring operations based on interaction between the developer and the refactoring tool.

We propose a novel interactive recommendation tool for software refactoring that dynamically adapts and suggests refactorings to developers based on their feedback and introduced code changes. Our approach starts by finding upfront a set of refactoring solutions using a multi-objective evolutionary algorithm NSGA-II, proposed by Deb [7], to improve software quality, reduce the number of refactorings and increase semantic coherence. The output of NSGA-II is a set of non-dominated refactoring solutions that find a good trade-off between these three objectives. One of the challenges when adapting a multi-objective technique to a software engineering problem is how to select the best solution from the set of non-dominated ones, called the Pareto front [8]. To this end, we propose, for the first time, the use of innovization (innovation through optimization) [9] to analyze and explore the Pareto front interactively with the developers. Our innovization algorithm starts by finding the most frequent refactoring pattern/operations between the set of non-dominated refactoring solutions. Based on this analysis, the suggested refactorings are ranked and suggested to the developer one by one. The developer can approve, modify or reject each suggested refactoring. This feedback is then used to update the ranking of the suggested refactorings. After a number of introduced code changes, a local search [1] is executed to update and adapt the set of refactoring solutions suggested by NSGA-II. We implemented our proposed approach and evaluated it on four open source systems, as well as one industrial system provided by our industrial partner.

## 2. DINAR: DYNAMIC INTERACTIVE MULTI-OBJECTIVE REFACTORING

We first present an overview of our technique called DINAR (Dynamic INterActive Refactoring) and then provide the details of our problem formulation and the solution approach.

## 2.1 Overview

The goal of our approach is to propose a dynamic, interactive way for software developers to refactor their systems. The general structure of the DINAR approach is sketched in Figure 1.

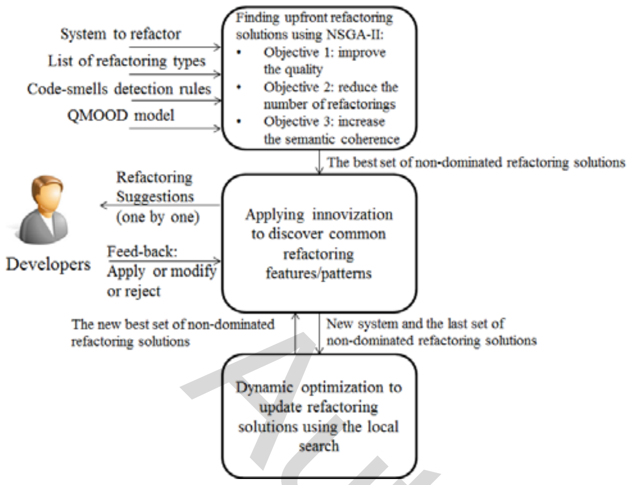

System to refactor
List of refactoring types
Code-smells detection rules
QMOOD model
Finding upfront refactoring solutions using NSGA-II:
Objective 1: improve the quality
Objective 2: reduce the number of refactorings
Objective 3: increase the semantic coherence
The best set of non-dominated refactoring solutions
Refactoring Suggestions (one by one)
Developers
Feed-back: Apply or modify or reject
Applying innovization to discover common refactoring features/patterns
The new best set of non-dominated refactoring solutions
New system and the last set of non-dominated refactoring solutions
Dynamic optimization to update refactoring solutions using the local search

**Figure 1. Approach overview: DINAR.**

The three main components of DINAR are: 1- finding the best set of refactoring solutions that satisfies three objectives using NSGA-II [9]; 2- applying our innovization algorithm to rank the refactorings and suggest them to developers; and 3- updating the refactoring solutions after a number of interactions with developers using a multi-objective local search algorithm [1].

Our refactoring framework starts by finding upfront the list of best refactoring solutions that represents a good trade-off between improving software quality which corresponds to minimizing the number of code-smells and improving QMOOD (Quality Model for Object-Oriented Design) quality metrics [10]; minimizing the size of the refactoring solutions (number of refactorings) and maximizing/preserving the semantic coherence of the design. Therefore, we use a multi-objective optimization algorithm, NSGA-II, to compute this optimal sequence of refactorings based on our previous work [4].

The output of this first step is a set of non-dominated refactoring solutions, called the Pareto front [7], which optimizes the three objectives described above. The second component of DINAR explores this Pareto front in an intelligent manner using an innovization algorithm (INNOVation through optimIZATION) [9] to rank recommended refactorings, and suggest them to the developer one by one as a sequence of transformations, based on several features: number of occurrences of a refactoring operation in all the Pareto front solutions, the order of the refactoring in the sequence, and developer feedback. In fact, the feedback from the developer can be to approve, apply, modify or reject the suggested refactoring. This feedback is used by our innovization algorithm to guide, implicitly, the exploration of the Pareto front to find the optimal solution that sometimes does not correspond exactly to a solution generated by NSGA-II.

After number of interactions, developers may have modified or rejected a high number of suggested refactorings or introduced several new code changes (new functionalities, etc.). In this case, the third component of DINAR is executed to update the last set of non-dominated refactoring solutions using an indicator-based local search [1] based on the three objectives defined in the first component. We selected indicator-based local search since it is a well-known quick local search algorithms that can update the solutions quickly based on new changes in the inputs.

The output of this third component is a new set of updated refactoring solutions that will be recommended to the developer one by one using the innovization component. The second component of DINAR is executed when the developers decide to stop refactoring the system and the third component is executed periodically after a high number of changes have been performed on the system.

## 2.2 Solution Approach

We describe in the following sections the details of the two main interactive components of DINAR.

### *2.2.1 Interactive Recommendation of Refactorings*

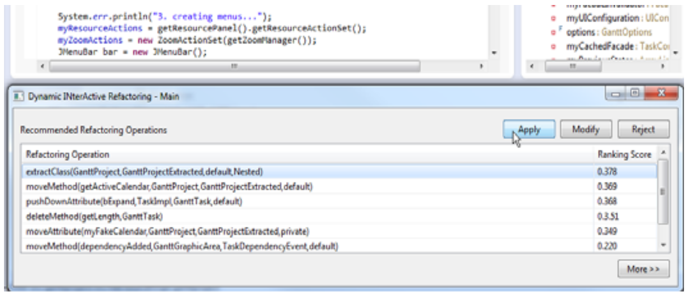
After the multi-objective optimization task described in the previous step, a set of optimal refactoring solutions are generated that find a good trade-off between the three objectives. We can now analyze these solutions to investigate if there exists some common principles among all or many of these optimal refactoring solutions. For example, it is interesting to see if most of the refactoring solutions have some common features such as common refactoring operations among most of the solutions and/or a specific common order in which to apply the refactorings. Such information is used to rank the suggested refactorings for developers using the following formula:

$$Rank(R_{i,j}) = \frac{number\_occurence}{\max\_number\_occurence} + \frac{\sum_{k=1}^{i} Sim(R_{k,j}, \text{recommende d\_ref})}{\#recommende\ d\_ref}$$

where $R_{i,j}$ is the refactoring operation number (index in the solution vector) *i* of solution number *j*. The first component of the ranking formula counts the number of occurrences of the refactoring operation $R_{i,j}$ among all the Pareto front solutions and normalizes it between 0 and 1. The second component compares the previous refactoring operations already applied by the developers (feedback) and the solution *j*.

The ranking of refactorings is updated automatically after every feedback (interaction) from the developer. DINAR proposes three levels of interaction as described in Figure 2. The developer can check the ranked list of refactorings then he can apply, modify or reject the refactoring. If the selected action is ‘apply’ then the refactoring will be automatically applied. If the developer prefers to modify it then DINAR can assist him during the modification process as described in Figure 3. In fact, DINAR proposes to the developer a set of recommendations to modify the refactoring based on the history of changes applied in the past and the semantic similarity between code elements (classes, methods, etc.). For example, if the developer wants to modify a move method refactoring then, once the source method has been specified, DINAR can suggest a list of possible target classes ranked based on the history of changes and semantic similarity. This is an interesting feature since developers may know which method to move but may not be certain which the best target class is. The same observation is valid for the other refactoring types. Another action that the developers can select is to reject/delete a refactoring from the list. After every action selected by the developer, the ranking is updated based on developer feedback as described in the ranking formula where the second component is updated dynamically.

After a number of modified or rejected refactorings, or the introduction of several new code changes, the generated Pareto front of refactoring solutions by NSGA-II is updated since the system was modified in different locations. Thus, the next component of DINAR is executed to update the refactoring solutions based on the new software system. The next section describes this step.



```
System.err.println("3. creating menus...");
myResourceActions = getResourcePanel().getResourceActionSet();
myZoomActions = new ZoomActionSet(getZoomManager());
JMenuBar bar = new JMenuBar();
```

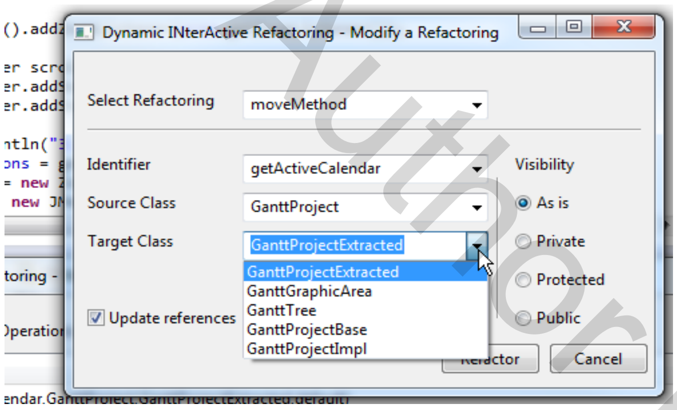
myUIConfiguration : UICon
options : GanttOptions
myCachedFacade : TaskCo

Dynamic INterActive Refactoring - Main

Recommended Refactoring Operations

| Refactoring Operation | Ranking Score |
|---|---|
| extractClass(GanttProject,GanttProjectExtracted,default,Nested) | 0.378 |
| moveMethod(getActiveCalendar,GanttProject,GanttProjectExtracted,default) | 0.369 |
| pushDownAttribute(bExpand,TaskImpl,GanttTask,default) | 0.368 |
| deleteMethod(getLength,GanttTask) | 0.3.51 |
| moveAttribute(myFakeCalendar,GanttProject,GanttProjectExtracted,private) | 0.349 |
| moveMethod(dependencyAdded,GanttGraphicArea,TaskDependencyEvent,default) | 0.220 |

**Figure 2. Refactorings recommended by DINAR.**



Dynamic INterActive Refactoring - Modify a Refactoring

Select Refactoring: moveMethod

Identifier: getActiveCalendar

Source Class: GanttProject

Target Class: GanttProjectExtracted

GanttProjectExtracted
GanttGraphicArea
GanttTree
GanttProjectBase
GanttProjectImpl

Visibility: [✓ As is] Private Protected Public

[✓] Update references

**Figure 3. Recommended target classes by DINAR for a move method refactoring to modify.**

### 2.2.2 *Dynamic Update of Refactorings Recommendation*

The input of this component is the new system after developers have made major changes to the original one, and the latest set of good refactoring solutions. The output is a new updated set of non-dominated refactoring solutions that are adapted to the new system. Then, the second component (innovization) is executed based on the output of this dynamic optimization component.

In the non-interactive refactoring approaches, the set of refactorings comprising the best chosen solution, needs to be fully executed in order to reach the solution's promised results. Thus, any changes applied to the set of refactorings such as changing or skipping some of them could deteriorate the resulting system's quality. This represents a major limitation to the existing work since it limits the developer's intervention to only choosing the preferred set of refactorings. In this context, the goal of this work is to cope with the above mentioned limitation by granting to the developer the possibility to customize the set of suggested refactorings either by accepting, modifying or rejecting them. The novelty of this work is its ability to take into account the developer's interaction, in terms of introduced customization to the existing solution, by conducting a local search to locate a new solution in the Pareto front that is nearest to the newly introduced changes. As the ranking of solutions is updated with after interaction, we use a quick multi-objective local search algorithm to locate the best solution in the new ranking and then update the refactoring solutions based on the new changes performed by developers. In fact, this feature avoids executing again the NSGA-II algorithm since the system to refactor is only slightly modified from the original one. Thus, the refactoring solutions will be quickly updated. To this end, we selected the indicator-based local search [1] with the same three objectives of the original problem.

This component is executed automatically to update the refactoring solutions but the developer can also select to run it at any time if it is found that the solutions should be updated.

Our approach has narrowed the gap that exists between automated search-based refactoring techniques and human intensive development by incorporating the search into the development process, and allowing the developer the choice of the best recommended refactoring that best matches their coding preferences.

## 3. EVALUATION STUDY

### 3.1 Research Questions

We defined four research questions to address in our experiments.

**RQ1:** Can DINAR help developers to refactor efficiently their systems and improve their quality?

**RQ2:** Can DINAR help developers to find useful refactorings quickly?

**RQ3:** How does DINAR perform compared to fully-automated and manual refactoring techniques?

**RQ4:** Can DINAR be useful for developers during the development of software systems?

To answer RQ1, it is important to validate the proposed refactoring solutions from both quantitative and qualitative perspectives. For the qualitative validation, we asked groups of potential users (software engineers) of DINAR to evaluate, manually, whether the suggested operations are feasible and efficient in improving the quality and their maintainability objectives. We define the metric "manual correctness" (*MC*) which corresponds to the number of meaningful operations over the total number of suggested operations. The *MC* metric is computed after the user interaction and is given by the following equation:

$$MC = \frac{\#\text{coherent/applied refactorings}}{\#\text{proposed refactorings}}$$

For the quantitative validation, we asked a group of developers to analyze and apply manually several refactoring types using Eclipse on several code fragments extracted from different systems where most of them correspond to code smells identified using in previous studies that should be refactored [4][6]. Then, we calculated precision and recall scores to compare between recommended refactorings by DINAR and those suggested manually:

$$RC_{recall} = \frac{|\text{suggested operations}| \cap |\text{expected operations}|}{|\text{expected operations}|} \in [0,1]$$

$$PR_{precesion} = \frac{|\text{suggested operations}| \cap |\text{expected operations}|}{|\text{suggested operations}|} \in [0,1]$$

To answer RQ2, we evaluated the time *T* required by developers to refactor several code fragments extracted from several systems with DINAR and without DINAR. This metric *T* includes all the activities from the execution time of the tool until finishing applying and validating all the refactorings (developers reach all the objectives of refactoring session such as fixing code smells). In addition, we defined a metric *PRT* that calculates the percentage of refactorings selected by developers from the top 5 refactorings of our recommendations list. We define also another metric *PSC* that corresponds to the percentage of selected code elements (methods, classes, etc.) from the top 5 ranked elements when developers want to modify a refactoring.

To answer RQ3, we compare our approach to three other existing fully-automated search-based refactoring techniques: Kessentini et al. [6], Ouni et al. [4] and Harman et al. [5]. We considered also in our experiments another popular design defects detection and correction tool JDeodorant [16] that does not use heuristic search techniques. We used the metrics *MC, RC, PR, NF, T* and *G* for the comparison.

To answer RQ4, we used a post-study questionnaire that collects the opinions of developers on DINAR.

## 3.2 Experimental setting

We used a set of well-known open-source Java projects and one project from our industrial partner, the Ford Motor Company. We applied our approach to four open-source Java projects: Xerces-J, JFreeChart, GanttProject, and JHotDraw. Xerces-J is a family of software packages for parsing XML. Table 1 provides some descriptive statistics about these six programs.

**Table 1.Statistics of the studied systems.**

| Systems | Release | # classes | KLOC | #Code smells | #Refactorings |
|---|---|---|---|---|---|
| Xerces-J | v2.7.0 | 991 | 240 | 91 | 83 |
| JHotDraw | v6.1 | 585 | 21 | 25 | 49 |
| JFreeChart | v1.0.9 | 521 | 170 | 72 | 88 |
| GanttProject | v1.10.2 | 245 | 41 | 49 | 56 |
| JDI-Ford | v5.8 | 638 | 247 | 83 | 94 |

Parameter setting influences significantly the performance of a search algorithm on a particular problem. For this reason, for each algorithm and for each system, we perform a set of experiments using several population sizes: 50, 100, 200, 300 and 500. The stopping criterion was set to 100,000 evaluations for all algorithms in order to ensure fairness of comparison. The other parameters' values were fixed by trial and error and are as follows: (1) crossover probability = 0.8; mutation probability = 0.5 where the probability of gene modification is 0.3; stopping criterion = 100,000 evaluations. For the indicator-based multi-objective local search [1], we used 30 iterations, number of neighbors is fixed to 5, number of genes to modify by mutation is 20% of the solution length and the number of fails is 10 (the algorithm is stopped when there is no improvement in the fitness functions during 10 iterations otherwise the stopping criterion is 30 iterations). Each algorithm is executed 31 times with each configuration and then comparison between the configurations is done using the Wilcoxon test. We used 23 refactoring types in our experiments, namely Add Parameter, Rename Method Encapsulate Collection/Downcast/Field, Collapse Hierarchy, Hide Method, Extract Class/Interface/Method/Subclass/Superclass, Inline Class/Method, Move Field/Method, Pull Up Field/Method, Push Down Field/Method and Remove Parameter/Setting Method.

Our study involved 8 subjects from the University of Michigan and 3 software engineers from Ford Motor Company and another large software company. Subjects include 2 master students in Software Engineering, 6 PhD students in Software Engineering and 3 software developers. All the subjects are volunteers and familiar with java development and refactoring. The experience of these subjects on Java programming ranged from 3 to 17 years.

We designed our study to answer our research questions. The subjects were invited to fill a questionnaire that aims to evaluate our suggested refactorings.

## 3.3 Results and Discussions

*Results for RQ1:* We reported the results of our empirical qualitative evaluation in Figure 4 (*MC*). As reported in Figure 4, the majority of the refactoring solutions recommended by DINAR were approved by developers and DINAR performed clearly better than all other existing approaches. On average, for all of the five studied systems, 85% of proposed refactoring operations were considered to be semantically feasible, improve software quality and appear useful to the software engineers. In addition to the qualitative evaluation, we automatically evaluate our approach without using the feedback of potential users to give more quantitative evaluation to answer RQ1. Thus, we compare the proposed operations with some expected ones defined manually by the different groups for several code fragments extracted from the five systems where most of them represent code smells detected using previous work [6]. We used also Ref-Finder [11] to identify operations that are applied between the program version under analysis and the next version. Figure 5 and Figure 6 summarize our finding. We found that a considerable number of proposed operations (an average of more than 75% in terms of precision and recall) that are already applied to the next version by software development team or suggested manually. The recall scores are higher than precision ones since we found that the manual suggested refactorings by developers are incomplete compared to the solutions provided by DINAR and this is was confirmed by the qualitative evaluation (*MC*).

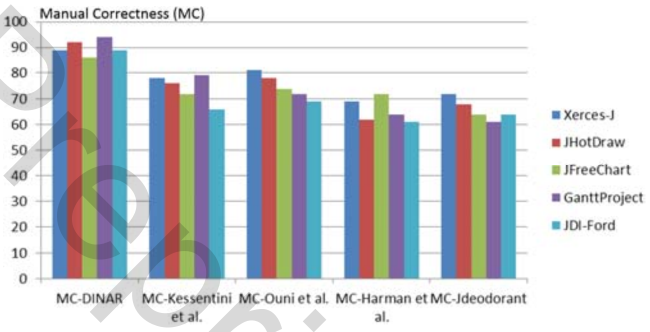


**Figure 4. Median manual correctness (*MC*) value over 31 runs on all the five systems using the different refactoring techniques with a 99% confidence level ($\alpha < 1\%$).**

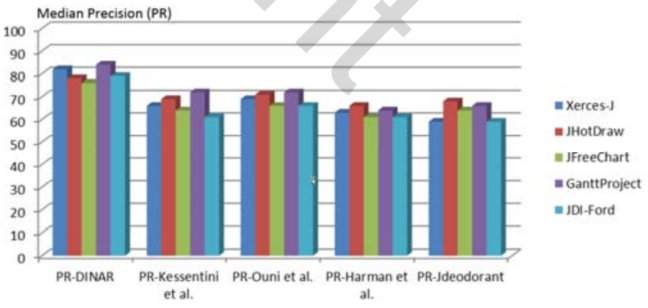


**Figure 5. Median precision (*PR*) value over 31 runs on all the five systems using the different refactoring techniques with a 99% confidence level ($\alpha < 1\%$).**

*Results for RQ2:* We evaluated the ability of DINAR to help software engineers to quickly find good refactorings. Figure 7 evaluates the average time *T* required by developers to finalize a refactoring session using DINAR. The average time is around 1 hour and a half including all the refactoring activities: execution of NSGA-II, interactive refactoring with the developer and the local search. There is a slight variation of the refactoring time required since it depends on the system to refactor. An average of one hour and a half is reasonable since developers will use DINAR during the development to maintain the quality of their systems and they can switch between modifying existing functionality and refactoring.

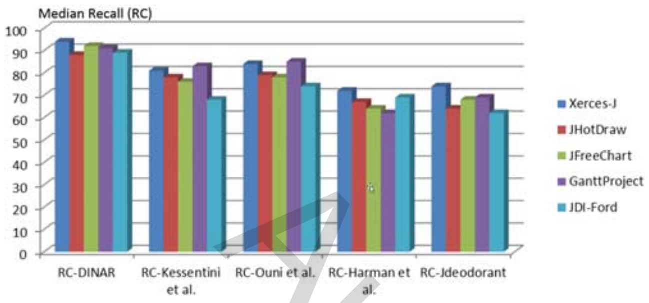


**Figure 6. Median recall (*RC*) value over 31 runs on all the five systems using the different refactoring techniques with a 99% confidence level (α < 1%).**

We considered two other metrics *PRT* and *PSC* to evaluate the efficiency of DINAR in ranking the refactorings and code elements to modify since this helps the developers to find quickly good refactoring to apply. Figure 8 shows that more than 92% of applied or modified refactorings (*PRT*) are among the top 5 recommended ones by DINAR at every iteration. In addition, most of the code elements recommended to developers when they modify a refactoring are among the top 5, also with an average of more than 90%. To conclude, it is clear that DINAR helps software engineers to quickly find good refactoring (this answers RQ2).

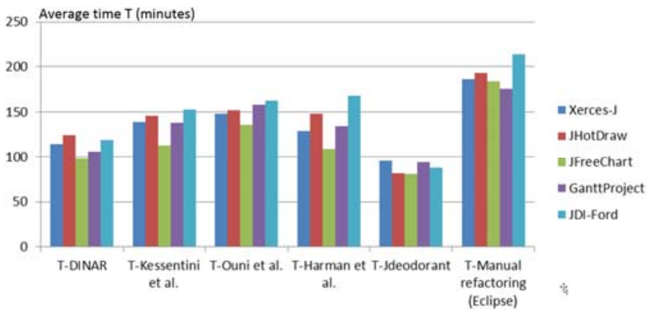


**Figure 7. Average time *T* (minutes) required by developers to finalize a refactoring session.**

*Results for RQ3:* Figures 4, 5, 6 and 7 confirm the better performance of DINAR compared to both fully automated and manual refactoring techniques. Figure 4 shows that DINAR provides significantly higher manual correctness results (*MC*) than all other approaches having *MC* scores respectively between 50% and 75% on average across the different systems. The same observation is valid for the quality gain, precision and recall as depicted in Figures 5 and 6. Figure 7 shows that DINAR can help developers to find suitable refactorings quicker than existing search-based refactoring approaches and manual approaches, with the notable exception of JDeodorant. This can be explained by the fact that JDeodorant is not using heuristic search but proposing a template solution to fix certain types of code smells. However, the time required to use DINAR is still comparable to JDeodorant, and is able to provide more effective refactoring solutions.

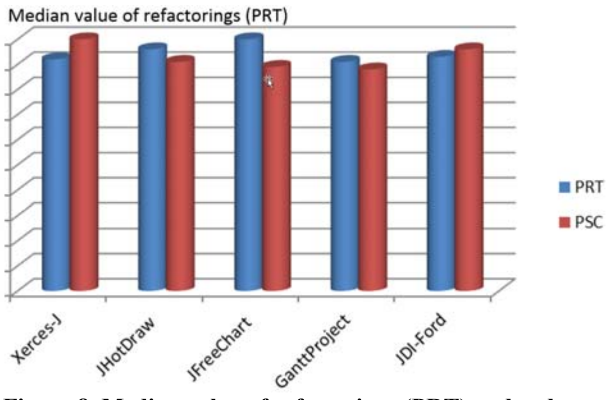


**Figure 8. Median value of refactorings (PRT) and code elements selected from the top5 on all the five systems.**

*Results for RQ4:* Most of the participants mention that DINAR is faster than manual refactoring since they spent a long time during manual refactoring to find the locations where the refactorings should be applied. For example, developers spend time in applying Extract Class to find the methods to move to the new created class, or when applying Move Method then it takes time to find the best target class by manual exploration of the source code. Thus, the developers liked the functionality of DINAR that helps them to modify a refactoring and find quickly the right parameters based on the recommendations. Furthermore, refactorings may affect several locations in the source code, which can be a time-consuming task to perform manually but can be performed in an instant using DINAR.

The participants found DINAR helpful for both *floss refactoring*, to maintain a good quality of the design and also *root canal refactoring* to fix some quality issues such as code smells. The developers justify their conclusions by the following interesting features in DINAR: a) the list of recommended refactorings helps them to choose the desired refactoring very quickly, b) DINAR offers them the possibility to modify the source code (to add new functionality) while doing refactoring since the list of recommendations are updated dynamically. So developers can easily switch between both activities: refactoring and modifying the source code to augment or amend existing functionality, c) DINAR allows developers to access all the functionality of the IDE (e.g. Eclipse). d) the suggested refactorings by DINAR can fix code smells (root canal refactoring) or improve some quality metrics (floss canal refactoring) due to the use of the multi-objective approach. Another important feature that the participants mention is that DINAR allows them to take advantage of using multi-objective optimization for software refactoring without the need to learn anything about optimization and exploring explicitly the Pareto front to select one "ideal" solution. The implicit exploration of the Pareto front in an interactive fashion represents an important advantage of DINAR along with the dynamic update of the recommended list of refactoring using innovization. In fact, developers had a lot of difficulties in using the multi-objective tool of Ouni et al. [4] to explore the Pareto front to find a good refactoring solution. In addition, they did not appreciate the long list of refactorings suggested in [4] since they want to take control of modifying and rejecting some refactorings. In addition, the validation of this long list of refactorings is time consuming. Thus, they appreciate that DINAR suggests refactoring one by one and updates the list based on developer feedback.

## 4. THREATS TO VALIDITY

Some threats need to be considered when interpreting our study results.

The first threat is the limited number of subjects and evaluated systems (11 participants and 5 systems), which externally threatens the generalizability of our results. In addition, we cannot conclude that DINAR can perform very well also when helping developers since our study was limited to the use of 23 types of refactorings and evaluating 8 types of code smells. Future replications of this study are necessary to confirm our findings.

The second threat is related to the variation of correctness and speed between the different groups when using DINAR and other tools such as JDeodorant. In fact, DINAR may not be the only reason for the improved performance because the subjects have different programming skills and familiarity with refactoring tools. However, developers were not assigned to groups randomly but according to their programming experience to reduce the gap between the different groups. Another threat concerns the data about the actual refactorings of the studied systems. In addition to the documented refactorings, we use Ref-Finder, which is known to be efficient. Indeed, Ref-Finder was able to detect refactoring operations with an average recall of 95% and an average precision of 79% [11]. To ensure the precision, we manually inspect the refactorings found by Ref-Finder by randomly selecting a set of detected changes and evaluate them by the participants of our experiments.

## 5. CONCLUSION AND FUTURE WORK

We proposed, in this paper a novel interactive recommendation tool, called DINAR, for software refactoring that dynamically adapts and suggests refactorings to developers based on their feedback and introduced code changes. To evaluate the effectiveness of DINAR, we conducted a human study with 11 software developers who evaluated the tool and compared it with the state-of-the-art refactoring techniques. Our evaluation results provide strong evidence that DINAR improves the applicability of software refactoring and proposes a novel way for software developers to refactor their systems.

Future work should validate DINAR with additional refactoring types, new systems and code smell types in order to draw conclusions about the general applicability of our methodology. We will also compare DINAR with other refactoring techniques [14][15]. Furthermore, in this paper, we only focused on the recommendation of refactorings. We are planning to extend the approach by automating the test and verification of applied refactorings. In addition, we will consider the importance of code smells during the correction step using previous code changes, class complexity, etc. Another future research direction related to our work is to adapt our interactive refactoring recommendation tool to several other software engineering problems such as software remodularization, change detection and the next release problem.

## ACKNOWLEDGEMENTS

This work was supported, in part, by the Institute for Advanced Vehicle Systems-Michigan grant and the Science Foundation Ireland grant 10/CE/I1855 to Lero - the Irish Software Engineering Research Centre.

## 6. REFERENCES

[1] Basseur, M. Liefooghe, A. Le, K. and Burke, E. K. 2012. The efficiency of indicator-based local search for multi-objective combinatorial optimisation problems. *Journal of Heuristics*. vol. 18, issue 2, pp 263-296.

[2] Fowler, M. Beck, K. Brant, J. Opdyke, W. and Roberts, D. 1999. Refactoring – Improving the design of existing code. Addison Wesley, ISBN 978-0201485677.

[3] Murphy-Hill, E. R. Parnin, C. and Black, A. P. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 5–18, 2012.

[4] Ouni, A. Kessentini, M. Sahraoui H. and Boukadoum, M. 2012. Maintainability Defects Detection and Correction: A Multi-Objective Approach. *Journal of Automated Software Engineering*, Springer. vol. 20, issue 1, pp 47-79.

[5] Harman, M. and Tratt, L. Pareto optimal search based refactoring at the design level. *GECCO07*. pp. 1106-1113.

[6] Kessentini, M. Kessentini, W. Sahraoui, H. Boukadoum, M. and Ouni, A. 2011. Design Defects Detection and Correction by Example, 19th IEEE International Conference on Program Comprehension. pp. 81-90.

[7] Deb, K. Pratap, A. Agarwal, S. and Meyarivan, T. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *TEC*. vol. 6, pp. 182–197.

[8] Harman, M. Mansouri, S. A. and Zhang, Y. Search-based software engineering: 2012. Trends, techniques and applications. *ACM Computing Surveys*, vol. issue 45, no. 1.

[9] Deb, K. and Srinivasan, A. Innovization: innovating design principles through optimization. 2006. *GECCO*. pp. 1629-1636.

[10] Bansiya, J. and Davis, C. G. 2002. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*. vol. 28. pp. 4–17.

[11] Prete, K. Rachatasumrit, N. Sudan, N. and Kim, M. 2010. Template-based reconstruction of complex refactorings. *Proceedings of the 26th IEEE International Conference on Software Maintenance*. pp. 1–10.

[12] Hall, M. Walkinshaw, N. McMinn, P. 2012. Supervised software modularization. *ICSM12*. pp. 472-481.

[13] Bavota, A. Carnevale, F. De Lucia, A. and Di Penta, M. 2012. Putting the Developer in-the-Loop: An Interactive GA for Software Re-modularization. *SSBSE*. pp. 75-89

[14] Lucia, Lo, D. Jiang, and L. Budi, A. 2012. Active refinement of clone anomaly reports. *International Conference on Software Engineering*. pp. 397-407.

[15] Gong, L. Lo, D. Jiang, L. and Zhang, H. 2012. Interactive fault localization leveraging simple user feedback. *ICSM2012*. pp. 67-76.

[16] Fokaefs, M. Tsantalis, N. Stroulia, E. and Chatzigeorgiou, A. 2011. JDeodorant: identification and application of extract class refactorings. *ICSE2011*. pp. 1037-1039.

[17] Kessentini, W. Kessentini, M. Sahraoui, H. Bechikh, S. and Ouni, A. 2014. A Cooperative Parallel Search-Based Software Engineering Approach for Code-Smells Detection, *IEEE Transactions on Software Engineering*, 2014, to appear.