Scheduling Refactoring Opportunities Using Computational Search

by

Nivin Tama

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science
(Software Engineering)
in The University of Michigan-Dearborn
2016

Master's Thesis Committee:

Assistant Professor Marouane Kessentini , Chair
Professor William Grosky
Associate Professor Bruce Maxim

To The Memory of My Father.

ACKNOWLEDGEMENTS

It is with a great joy that I reserve these few lines of gratitude and deep appreciation to all those who directly or indirectly contributed to the completion of this work:

I express my greatest gratitude to Dr. Marouane Kessentini, who dedicated all his wonderful time to collaborate, support and lead me to the end of this piece of work. His advices, dedication, availability, relevant comments, corrections and committeemen led to the success of this work.

I also express my greatest thanks to SBSE members who supported me with valuable feedback and always kindly encouraged me to succeed this project.

I thank all the lecturers of the software Engineering master degree who have used their valuable time to transmit the knowledge that help in putting this work together me.

Finally, I wish to express my deep gratitude and thank my family who has consistently expressed its unconditional support and encouragement.

All those who contributed in one way or another, to make this work, can be found here, the crowning of their efforts.

Thank you.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Maintaining a high-level code quality can be extremely expensive since time and monetary pressures force programmers to neglect improving the quality of their source code. Refactoring is an extremely important solution to reduce and manage the growing complexity of software systems. Developers often need to make trade-offs between code quality, available resources and delivering a product on time, and such management support is beyond the scope and capability of existing refactoring engines.

The problem of finding the optimal sequence in which the refactoring opportunities, such as bad smells, should be ordered is rarely studied. Due to the large number of possible scheduling solutions to explore, software engineers cannot manually find an optimal sequence of refactoring opportunities that may reduce the effort and time required to efficiently improve the quality of software systems. In this paper, we use bi-level multi-objective optimization to the refactoring opportunities management problem. The upper level generates a population of solutions where each solution is defined as an ordered list of code smells to fix which maximize the benefits in terms of quality improvements and minimize the cost in terms of number of refactorings to apply. The lower level finds the best sequence of refactorings that fixes the maximum number of code smells with a minimum number of refactorings for each solution (code smells sequence) in the upper level.

The statistical analysis of our experiments over 30 runs on 6 open source systems and 1 industrial project shows a significant reduction in effort and better improvements of quality when compared to state-of-art bad smells prioritization techniques. The manual evaluation performed by software engineers also confirms the relevance of our refactoring opportunities scheduling solutions.

Keywords— code smell; refactoring; optimization; bi-level; SBSE, scheduling.

CHAPTER 1: INTRODUCTION

Code-Smells, Design flaws, design defects, bad smells, or anomalies are altered aliases to signs of potential problems in source code of existing software systems. They are not bugs but they make a system difficult to change that may cause bugs [3]. Therefore, detecting and resolving code smells are critical steps to improve the quality of the system. However, in our project we will not focus on the detection phase. We assume that a series of code-smells have already been detected and need to be resolved.

In the context of code-smells resolution, refactoring has been used to resolve and clean up bad-smells in the source code by performing small restructuring changes without affecting its external behavior. XP-style and batch model are the two possible refactoring ways. In the first way, XP-style, small changes on few files are performed. On the other hand, a large system is thoroughly refactoring in on attempt in the other refactoring type, the batch model [3]. In this project, we aim to focus on the second type only.

In Batch model, different kinds of code-smells are targeted to be resolved. Moreover, multiple instances of each kind of code-smell are targeted as well. In what is known as instance and kind levels. Therefore, in a refactoring process, all code-smells detected a system at a specific state form a set. Each resolution sequence of this set of bad-smells may require different efforts because resolution of one kind may affect the resolution of other code-smells [3]. This is reflected on required code changes and their sequence. Research shows the significant role of

these code changes sequence on the cost and the quality of the refactoring process [2][3]. This strong correlation between code smells sequence and code changes needed inspired us to tackle ordering code-smell resolution as a bi-level problem.

The project aims to find the optimal sequence of code-smells resolution that minimizes the required refactoring code changes by solving it as Bi-level optimization problem.

The remainder of this thesis is as follows: Chapter 2 presents the relevant background, a motivating example for the presented work and an overview of the related work; Chapter 3 describes the search algorithm; an evaluation of the algorithm is explained and its results are discussed in Chapter 4. Finally, concluding remarks and future work are provided in Chapter 5

CHAPTER 2: RELATED WORK

We first detail some required background information to understand the problem addressed in this work, then we present a motivating example to illustrate the limitations of existing studies. Finally, we present an overview of existing work.

2.1 Problem statement

It has reported that software maintenance and evolution activities cost are more than 80% of total software cost [3]. Therefore, software refactoring has been used to improve readability and maintainability of software systems by correcting its detected code smells.

The sequence of addressing the detected code-smells reflects on the possible sequence of refactoring code changes. It may increase the number of code changes in the refactoring solution or decease it. Thu, there's a need to find the best order of the code-smells set of any system to be corrected.

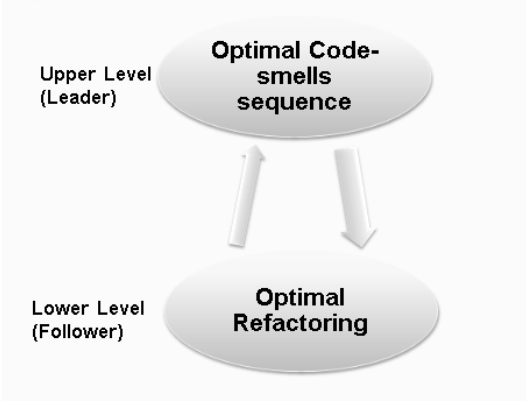

Fig 1. Bi-level optimization ilustration

Given that minimizing code changes when suggesting refactorings is important to reduce the effort and help developers understand the modified/improved design [3], the optimal sequence of code-smells correcting is the one that is corresponding to the most optimized refactoring solution.

In Other words, to find the optimal code-smell sequence we need to find the optimized refactorings for each possible solution. This nested nature of the proposed optimization problem goes along with the characteristics of bi-level optimization problem.

In this kind of problems, we find a nested optimization problem within the constraints of the outer optimization problem [1]. The outer optimization task is referred to as the upper level problem (or the leader) and the inner optimization task as the Lower level problem (or the follower).

To address the above mentioned optimization problem as bi-level problem, we consider finding optimized sequence of code-smells in upper level and finding the optimized refactorings in the lower level.

2.2 Motivating Example

The sequence of resolving the code smells carries important value in terms of how much effort will be spent overall on applying refactorings and how much regression issues will be newly introduced to the system due the fact that resolving one affects the detection or resolving the other one. Fixing one code smell before the other one may favor in maximizing the number of fixed code smells and minimizing the effort during applying the refactorings. We present a motivating example to demonstrate the importance of scheduling which code smells should be

fixed first and how our approach brings the optimized way to get benefit from changing the sequence of code smells. We used open source projects to validate our approach and one of the project among the 6 other projects is Gantt project. Applying our approach on Gantt project proved how effective it is to manage the sequence of code smells to be fixed. After the detection of code smells step, we randomly created a sequence of code smells to run our approach on. In Gantt project there is a duplication code in two different classes, TaskManagerImpl and GPTimeUnitStack. TaskManagerImpl implements the same logic in createLength function while GPTimeUnitStack has the logic in parseDuration function. These two functions are also suffering from long method code smell, there are some duplication code with in the function. These classes are picked specifically to demonstrate our motivating example. The point we want to prove is how effective it will be to change the sequence of in terms of effort to put for refactorings to be applied. One possible approach is to extract those functions from TaskManagerImpl and GPTimeUnitStack to another class where both classes will be calling the same function called “parseTime” and then fixing long method code smell in “parseTime” function by extracting the duplication code segment to another method and call the new method at the places where it is needed. On the other hand, if this was not the case, we would end up putting more effort in fixing two code smells, by fixing long method twice in parseDuration and createLength methods and then extract them to another class.

Hence the goal of our approach is to find a sequence which will prioritize the code smells in such a way that we will minimize the effort while maximizing the number of fixed code smells.

```
public TimeDuration parseDuration(String lengthAsString) throws ParseException {
      int state = 0;
      StringBuffer valueBuffer = new StringBuffer();
      Integer currentValue = null;
      TimeDuration currentLength = null;
      lengthAsString += " ";
      for (int i = 0; i < lengthAsString.length(); i++) {
        char nextChar = lengthAsString.charAt(i);
        if (Character.isDigit(nextChar)) {
          switch (state) {
        } else if (Character.isWhitespace(nextChar)) {
          switch (state) {
        } else {
          switch (state) {
          case 1:
            currentValue = Integer.valueOf(valueBuffer.toString());
          case 0:
            if (currentValue == null) {
              state = 2;
              valueBuffer.setLength(0);
          case 2:
            valueBuffer.append(nextChar);
            break;
          }
        }
      }
      if (currentValue != null) {
      return currentLength;
    }
```

Fig 2. Motivating Example. parseDuration (duplicated) method in GPTimeUnitStack class

```
public TimeDuration createLength(String lengthAsString) throws DurationParsingException {
  int state = 0;
  StringBuffer valueBuffer = new StringBuffer();
  Integer currentValue = null;
  TimeDuration currentLength = null;
  lengthAsString += " ";
  for (int i = 0; i < lengthAsString.length(); i++) {
    char nextChar = lengthAsString.charAt(i);
    if (Character.isDigit(nextChar)) {
      switch (state) {
    } else if (Character.isWhitespace(nextChar)) {
      switch (state) {
    } else {
      switch (state) {
      case 1:
        currentValue = Integer.valueOf(valueBuffer.toString());
      case 0:
        if (currentValue == null) {
          state = 2;
          valueBuffer.setLength(0);
      case 2:
        valueBuffer.append(nextChar);
        break;
      }
    }
  }
  if (currentValue != null) {
  return currentLength;
}
```

Fig 3. Motivating Example. createLength (duplicated) method in TaskManagerImpl class

```
case 2:
  TimeUnit timeUnit = findTimeUnit(valueBuffer.toString());
  if (timeUnit == null) {
    throw new ParseException(lengthAsString, i);
  }
  assert currentValue != null;
  TimeDuration localResult = createLength(timeUnit, currentValue.floatValue());
  if (currentLength == null) {
    currentLength = localResult;
  } else {
    if (currentLength.getTimeUnit().isConstructedFrom(timeUnit)) {
      float recalculatedLength = currentLength.getLength(timeUnit);
      currentLength = createLength(timeUnit, localResult.getValue() + recalculatedLength);
    } else {
      throw new ParseException(lengthAsString, i);
    }
  }
  state = 0;
  currentValue = null;
```

Fig 4. Duplication code with the function itself. This part will be extracted to fix long method

```
case 2:
  timeUnitFunction(valuBuffer.toString(), lengthAsString, timeUnit, currentValue);
  break;
```

Fig 5. timeUnitFunction is the extraction of the duplicated code

```
public void timeUnitFunction(String valueBuffer, String lengthAsString, int timeUnit, int currentValue)
{
  TimeUnit timeUnit = findTimeUnit(valueBuffer);
  if (timeUnit == null) {
    throw new ParseException(lengthAsString, i);
  }
  assert currentValue != null;
  TimeDuration localResult = createLength(timeUnit, currentValue.floatValue());
  if (currentLength == null) {
    currentLength = localResult;
  } else {
    if (currentLength.getTimeUnit().isConstructedFrom(timeUnit)) {
      float recalculatedLength = currentLength.getLength(timeUnit);
      currentLength = createLength(timeUnit, localResult.getValue() + recalculatedLength);
    } else {
      throw new ParseException(lengthAsString, i);
    }
  }
  state = 0;
  currentValue = null;
}
```

Fig 6. Extracted function, timeUnitFunction

2.3 Background and Related Work

Search-based techniques has been employed to solve Software engineering optimization problems [5]. By modeling a software engineering problem as a search problem, numerous approaches can be used to solving that problem. By browsing SBSE contributions in literature,

we find significant studies in the area of software maintenance and software refactoring [1][2]applying varieties of search techniques. However, to the best of our knowledge, Bi-level optimization have never been used to schedule code smell resolution [5]. The only SBSE usage of this search approach was by Sahin et al to contribute in code smell detection.

Fowler and Beck [4] have defined a list of code smell types that may be detected in the program source code with particular refactorings for each type. Several studies tried to prioritize code smell types based on different aspect. Ouni et al [6] automated code-smells prioritization considering importance, risk, severity, preferences of developers. However, their prioritization approach did take into consideration the complied refactoring. Whereas, Liu et al. [2] proposed an informal resolution sequence of nine kinds of bad smells based on their resolution effort. Though, we need a sequence that includes all detected code smells of any system (kind and instance levels) and leads to a minimal code changes refactoring solution. The main advantage of our bi-level optimization approach that it is not limited to some tested examples or some kinds of code smells.

Several researchers tried to schedule refactorings separately from code smells under different goals. We find scheduling refactorings to maximize refactoring effect [7] or to preserve the semantics of the design and its consistency with refactoring history [3]. Nevertheless, no previous work tried to use schedule of refactorings to guide the search of the optimized code smells resolution schedule.

Several SBSE researchers also studied the optimal schedule for next release planning under the umbrella of requirement management. Variety search techniques has been applied to prioritize requirements in regards of the problem of next release management [5]. Li et al. 2007 developed

an optimization tool integrating the requirements selection and scheduling for the release planning to find the optimal set of requirements with the maximum revenue to cater for budgetary constraints [8]. Next release problem was in subject in Zhang et. al. study to search for optimal/near optimal solutions for the allocation of the requirements in order to balance competing stake holder objectives in the next release.

Decision on which features to be included in the next release of the product is another example where next release approach has been taken place. Minimizing the cost as well as satisfying the customer was the two main objectives while deciding which features to add to the next release of the product. For this study, Durillo et.al. used NSGA-II as the reference algorithm with the comparison study of NSGA-II with other algorithms, MOCell, PAES and showed that NSGA-II outperformed other algorithms with the highest number of optimal solutions.

Another area where scheduling algorithms is highly used is bug-fixing. Bug-fixing is a very costly operation which is the largest contributor of software maintenance activities which takes almost 80% of all the costs. Most of the researchers are trying to develop optimal bug-fixing strategies considered as resource constrained scheduling problem with the help of scheduling algorithms. The main problem is finding the optimum way of assigning the bug-fixing activities to the developers based on their work load and skill sets. Xiao et al. proposing a multi-objective search-based resource scheduling method for optimum bug-fixing efforts. Their algorithm is able to suggest different efficient bug-fixing strategies based on three scenarios, having a strict/flexible deadline, having assigned different weights to severity and priority and the ability to foresee the resource requirements by adding virtual resources to meet the deadline. Anvik et al. [12] focuses on finding efficient way of bug-assignment efforts by considering the availability of resources and bug requirements. In their approach, support vector machine algorithm was

applied to suggest an assignment of a new bug report to a small number of developers. Defect effort schedule is proposed by Mockus et al. [11]. In their paper, they predict an optimum defect effort which is close to reality based on new feature changes by using a probability model for 11 software releases of large-scale real-time software system. Naïve Bayes classifier is applied to automatically assign bug reports to developers with 30% classification accuracy in [10].

Apart from above mentioned categories where scheduling algorithms are applied, there are some other cases where this approach is used due to its advantages to find the optimum sequence of solutions. For instance, scheduling/prioritization/sequencing approaches have been used in sequencing of requirements implementations, prioritization of work packages in project planning or sequencing of test cases for regression test case prioritization.

CHAPTER 3: SCHEDULING REFACTORING OPPERTUNITIES AS A BI-LEVEL OPTIMIZATION PROBLEM

In this chapter, we present an overview of our approach and then we provide the details of our problem formulation and the solution approach.

3.1 Approach Overview

Existing methods to solve bi- level optimization problems are classified into two families: (1) Classical methods and (2) evolutionary methods that are mainly Evolutionary Algorithms (EA) and simulated annealing (SA) [1]. The second type, which we adopted in our formulation, showed strong ability to solve similar bi-level problem in the phase of detecting code smells [1].We formulated our optimization problems in the upper and the lower levels as a Search Based Software Engineering SBSE problems.

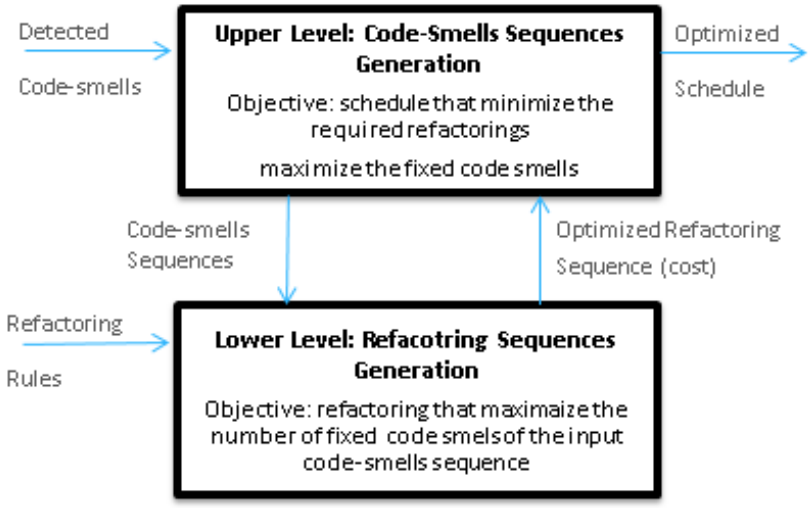

Fig 7. Bi-level optimization ilustration(detailed).

In order to perform the search for the optimal schedule for the code smells resolution, we formulated the multi objective optimization algorithm of the upper level. In this algorithm, the *solution* is the generated schedule, the *evaluation* of each solution is the output of running the lower level optimization problem for this specific schedule. The main objectives of the fitness function of this level are minimizing code changes and maximizing fixed code smells.

Similarly in the lower level, in order to perform the search for the optimal refactorings for a specific schedule as an input from the upper level, we solve it as a multi objective optimization algorithm. In this algorithm, the *solution* is the generated refactorings, the evaluation of each solution is the number of fixed code smells and the number of needed refeactorings. The main objectives of this level are minimizing code changes and maximizing fixed code smells under the constraint of the input schedule.

Fig 8 shows the algorithms of the upper level (a) and the lower level (b) and demonstrates the interaction between them.

Upper level algorithm: GAScheduleGeneration

Input: Detected code smells C Upper population size N1, Lower population size N2, Upper number of generations G1, Upper number of generations G2, Mutation factor M

Output: the best resolution sequence of code smells BSS

```
Begin
P0←  initialization(N1,C); /* random initialization */
For each CSS in P0 do     /* CSS is code smell sequence */
      BRS0←GARefactoringGeneration (CSS0, M, N2, G2); /*Call lower level*/
      CSS0 ← Evaluation (BRS0,)
End For
t ← 1;
While (t < G1) do
      Sort (Pt-1)
      i ←0;
      While (i<N1/0.25) do
          Pt← CSSi in Pt- 1;
          i←i+1;
       End While
       While (i<N1/0.5) do
          Pt← Variation (CSSi in Pt-1); /*mutate solution*/
          i←i+1;
       End While
       While (i<N1) do
          Pt← initialization(C); /*random solution generation*/
          i←i+1;
       End While
       i←N1/0.25;
       While (i<N1) do /*Evaluation for the new solutions in Pt*/
          BRSi ← GARefactoringGeneration (CSSi, M, N2, G2); /*Call lower level*/
           CSSi ← Evaluation (BRSi,); /* update solution fitness function based on lower level*/
       End While
End While
t ←t-1;
BSS ← FittestSelection (Pt); /*set the output of the algtithm*/
End
```

(a)

Upper level algorithm: GARefactoringGeneration

Input: Upper Level generated sequence ULS, Lower population size N, Lower number of generations G, Mutation factor M

Output: the best refactoring sequence for the upper level code smell sequence BRS

```
Begin
P0← initialization (N,ULS); /* random initialization */
P0← Evaluation(P0,ULS);
t ← 1;
While (t < G) do
      Sort (Pt-1)
      i ←0;
      While (i<N/0.25) do
          Pt← RSi in Pt- 1;  /*copy Refactoring Sequence RS*/
          i←i+1;
       End While
       While (i<N/0.5) do
          Pt← Variation (RSi in Pt-1); /*mutate refactoring sequence*/
          i←i+1;
       End While
       While (i<N) do
          Pt← initialization(ULS); /*random solution generation */
          i←i+1;
       End While
       i←N/0.25;
       While (i<N) do /*Evaluation for the new solutions in Pt*/
           RSi ← Evaluation (RSi); /* update solution fitness function */
          t ←t+1;
       End While
End While
BRS ← FittestSelection (Pt);
End
```

(b)

Fig 8. Pseudocode of the bil-level scheduling code smell resolution

CHAPTER 4: VALIDATION

In order to evaluate our approach for scheduling code-smells resolution using the proposed bi-level optimization approach, we conducted a set of experiments based on different open source systems: GanttProject, Xerces-J and JHotDraw. For each experiment, we analyzing the obtained results of our bi-level proposal by comparing them with an existing code-smells scheduling approach proposed by Liu et al [2] and with a random scheduling. In this section, we start by presenting our research questions and then show and discuss the obtained results.

4.1 Research Questions and Evaluation Metrics

We defined three research questions that address the applicability, performance in comparison to existing code smells scheduling approaches, and the usefulness of bi-level optimization approach. The three research questions are as follows.

RQ1: How does the bi-level scheduling approach perform to give a possible order of code smells resolution?

RQ2: How does the bi-level scheduling approach perform comparing to the existing code-smells scheduling approach proposed by Liu et al?

RQ3: How does the bi-level scheduling approach perform comparing to a random scheduling of code smells resolution?

4.2 Experimental Setting

The chosen systems were medium sized system (LOC 70,000 to 200,000) their quality deficit index is from 1.9 to 11.6 as the number of infections or code smells is ranged between 44 and 393. Table 1 presents the characteristics of each system.

System	Release	#Smells	KLOC
GanttProject	1.10.2	79	91,331
JhotDraw	6.1	46	71,708
XercessJ	2.7.0	393	200,458

Table 1. Experiments Systems

4.3 Results

This section describes and discusses the results obtained for the different research questions. By feeding part of the detected code smells lists to our system and got its proposed optimized order with its percentage ability of fixing code smells and the required effort or refactorings to the code.

Subject System:	XercessJ (35 Code Smells)	Ganttproject (44 Code Smells)	JHotDraw (32 Code Smells)
Bi-level Scheduling	Refactoring 52 Fixed (62%)	Refactoring 62 Fixed (60%)	Refactoring 37 Fixed (71%)
Random Resolution	Refactoring 90 Fixed (34%)	Refactoring 125 Fixed (15%)	Refactoring 87 Fixed (15%)
Ordered Resolution	Refactoring 89 Fixed 11 (31%)	Refactoring 124 Fixed 8 (18%)	Refactoring 89 Fixed6 (18%)

Table 2. Experiments Results

Table 2 shows the results proposed by our algorithm for each subject system as long as the results of a random or resolution order. In addition, it shows the evaluation of the order proposed by [2].

As Fig. 9 demonstrate, it's noticeable that our algorithm showed a significant improvement in the total fixed code smells and the required refactorings. To discuss our results further, we present the answers for our research question:

1. Results for RQ1:
   The schedules proposed by out bi-level optimization algorithm in all conducted experiments showed high fixing code smells impact (60-71%) with minimum effort.

2. Results for RQ2:
   In comparison with a random resolution order, we reached a 50% increase in the number of fixed code smells with a 60% decrease in required refactorings

3. Results for RQ3:
   In comparison with the schema order proposed by Lui et al [2], our approach's results in the conducted experiments showed 42-51%increase in the number of fixed code smells with a 50-70% decrease in required refactorings.

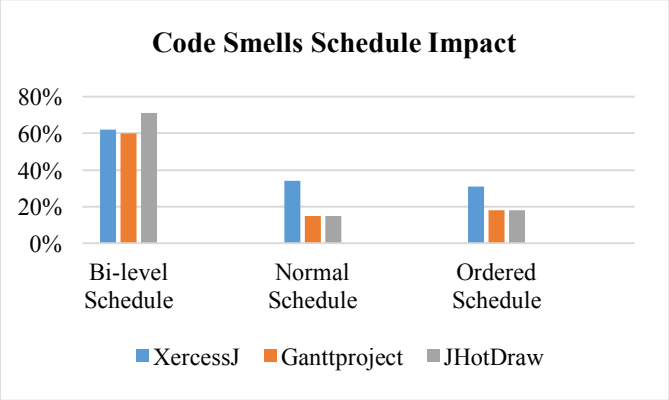

Fig 9. Experiments Results

CHAPTER 5: CONCLUSION

Cleaning large systems from all its defects is challenging and error provoking. Considering numerous detected defects, there are hundreds of possible schedules. Each order leads to a better or worse impact. The optimal order that leads to maximum fixed code smells with the minimum code changes, refactoring.

In this Project, we aimed to approach the scheduling of detected code smells before correcting them as a bi-level optimization problem. In other words, we attempted to find the optimal order of code smells as the main optimization problem depending on the nested follower optimization problem, finding the optimal refactorings.

The main advantage of bi-level optimization approach that it is not limited to some tested examples analysis or some kinds of code smells. It is general and can be applied to order any software system code-smells to resolve them.

Our approach, showed significant efficiency by proposing resolution schedules that led to increase fixing code smells percentage and decrease required refactorings.

REFERENCES

[1] Sahin, Dilan, et al. "Code-smell detection as a bilevel problem." ACM Transactions on Software Engineering and Methodology (TOSEM) 24.1 (2014): 6.

[2] Liu, Hui, et al. "Schedule of bad smell detection and resolution: A new way to save effort." Software Engineering, IEEE Transactions on 38.1 (2012): 220-235.

[3] Ouni, Ali, et al. "Multi-criteria code refactoring Using Searched Based Software Engineering: An Industrial Study"

[4] Martin Fowler, Kent Beck, John Brant, William Opdyke and Don Roberts. 1999. Refactoring – Improving the Design of Existing Code, 1st ed. Addison-Wesley.

[5] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. 2012. Search-based software engineering: Trends, techniques and applications. ACM Comput. Surv. 45, 1, Article 11 (December 2012), 61 pages.

[6] Ali Ouni, Marouane Kessentini, Slim Bechikh, Houari Sahraoui, "Prioritizing code-smells correction tasks using chemical reaction optimization", Software Quality Journal (SQJ), Springer, volume 23, pp. 1-39, 2014.

[7] Liu, Hongying, et al. "Conflict-aware schedule of software refactorings."Software, IET 2.5 (2008): 446-460.

[8] Li, Chen, et al. "Integrated requirement selection and scheduling for the release planning of a software product." Requirements Engineering: Foundation for Software Quality. Springer Berlin Heidelberg, 2007.93-1

[9] Stefan Gueorguiev , Mark Harman , Giuliano Antoniol, Software project planning for robustness and completion time in the presence of uncertainty using multi objective search based software engineering, Proceedings of the 11th Annual conference on Genetic and evolutionary computation, July 08-12, 2009, Montreal, Québec, Canada

[10] D. Cubranic and G. Murphy, "Automatic bug triage using text categorization," in Procs. of the 16th Int. Conf. on SW Eng. and Knowledge Eng., 2004.

[11] A. Mockus, D. M. Weiss, and P. Zhang, "Understanding and predicting effort in software projects," in Procs. of the 25th Int. Conf. on SW Eng. IEEE Computer Society, 2003.

[12] J. Anvik, L. Hiew, and G. Murphy, “Who should fix this bug?” in Procs. of the Int. Conf. on Software Engineering, 2006.

[13] Hui Liu, Limei Yang, Zhendong Niu, Zhyi Ma, and Weizhong Shao. 2009. Facilitating software refactoring with appropriate resolution order of bad smells. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering* (ESEC/FSE '09). ACM, New York, NY, USA, 265-268.

[14] G. Antoniol, M. Di Penta and M. Harman, "Search-based techniques applied to optimization of project planning for a massive maintenance project," *21st IEEE International Conference on Software Maintenance (ICSM'05)*, 2005, pp. 240-249.

[15] Matthieu Basseur, Arnaud Liefooghe, K. Le, Edmund K. Burke: The efficiency of indicator-based local search for multi-objective combinatorial optimisation problems. J. Heuristics 18(2): 263-296 (2012)

[16] Gabriele Bavota, Bernardino De Carluccio, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, Orazio Strollo: When Does a Refactoring Induce Bugs? An Empirical Study. SCAM 2012: 104-113

[17] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E. Johnson, Danny Dig: A Comparative Study of Manual and Automated Refactorings. ECOOP 2013: 552-576

[18] Danny Dig: A Refactoring Approach to Parallelism. IEEE Software 28(1): 17-22 (2011)

[19] W. J. Brown, R. C. Malveau, W. H. Brown, and T. J. Mowbray, “Anti patterns: refactoring software, architectures, and projects in crisis,” John Wiley & Sons, 1998, ISBN 978-0471197133.

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

[21] W. F. Opdyke, R. E. Johnson, Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems. In Proceedings of the Symposium on Object Oriented Programming Emphasizing Practical Applications (SOOPPA), September 1990.

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

[23] E. R. Murphy-Hill and A. P. Black, Refactoring tools: Fitness for purpose, IEEE Software, vol. 25, no. 5, pp. 38–44, 2008.

[24] Xi Ge, E. R. Murphy-Hill, BeneFactor: a flexible refactoring tool for eclipse. OOPSLA Companion 2011: 19-20

[25] D. Silva, R. Terra, M. T. Valente, Recommending Automated Extract Method Refactorings, Internation Conference on Program Comprehension ICPC 2014.

[26] A. Ouni, M. Kessentini, H. Sahraoui and M. Boukadoum, Maintainability Defects Detection and Correction: A Multi-Objective Approach. J. of Autmated Software Engineering, Springer, 2012.

[27] M. Harman, and L. Tratt, Pareto optimal search based refactoring at the design level, In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'07), pp. 1106-1113, 2007.

[28] M. O'Keeffe, and M. O. Cinnéide, Search-based Refactoring for Software Maintenance. J. of Systems and Software, 81(4), 502–516.

[29] T. Mens, T. Tourwé: A Survey of Software Refactoring. IEEE Trans. Software Eng. 30(2), pp. 126-139, 2004.

[30] M. Kessentini, W. Kessentini, H. Sahraoui, M. Boukadoum, A. Ouni, Design Defects Detection and Correction by Example, 19th IEEE ICPC11, pp. 81-90, Canada, 2011.

[31] A. Ouni, M. Kessentini, H. Sahraoui, M. Hamdi: The use of development history in software refactoring using a multi-objective evolutionary algorithm. GECCO 2013

[32] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, A fast and elitist multiobjective genetic algorithm: NSGA-II, IEEE Trans. Evol. Comput., vol. 6, pp. 182–197, Apr. 2002.

[33] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," ACM Computing Surveys, vol. 45, no. 1, 61 pages.

[34] Kalyanmoy Deb, Aravind Srinivasan: Innovization: innovating design principles through optimization. GECCO 2006: 1629-1636

[35] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, JDeodorant: identification and application of extract class refactorings. International Conference on Software Engineering (ICSE), pp. 1037-1039, 2011.

[36] J. Bansiya and C. G. Davis. A hierarchical model for object-oriented design quality assessment. IEEE Trans. Softw. Engg., 28(1): 4–17, 2002.

[37] Ali Ouni, Marouane Kessentini, Houari A. Sahraoui, Mohamed Salah Hamdi: Search-based refactoring: Towards semantics preservation. ICSM 2012: 347-356

[38] Arcuri A. and Fraser G., 2013. Parameter tuning or default values? An empirical investigation in search-based software engineering, Empirical Software Engineering, 18(3).

[39] Arcuri, A. and Briand, L. C. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In Proceedings of the 33rd International Conference on Software Engineering (ICSE '11). ACM, New York, NY, USA, 1-10. DOI=10.1145/1985793.1985795.

[40] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim, "Template-based reconstruction of complex refactorings," in Proceedings of the 26th IEEE International Conference on Software Maintenance. pp. 1–10. Timisoara, Romania, IEEE CS Press, 2010.

[41] B. Du Bois, S. Demeyer, and J. Verelst, Refactoring—Improving Coupling and Cohesion of Existing Code, Proc. 11th Working Conf. Reverse Eng (WCRE). pp. 144-151, 2004.

[42] Emerson R. Murphy-Hill, Andrew P. Black: Programmer-Friendly Refactoring Errors. IEEE Trans. Software Eng. 38(6): 1417-1431 (2012)

[43] Emerson R. Murphy-Hill, Andrew P. Black: Breaking the barriers to successful refactoring: observations and tools for extract method. ICSE 2008: 421-430

[44] S. R. Foster, W. G. Griswold, and S. Lerner. WitchDoctor: IDE support for real-time auto-completion of refactorings. In Proceedings of the International Conference on Software Engineering, pages 222–232, 2012.

[45] X. Ge, Q. L. DuBose, and E. Murphy-Hill. Reconciling manual and automatic refactoring. In Proceedings of the International Conference on Software Engineering, pages 211–221, 2012.

[46] L. Tahvildari, K. Kontogiannis, A metric-based approach to enhance design quality through meta-pattern transformation. In: Proceedings of the 7st European Conference on Software Maintenance and Re-engineering , Benevento, Italy, pp. 183–192, 2003.

[47] Y. Kataoka, M. D. Ernst, W. G. Griswold, and D. Notkin, Automated support for program refactoring using invariants, in Int. Conf. on Software Maintenance (ICSM), pp. 736–743, 2001.

[48] O. Seng, J. Stammel, and D. Burkhart, Search-based determination of refactorings for improving the class structure of object-oriented systems, In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'06), pp. 1909–1916, 2006.

[49] D. Fatiregun, M. Harman, and R. Hierons. Evolving transformation sequences using genetic algorithms. In SCAM 04, pages 65–74, Los Alamitos, California, USA, Sept. 2004. IEEE Computer Society Press.

[50] H. Kilic, E. Koc, and I. Cereci. Search-based parallel refactoring using population-based direct approaches. In Proceedings of the Third international Conference on Search Based Software Engineering, SSBSE'11, pages 271–272, 2011.

[51] M. Ó Cinnéide, L. Tratt, M. Harman, S. Counsell, and I. H. Moghadam, Experimental Assessment of Software Metrics Using Automated Refactoring, Proc. Empirical Software Engineering and Management (ESEM), pages 49-58, September 2012.

[52] Mathew Hall, Neil Walkinshaw, Phil McMinn: Supervised software modularization. ICSM 2012: 472-481

[53] Abriele Bavota, Filomena Carnevale, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto: Putting the Developer in-the-Loop: An Interactive GA for Software Re-modularization. SSBSE 2012: 75-89

[54] Lucia, David Lo, Lingxiao Jiang, Aditya Budi: Active refinement of clone anomaly reports. ICSE 2012: 397-407

[55] Liang Gong, David Lo, Lingxiao Jiang, Hongyu Zhang: Interactive fault localization leveraging simple user feedback. ICSM 2012: 67-76