UNIVERSITÉ DE MONTRÉAL

AUTOMATED IMPROVEMENT OF SOFTWARE DESIGN BY SEARCH-BASED REFACTORING

RODRIGO MORALES ALVARADO
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION
DU DIPLÔME DE PHILOSOPHIÆ DOCTOR
(GÉNIE INFORMATIQUE)
DÉCEMBRE 2017

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée :

AUTOMATED IMPROVEMENT OF SOFTWARE DESIGN BY SEARCH-BASED REFACTORING

présentée par : MORALES ALVARADO Rodrigo
en vue de l'obtention du diplôme de : Philosophiæ Doctor
a été dûment acceptée par le jury d'examen constitué de :

Mme NICOLESCU Gabriela , Doctorat, présidente
M. KHOMH Foutse, Ph. D., membre et directeur de recherche
M. ANTONIOL Giuliano, Ph. D., membre et codirecteur de recherche
M. CHICANO Francisco, Ph. D., membre et codirecteur de recherche
M. MERLO Ettore , Ph. D., membre
Mme SARRO Federica, Ph. D., membre externe

DEDICATION

To my beloved Mother MaryTony for encouraging me to follow my dreams, and support me in my darkness hours, and to my grand-father Raul who planted the seed of love in my early years, and specially to all that believe in God and found inspiration and hope in its teachings.

ACKNOWLEDGEMENTS

I want to thank my supervisors Foutse Khomh, Francisco Chicano and Giuliano Antoniol for leading me through the winding road of research, for the economical and emotional support, and for sharing an invaluable amount of time to revise my work for the duration of my program. For sure, I would not be able to complete my program without their opportune advice and care.

I would like to thank my mother and sister for their unconditional support and love through all of these years. Your words always motivate me to be a better person.

To Nazli, whose words of relief help me to endure during the exhaustive hours of works, and who lend her hears to listen difficulties, even if they were unrelated to her major and hard to understand.

I also like to thank my colleagues who collaborate with me during my studies at Polytechnique, some of them as co-authors in some papers, and some others as heroes anonymous. Not less important are all the reviewers that bring new ideas and propose improvements to all my papers published that are the base of this dissertation.

Finally, I like to thank CONACyT who economically support me for more than a year at the end of my program.

RÉSUMÉ

Le coût de maintenance du logiciel est estimé à plus de 70% du coût total du système, en raison de nombreux facteurs, y compris les besoins des nouveaux utilisateurs, l'adoption de nouvelles technologies et la qualité des systèmes logiciels. De ces facteurs, la qualité est celle que nous pouvons contrôler et continuellement améliorer pour empêcher la dégradation de la performance et la réduction de l'efficacité (par exemple, la dégradation de la conception du logiciel). De plus, pour rester compétitive, l'industrie du logiciel a raccourci ses cycles de lancement afin de fournir de nouveaux produits et fonctionnalités plus rapidement, ce qui entraîne une pression accrue sur les équipes de développeurs et une accélération de l'évolution de la conception du système. Une façon d'empêcher la dégradation du logiciel est l'identification et la correction des anti-patrons qui sont des indicateurs de mauvaise qualité de conception. Pour améliorer la qualité de la conception et supprimer les anti-patrons, les développeurs effectuent de petites transformations préservant le comportement (c.-à-d., refactoring). Le refactoring manuel est coûteux, car il nécessite (1) d'identifier les entités de code qui doivent être refactorisées ; (2) générer des opérations de refactoring pour les classes identifiées à l'étape précédente ; (3) trouver le bon ordre d'application des refactorings générés, pour maximiser le bénéfice pour la qualité du code et minimiser les conflits. Ainsi, les chercheurs et les praticiens ont formulé le refactoring comme un problème d'optimisation et utilisent des techniques basées sur la recherche pour proposer des approches (semi) automatisées pour le résoudre.

Dans cette thèse, je propose plusieurs méthodes pour résoudre les principaux problèmes des outils existants, afin d'aider les développeurs dans leurs activités de maintenance et d'assurance qualité.

Ma thèse est qu'il est possible d'améliorer le refactoring automatisé en considérant de nouvelles dimensions : (1) le contexte de tâche du développeur pour prioriser le refactoring des classes pertinentes ; (2) l'effort du test pour réduire le coût des tests après le refactoring ; (3) l'identification de conflit entre opérations de refactoring afin de réduire le coût de refactoring ; et (4) l'efficacité énergétique pour améliorer la consommation d'énergie des applications mobiles après refactoring.

Je propose quatre méthodes : (1) ReCon, qui exploite le contexte de tâche du développeur. Cette méthode permet de réduire une médiane de 50% des anti-patrons au cours des tâches de codage régulières, sans perturber le flux de travail du développeur dans les sessions de refactoring dédiées. (2) RePOR, une méthode de refactoring capable de réduire l'effort de

refactoring et le temps d'exécution de 80%. (3) TARF, une méthode de refactoring qui prend en compte l'effort requis pour tester le logiciel. Les résultats montrent que nous pouvons réduire une médiane de 48% de l'effort de test après la refactorisation. (4) EARMO, une approche automatisée pour les applications mobiles, capable de supprimer 84% des anti-patrones et d'allonger la durée de vie de la batterie jusqu'à 29 minutes pour une application multimédia exécutant en continu un scénario typique.

J'ai appliqué et validé les méthodes proposées sur plusieurs systèmes open source pour démontrer leur impact sur la qualité de la conception en utilisant des modèles de qualité bien connus, et les commentaires de certains auteurs des systèmes étudiés.

Mots-clés : Maintenance de logiciels, qualité de conception de logiciel, anti-patrons, refactoring, efficacité énergétique.

ABSTRACT

Software maintenance cost is estimated to be more than 70% of the total cost of system, because of many factors, including new user's requirements, the adoption of new technologies and the quality of software systems. From these factors, quality is the one that we can control and continually improved to prevent degradation of performance and reduction of effectiveness (*a.k.a.* design decay). Moreover, to stay competitive, the software industry has shortened its release cycles to deliver new products and features faster, which results in more pressure on developer teams and the acceleration of system's design evolution. One way to prevent design decay is the identification and correction of anti-patterns which are indicators of poor design quality. To improve design quality and remove anti-patterns, developers perform small behavior-preserving transformations (*a.k.a.* refactoring). Manual refactoring is expensive, as it requires to (1) identify the code entities that need to be refactored; (2) generate refactoring operations for classes identified in the previous step; (3) find the correct order of application of the refactorings generated, to maximize the quality effect and to minimize conflicts. Hence, researchers and practitioners have formulated refactoring as an optimization problem and use search-based techniques to propose (semi)automated approaches to solve it.

In this dissertation, we propose several approaches to tackle some of the major issues in existing refactoring tools, to assist developers in their maintenance and quality assurance activities.

Our thesis is that it is possible to enhance automated refactoring by considering new dimensions: (1) developer's task context to prioritize the refactoring of relevant classes; (2) testing effort to improve testing cost after refactoring; (3) refactoring's conflict awareness to reduce refactoring effort; and (4) energy efficiency to improve energy consumption of mobile applications after refactoring.

We propose four approaches: (1) ReCon, which leverages developer's task context to prioritize the refactoring of classes that are relevant to the developer's activity. Using ReCon, developers can remove a median of 50% of anti-patterns during regular coding tasks, without disrupting their workflow. (2) RePOR, for an efficient refactoring scheduling, which results in a reduction of refactoring effort and execution time by 80%. (3) TARF controls for the testing effort while refactoring. Results show that TARF can reduce a median of 48% of the testing effort of a system after refactoring. (4) EARMO, is an automated approach for the refactoring of mobile applications, which is able to remove 84% of anti-patterns and extend the battery life of devices by up to 29 minutes (for a multimedia app running continuously a

typical scenario).

We apply and validate our proposed approaches on several open-source systems to demonstrate their impact on design quality using well known quality models, and feedback from some authors of the systems studied.

Keywords: Software maintenance, design quality, anti-patterns, refactoring, search-based software engineering, energy efficiency, task context, testing effort.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

EMO	Evolutionary Multiobjective Optimization
GA	Genetic Algorithm
EARMO	Energy-Aware Refactoring approach for MObile apps
EAs	Evolutionary Algorithms
ECG	Enhanced Call-graph
HV	Hypervolume
IDE	Integrated Development Environment
IH	Interaction Histories
MaDUM	Minimal Data members Usage Matrix
MO	Multiobjective optimization
MOCell	Multiobjective Cellular Genetic Algorithm
NSGA-II	Non-dominated sorting genetic algorithm
OO	Object-Oriented
OSS	Open-Source Systems
PADL	Pattern and Abstract-level Description Language
QMOOD	Quality Model for Object-Oriented Design
ReCon	Refactoring approach based on task Context
RePOR	Refactoring approach based on Partial Order Reduction
SA	Simulated Annealing
SAD	Software Architectural Defects
SBSE	Search-Based Software Engineering
SPEA2	The Strength Pareto Evolutionary Algorithm 2
TARF	Testing-Aware ReFactoring approach
VCS	Version Control System
VNS	Variable Neighborhood Search

CHAPTER 1 INTRODUCTION

1.1 Research Context: Software Quality

Software maintenance is defined as the process of modifying a software system in order to add new features, correct faults or improve functionality. In previous studies [1], the cost of software maintenance has been estimated to be more than 70% of the total cost of a software. Thus, researchers have focused their effort on studying the quality of software systems and proposed metrics and methodologies to assess it.

As systems evolve, they tend to grow in complexity and degrade in effectiveness [2], unless the quality of the systems is controlled and continually improved. This phenomenon is known as design decay. When the design of a system is poor, new changes to the system often degrade quality instead of improving it. Some indicators of poor design quality are anti-patterns [3], which are symptoms of bad design-choices that makes it hard to understand, modify and extend a software system. The presence of anti-patterns in a system increases the risk of faults [4] and the cost of future maintenance. They are not technically incorrect, but result in negative consequences in the long run [5]. An example of anti-pattern is *Spaghetti Code*, which is a class without structure that declares long methods without parameters [6]. This anti-pattern depicts an abuse of procedural programming in object-oriented systems, that prevents code reuse. In a previous study, Bavota et al. [7] found that industrial developers assign a very high severity level to this anti-pattern. Another example of anti-pattern is the *Lazy class*, which is a class with a few methods and a low complexity that does not "pay off" its inclusion in the system. Abbès et al. [8] showed that anti-patterns affect the understandability of systems and Khomh et al. [4] found that there is a strong correlation between the occurrence of anti-patterns and the fault-proneness of source code files. In addition to this, the length of time that an anti-pattern can remain in a system after its introduction for the first time is uncertain, though there is evidence to suggest that they tend to linger for several releases [9, 10]. It is therefore advisable to correct anti-patterns on a regular basis, to avoid their negative impact on future releases of the system.

To combat design decay and remove anti-patterns, practitioners perform refactoring [11, 12], which is the process of reorganizing, and rewriting existing code, without altering its original behavior. Several studies have assessed the benefits of refactorings, both in academic and industrial settings. Rompaeyet al. [13] report a significant reduction over 50% of memory usage, and 33% startup time improvement in a telecommunication company after performing refactoring. In another study including several revisions of an open source system, Soetens

and Demeyer [14] found that most refactorings tend to reduce the Cyclomatic complexity [15], especially when they target duplicate code. Du Bois et al. [16], performed an experiment with students and observed that refactoring God classes improves the comprehensibility of the source code. In an industrial setting at Microsoft, Kim et al. [17] found that modules that underwent refactoring have less inter-module dependencies and less post-release faults.

The drawback to refactoring is the high cost incurred when performing it manually. First, it is necessary to identify classes and code fragments that need to be improved. Then, we determine which refactorings should be applied to the identified places, considering that some refactorings may enable or block further refactorings. Hence, this selection determines the quality improvement effect achieved in a refactoring session. Since quality is a difficult attribute to measure (it can be interpreted in different ways), companies do not assign enough resources to improve it [18, 19]. Moreover, big organizations and companies like Mozilla and Google have shorted their release cycles to be more competitive, reducing even more the already limited budget assigned for performing software maintenance tasks [20].

To provide a solution for the complex task of manual refactoring, researchers have formulated refactoring as a combinatorial optimization problem and applied search-based techniques to solve it [21]. The idea is to generate a sequence of refactoring operations that should be applied by developers to improve software design quality. Some of these works formulated refactoring as a single-objective optimization problem in which the main goal is to improve design quality [22, 23, 24, 25]. Other works have considered additional objectives like controlling code semantic [26] and using development history [27]. However, the use of development history and semantic information are not the only sources of information that one can use to support refactoring activity. We believe that search-based refactoring can be improved by considering other dimensions like developer's task context, testing effort, energy efficiency, etc.

1.2 Problem Statement

Some problems with existing semi-automated refactoring approaches are the following.

1. Developers have to review a long list of refactoring operations due to the lack of context. For example, developers working in a specific module or subproject might not be interested in modify classes out of their context, or beyond their code ownership.
2. The cost of applying the refactoring solutions proposed is too high, compared to the possible benefits for the long term. For example, if the refactoring solutions proposed by a tool, require to dramatically change the existing design, in a sense that the devel-

oper team is not longer familiar with it, or add excessive effort to maintain and test, developers will dismiss the proposed solutions.

3. An approach requires to provide complex inputs, beside the system to be refactored. For example, a tool that require developers to gather an oracle of bad design examples to generate useful solutions it is likely to discourage developers to adopt it. Indeed, Murphy et al. found that developers tend to use default parameters of refactoring tools and avoid tools with complex settings and inputs [28].
4. The refactorings solutions proposed do not consider all critical concerns of the software system domain. For example, energy concerns which are very relevant in embedded and mobile systems domain. Hence, refactoring solutions that do not control for energy concerns, are likely to be dismissed by developers working on these domains.

Note that this is not an exhaustive list, but it gives an indication of several points for improvement.

In this dissertation, we proposed different approaches to improve search-based refactoring with respect to various dimensions (*e.g.,* developer's task context, refactoring effort, testing effort, energy efficiency) to overcome the problems mentioned above. We aim to make refactoring cheaper and useful to software developers and maintainers. Thus, our thesis is:

1.2.1 Thesis

Search-based refactoring can be readily used for automatically improving software design quality while (1) cutting the cost of anti-patterns detection and correction; (2) making more efficient use of computational resources than existing approaches; and (3) providing useful solutions from developer's perspective

To reduce the number of refactoring candidates that a developer has to review from an automated approach, we propose the use of developers' task context to prioritize the refactoring of relevant classes during maintenance activities and in code review sessions. This is a more natural approach and follows the recommendations of agile development practices like XP (extreme programming), that encourage developers to perform refactoring incrementally [29] and do not postpone it until becomes very expensive.

To reduce the cost of refactoring while preserving the quality improvement effect, we propose to define a refactoring order that considers conflicts and removes redundant solutions.

Moreover, after developers are satisfied with the quality improvement achieved by an automated-refactoring tool, one still needs to asses the effect of the refactoring activity on other non-functional properties of a system. For example, what is the current cost of testing a system that has undergone a complete design overhaul? And how does it compare to the cost of testing the initial system design? To address this issue, we propose a refactoring approach that considers testing effort along with design quality improvement.

From the domain of embedded systems and mobile platforms, recent studies [30, 31, 32] have raised concern about the effect of software design quality on energy efficiency. Therefore, mobile application's developers have to consider the effect of design choices not only in terms of object-oriented design quality, but to improve energy efficiency of their applications. To address this concern, we propose an appropriate method to measure energy consumption, and to control for energy efficiency of the refactored designs.

1.3 Thesis Contributions

All the refactoring approaches proposed in this dissertation target instances of the same problem looking at different dimensions and can be summarized as follows.

1. An automated refactoring approach that leverages task context to prioritize the correction of anti-patterns in relevant classes.
2. An automated refactoring approach, based on partial order reduction, to reduce the search-space size of the problem.
3. A multiobjective automated refactoring approach that considers testing effort and design quality.
4. A multiobjective automated refactoring approach that considers energy efficiency and design quality in mobile apps.
5. An Eclipse plug-in to automatically detect and correct software anti-patterns that can be used by developers to improve the design quality of their systems.

1.4 Thesis Organization

The remainder of this dissertation is organized as follows. Chapter 2 provides background information, while chapter 3 surveys related work on anti-pattern's detection, refactoring, and the new dimensions proposed in this dissertation to enhance automated refactoring. Chapter 4 presents an automated refactoring approach that prioritize the refactoring of relevant classes by considering developer's task context. Chapter 5 presents an automated refactoring

approach based on partial order reduction to reduce refactoring search-space; the proposed approach is implemented as an Eclipse plug-in and publicly released. While Chapters 4, 5 present single objective approaches, the rest of the dissertation (Chapters 6, 7) focuses on multiobjective approaches that combine design improvement with other quality dimensions. Specifically, Chapter 6 presents a multiobjective automated refactoring approach that considers testing effort and design improvement. Chapter 7 presents a multiobjective automated refactoring approach for mobile apps, that considers energy efficiency and design improvement. Finally, Chapter 8 draws conclusions and outlines avenues for future works.

1.5 Related Publications

The research and findings presented in this dissertation resulted in several publications and can be traced in various chapters as follows.

- All - **Morales, R**, Chicano, F., Khomh, F., Antoniol, G. (2017). Exact search-space size for the refactoring scheduling problem. Automated Software Engineering. DOI:s10515-017-0213-6.

Ch. 4 **Morales, R**., Soh, Z., Khomh, F., Antoniol, G., Chicano, F. (2017) On the use of developers' context for automatic refactoring of software anti-patterns. Journal of Systems and Software 128, (2017): 236-251. DOI:10.1016/j.jss.2016.05.042.

Ch. 5 **Morales, R**, Chicano, F., Khomh, F., Antoniol, G. (2017). Efficient Refactoring Scheduling based on Partial Order Reduction. Journal of Systems and Software. Under review.

Ch. 6 **Morales, R**, Sabane, A., Musavi, P., Khomh, F., Chicano, F., Antoniol, G. (2016). Finding the Best Compromise Between Design Quality and Testing Effort During Refactoring. Paper presented at the 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER). DOI:10.1109/SANER.2016.23.

Ch. 7 **Morales, R.**, Saborido, R., Khomh, F., Chicano, F., & Antoniol, G. (2017). EARMO: An Energy-Aware Refactoring Approach for Mobile Apps. Transactions on Software Engineering. DOI:10.1109/TSE.2017.2757486.

The following publication is not directly related to the material in this dissertation, but it was produced in parallel to the research contained for this dissertation.

— **Morales, R.**, McIntosh, S., Khomh, F. (2015). Do Code Review Practices Impact Design Quality? A Case Study of the Qt, VTK, and ITK Projects. Paper presented at the 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER). DOI:10.1109/SANER.2015.7081827 .

CHAPTER 2 BACKGROUND

2.1 Introduction

In this chapter, we provide the necessary background for software refactoring, anti-patterns, refactoring styles, refactoring scheduling problem, metaheuristic techniques, multiobjective optimization, and quality models. This chapter is aimed at readers who are unfamiliar with these concepts.

2.2 Refactoring

Refactoring is a software maintenance activity for improving code internal structure, while preserving its external behavior [11]. In the last decade, many works have reported that refactoring can reduce software complexity, improve developer comprehensibility and also improve memory efficiency and start-up time [16, 13]. Hence, developers are advised to perform refactoring operations on a regular basis [33]. According to Mens et al. [34], we can divide the refactoring process in the following steps:

1. **Identify the code entities that need to be refactored**. In this dissertation we use the term code entities to refer to classes as we focus on object-oriented systems. The most widely-use approach to detect code entities that need refactoring (including this dissertation) is the detection of anti-patterns and/or code smells [34]. Anti-patterns are poor design choices [3] to recurring design problems. Typically, they are introduced by inexperienced developers, and represent common pitfalls in software development. According to Coplien and Harrison [5], they are "something that looks like a good idea, but which backfires badly when applied". The difference between anti-patterns and code smells is that the latter ones are local problems which indicate the presence of more general design problems (*i.e.,* anti-patterns). For example, Long Methods, Large class, Low cohesion are just a few symptoms of a Blob Class anti-pattern [33]. Note that the use of anti-patterns to identify code entities to be refactored vary according to the domain of the applications. For example, if we focus on mobile and/or embedded systems, we might consider anti-patterns related to energy efficiency in addition to traditional design anti-patterns, to provide a complete design solution.
2. **Determine which are the most suitable refactorings to be applied.** Fowler proposed a catalog of 22 refactorings and informally linked anti-patterns to refactorings [33] These informal guidelines have been adapted in previous works to generate refactoring

opportunities to remove anti-patterns [35, 36, 37] in a semi-automated fashion.

3. **Ensure that the applied refactorings preserve behavior.** Typically, existing refactoring approaches followed the refactoring pre- and post- conditions proposed by Opdyke [11] to preserve the behavior of the refactored system.
4. **Measure the effect of the refactorings applied on desired quality attributes (*e.g.,* understandability, flexibility, complexity, etc.)**. "You can't control what you can't measure" [38]. Once we applied a set of refactorings, we should be able to assess its impact on design quality. Typically, researchers use code and object oriented metric suites [39, 40], and anti-pattern's/design patterns occurrences [41] to evaluate design improvement.
5. **Maintain consistency between software design and other software artifacts (*e.g.,* documentation, tests, etc.).** After applying and evaluating a set of refactorings, the risk of impacting the consistency between source code and other software artifacts including test suite, design models and documentation is high. Hence, it is recommendable to count with mechanisms to keep their consistency.

The focus of this dissertation and the derived approaches, cover the first 5 steps, while step 6 (the link with other software artifacts) is left to future work. As refactoring have a direct impact on source code, software artifacts like design models (in the form of UML class diagrams), and unit test suites are mainly affected. The former ones could be directly regenerated from the refactored source code using the tool support included in most popular IDEs (*e.g.,* Eclipse, Visual Studio, etc.), while the latter one is a current research area.

2.2.1 Refactoring Styles

In general, developers follow two main refactoring styles [28]: floss refactoring and root-canal refactoring. Floss refactoring consists in applying refactoring to the code while performing other development or maintenance activities, *e.g.,* adding new features, or fixing a bug. In the floss refactoring strategy, the refactoring of the code is not the main goal; developers combine different types of code changes with refactoring. In the case of root-canal refactoring, developers perform the refactoring of the code exclusively. Floss refactoring is a recommended and a common strategy followed by developers according to previous works [28, 42]. However, automating floss refactoring is challenging. So far, existing approaches perform anti-patterns detection on the whole program, and provide a final solution comprised of a set of low-level refactorings which have to be applied (sometimes on classes unknown from the developer) in order to improve the quality [43, 44, 45]. Other approaches, consider more than one objective when refactoring, *e.g.,* semantic-similarity [26] and historical information [46] but still pro-

pose a long sequence of refactorings that often affect classes unknown to developers. These previous approaches are more suitable for root-canal refactoring sessions, where developers set aside a dedicated session just for refactoring.

2.2.2 Anti-patterns and refactoring strategies

In this dissertation we study the occurrences of different anti-patterns types for Object-oriented (OO) and Android Open-Source Systems (OSS), and propose different refactoring approaches to improve design quality by correcting them.

We present in Table 2.1 details of the considered anti-pattern types and the refactoring strategies used to remove them.

To detect anti-patterns, we use DECOR, which is an anti-pattern detection approach, known to attain a good level of precision [52]. DECOR uses a set of rules defined in a domain specific language (DSL) to characterize anti-patterns. These rules are derived from metrics, structural and semantic properties.

In Listing 2.1, we present an example of the Blob detection rule card. The detection of a Blob is the result of the association of a *mainclass* to one or more *DataClass(es)* (line 2). To detect the main class (i.e., the Blob class) the rule used is the result of the union between *Large-ClassLowCohesion*, and *ControllerClass* rules (line 3-6). The union operator is interpreted as logic OR. *LargeClassLowcohesion* is the application of the union operator to *LargeClass* and *LowCohesion*. *LargeClass* is measured by adding number of methods plus number of attributes (NMD+NAD); *LowCohesion* is measured using Lack of Cohesion Among Method of Class (LCOM5) metric [39], which measures the extent of intersections between individual method parameter types lists and the parameter type list of all methods in the class. *Very high* is an ordinal value (*e.g.,* very high, high, medium, low etc.) that represents a threshold value for identifying classes that deviate from the rest of the system. These values are computed using the box-plot statistical technique [53] to relate ordinal values with concrete metric values while avoiding setting artificial thresholds. The number after the ordinal value (*i.e.,* VERY_HIGH) represents the degree of fuzziness, which is the acceptable margin around the threshold relative to the ordinal value (line 5,6). The Blob rule card also includes a lexical property, that is the vocabulary used to name the methods and the class (line 8-11), i.e., using words like Process, Control, etc. Finally, it is necessary that the *mainClass* be associated to one or more data class(es). A data class is a class where the *accesor ratio* (number of accessors/number of methods) is greater or equal to 90% (line 12).

1. `https://source.android.com/devices/tech/`

Table 2.1 List of studied Anti-patterns.

Type	Description	Refactoring(s) strategy
Object-oriented anti-patterns		
Blob (BL) [3]	A large class that absorbs most of the functionality of the system with very low cohesion between its constituents.	Move method (MM). Move the methods that does not seem to fit in the Blob class abstraction to more appropriate classes [44].
Lazy Class (LC) [47]	Small classes with low complexity that do not justify their existence in the system.	Inline class (IC). Move the attributes and methods of the LC to another class in the system.
Long-parameter list (LP) [47]	A class with one or more methods having a long list of parameters, specially when two or more methods are sharing a long list of parameters that are semantically connected.	Introduce parameter-object (IPO). Extract a new class with the long list of parameters and replace the method signature by a reference to the new object created. Then access to this parameters through the parameter object
Refused Bequest (RB) [47]	A subclass uses only a very limited functionality of the parent class.	Replace inheritance with delegation (RIWD). Remove the inheritance from the RB class and replace it with delegation through using an object instance of the parent class.
Spaghetti Code (SC) [3]	A class without structure that declares long methods without parameters	Extract Super Class, Replace method with method object. Extract long methods to new classes and extract a super class with the attributes and methods shared by the SC class and the new extracted classes
Speculative Generality (SG) [47]	There is an abstract class created to anticipate further features, but it is only extended by one class adding extra complexity to the design.	Collapse hierarchy (CH). Move the attributes and methods of the child class to the parent and remove the *abstract* modifier.
Android anti-patterns		
Binding Resources too early (BE) [48]	Refers to the initialization of high-energy-consumption components of the device, *e.g.,* GPS, Wi-Fi before they can be used.	Move resource request to visible method (MRM). Move the method calls that initialize the devices to a suitable Android event. For example, move method call for `requestlocationUpdates`, which starts GPS device, after the device is visible to the app/user (`OnResume` method).
HashMap usage (HMU) [49]	From API 19, Android platform provides *ArrayMap* [50] which is an enhanced version of the standard *Java HashMap* data structure in terms of memory usage. According to Android documentation, it can effectively reduce the growth of the size of these arrays when used in maps holding up to hundreds of items.	Replace HashMap with ArrayMap (RHA). Import ArrayMap and replace HashMap declarations with ArrayMap data structure.
Private getters and setters (PGS) [49, 51]	Refers to the use of private getters and setters to access a field inside a class decreasing the performance of the app because of simple inlining of Android virtual machine [1] that translates this call to a virtual method called, which is up to seven times slower than direct field access.	Inline private getters and setters (IGS). Inline the private methods and replace the method calls with direct field access.

Listing 2.1 Rule card of Blob anti-pattern from DECOR

```
1 RULE_CARD : Blob {
2    RULE : Blob { ASSOC: associated FROM: mainClass ONE TO: DataClass MANY };
3    RULE : mainClass { UNION LargeClassLowCohesion ControllerClass };
4    RULE : LargeClassLowCohesion { UNION LargeClass LowCohesion };
5    RULE : LargeClass { (METRIC: NMD + NAD, VERY_HIGH, 0) };
6    RULE : LowCohesion { (METRIC: LCOM5, VERY_HIGH, 20) };
7    RULE : ControllerClass { UNION (SEMANTIC: METHODNAME, {Process, Control, Ctrl, Command
      , Cmd, Proc, UI, Manage, Drive}), (SEMANTIC: CLASSNAME, {Process, Control, Ctrl,
      Command, Cmd, Proc, UI, Manage, Drive, System, Subsystem}) };
8    RULE : DataClass { (STRUCT: METHOD_ACCESSOR, 90) }; };
```

2.2.3 Refactoring Scheduling problem

Manual refactoring is a complicated task, as there could be more than one solution depending on the design quality attributes that one is interested to improve. Moreover, the application order of refactoring operations matters, as some refactorings can enabled/disable future refactorings. Hence, finding the right sequence of refactorings to apply on a software system is usually a hard task for which no polynomial-time algorithm is known.

We define the refactoring scheduling problem as an optimization problem which consists of finding the best combination of refactorings that maximize the design quality improvement of a software system. This problem can be solved using search-based techniques [54], and the discipline that studies how to apply search based techniques to solving engineering problems is known as Search-Based Software Engineering (SBSE).

Search algorithms typically start by generating one or more random sequences. Next, the quality of each sequence is computed by applying it to the software system in question, and measuring the improvement in the quality attributes of interest using an objective function (*a.k.a.*fitness function). For example, imagine that we have a set of refactorings: $R = \{A, B\}$ to be scheduled. According to our previous work [55], we find that the number of refactoring sequences (S) that we could generate having n refactoring operations is given by Equation 2.1.

$$S = \begin{cases} \lfloor e \cdot n! \rfloor & \forall n \geq 1 \\ 1 & n = 0 \end{cases} \tag{2.1}$$

Applying Equation 2.1 to our example gives us 5 possible sequences ($\lfloor e \cdot 2! \rfloor = 5$): $<>$, $< A >$, $< B >$, $< A, B >$, $< B, A >$, if and only if (iff) we assume that each permutation leads to a different solution. Here the term solution refers to the outcome of applying a refactoring sequence to a system, *i.e.,* the resultant design. Otherwise, $< A, B >$ is commutative and equivalent to $< B, A >$, then only 4 different solutions exist.

In the case of refactorings that affect the same class, the resultant design may vary depending on the order of application of the refactorings, as the application of one refactoring can enable or disable the rest of refactorings.

One of the theoretical contributions of this dissertation, is providing an exact expression to accurately compute the size of the search-space of the refactoring scheduling problem [55]. We leverage this formulation to propose new automated refactoring approaches that target different dimensions: refactoring prioritization using developer's task context, refactoring search-space size and effort reduction, testing effort reduction, and energy efficiency improvement. Chapters 4 to 7 deal with the aforementioned dimensions.

2.3 Metaheuristic techniques

One key component of the approaches presented in this dissertation are meta-heuristic techniques. Depending on the number of parameters, the scope (local or global search) and convergence time, the results may vary and can have an impact on the execution time and the solution's quality. Hence, to provide an insight into which metaheuristics are most effective using automated refactoring, we present the most relevant mono and multiobjective techniques.

2.3.1 Simulated Annealing

It is a metaheuristic technique [56] that imitates the process of metal annealing, by allowing movements of worse quality than the current solution, with a probability that decreases during the search process (when the temperature goes down), until only good quality solutions are accepted. In the first step, the probability toward improvement is low, allowing the exploration of the search space (consequently escaping from local optima), but this behavior changes gradually according to the cooling schedule which is crucial for the performance of the algorithm. The movements between designs are achieved by perturbing the current solution, generally a random one.

2.3.2 Genetic Algorithm

It is a type of evolutionary algorithm [57, 58], where a group of candidate solutions, called individuals or chromosomes are modified through some variation operators, *i.e.,* crossover, and mutation, in order to create new solutions. The selection operator selects the best solutions of each iteration (generation). The search process is guided by an evaluation function, *a.k.a.* fitness function, which assess the fitness of each individual. GA is a population-based algorithm, because it works with several solutions at the same time, contrary to trajectory-based methods like hill-climbing and simulated annealing that work with only one solution at a time.

2.3.3 Variable Neighborhood Search (VNS)

It consists of dynamically changing the neighborhood structures defined at the beginning of the search [59], which expands until a stopping condition is met. In its first step, a solution in the kth neighborhood (where $k = 1 \ldots k_{max}$ represents a neighborhood structure) is randomly selected and altered (shaking phase). Then, a process of local search starts from this point independently of the neighborhood structures. If the outcome solution of the

local search is better than the current solution, the first one is replaced by the new solution and the process restarts at the first neighborhood, otherwise k is incremented and a new shaking phase is started from a new neighborhood. The advantage of this metaheuristic is that (1) it provides diversification when changing neighborhoods in case of no improvement, (2) choosing a solution in the neighborhood of the best solution yields to preserving good features of the current one. For the evaluation of the approach proposed in chapter 4, the shaking phase consists of modifying i refactoring operations from the end of the sequence, until we reach the starting point. The local search mechanism is responsible of applying all the possible variations to the candidate solutions and selecting the best local optima.

2.3.4 Multiobjective optimization

Optimization problems with more than one objective do not have single solutions because the objectives are usually in conflict. Consequently, the goal in these problems is to find a set of solutions that are non-dominated, in the sense that there is no solution which is better with respect to one of the objective functions without achieving a worse value in at least another one.

More formally, let y_1 and y_2 be two solutions, for a multiobjective maximization problem, and $f_i, i \in 1 \dots n$ the set of objectives. The solution y_1 *dominates* y_2 if: $\forall i, f_i(y_2) \leq f_i(y_1)$, and $\exists j | f_j(y_2) < f_j(y_1)$.

The use of multiobjective algorithms have shown to be useful in finding good solutions in a search space. There is even a procedure called *multi-objectivization* that transforms a single-objective problem into a multiobjective one, by adding some helper functions [60]. Hence, the use of multiobjective optimization techniques is suitable to solve the refactoring scheduling problem as they lessen the need for complex combinations of different, potentially conflicting, objectives and allow software maintainers to evaluate different candidate solutions to find the best trade-off.

The set of all non-dominated solutions is called the *Pareto Optimal Set* and its image in the objective space is called Pareto Front. Very often, the search of the *Pareto Front* is NP-hard [61], hence researchers focus on finding an approximation set or reference front (RF) as close as possible to the Pareto Front.

Chapters 6, and 7 formulate the problem of refactoring as a multiobjective optimization problem, to improve the design quality, while controlling for other quality attributes like testing effort and energy consumption. To solve these problems, we use Evolutionary Multiobjective Optimization (EMO) algorithms, a family of metaheuristic techniques that are known to

perform well handling multiobjective optimization problems [62].

We provide a brief description of the EMO used in this dissertation below.

2.3.5 Non-dominated sorting genetic algorithm (NSGA-II)

NSGA-II [63] proceeds by evolving a new population from an initial population, applying variation operators like crossover and mutation. Then, it merges the candidate solutions from both populations and sorts them according to their rank, extracting the best candidates to create the next generation. If there is a conflict when selecting individuals with the same ranking, the conflict is solved using a measure of density in the neighborhood, *a.k.a.* crowding distance.

2.3.6 Strength Pareto Evolutionary Algorithm 2 (SPEA2)

SPEA2 [64] uses variation operators to evolve a population, like NSGA-II, but with the addition of an *external archive*. The archive is a set of non-dominated solutions, and it is updated during the iteration process to maintain the characteristics of the non-dominated front. In SPEA2, each solution is assigned a fitness value that is the sum of its strength fitness plus a density estimation.

2.3.7 Multiobjective Cellular Genetic Algorithm (MOCell)

It is a cellular algorithm [65], that includes an external archive like SPEA2 to store the non-dominated solutions found during the search process. It uses the crowding distance of NSGA-II to maintain the diversity in the Pareto front. Note that the version used in this dissertation is an *asynchronous* version of MOCell called aMOCell4 [66]. The selection consists in taking individuals from the neighborhood of the current solution (cells) and selecting another one randomly from the archive. After applying the variation operators, the new offspring is compared with the current solution and replaces the current solution if both are non-dominated, otherwise the worst individual in the neighborhood will be replaced by the offspring.

2.4 Quality Models

Many quality models have been proposed to assess the quality of OO systems. A quality model is a set of quality attributes related to a set of metrics. The main purpose of quality models is to facilitate the continuous improvement of a software system [67, 68]. In this

dissertation we use Quality Model for Object-Oriented design (QMOOD) [40] to evaluate the design quality of a system before and after refactoring. QMOOD defines six design quality attributes in the form of metric-quotient weighted formulas that can be easily computed on the design model of a software system, which makes it suitable for automated-refactoring experimentations. Another reason for choosing the QMOOD quality model is the fact that it has been used in many previous works on refactoring [44, 45, 69], which allows for a replication and comparison of the obtained results.

We present a brief description of the quality attributes used in this study.

— Reusability: the degree to which a software module or other work product can be used in more than one software program or software system.

— Flexibility: the ease with which a system or component can be modified for use in apps or environments other than those for which it was specifically designed.

— Understandability: the properties of a design that enables it to be easily learned and comprehended. This directly relates to the complexity of the design structure.

— Effectiveness: the design's ability to achieve desired functionality and behavior using OO concepts.

— Extendibility: The degree to which an app can be modified to increase its storage or functional capacity.

Formulas for computing these quality attributes are described in Table 2.2. In this work we do not consider the functionality quality attribute because refactoring being a behavior-preserving maintenance activity, should not impact the functionality of a software system.

Table 2.2 QMOOD Evaluation Functions.

Quality Attribute	Quality Attribute Calculation
Reusability	-0.25 * DCC + 0.25 * CAM + 0.5 * CIS + 0.5 * DSC
Flexibility	0.25 * DAM - 0.25 * DCC + 0.5 * MOA +0.5 * NOP
Understandability	-0.33 * ANA + 0.33 * DAM - 0.33 * DCC + 0.33 * CAM -0.33 * NOP - 0.33 * NOM - 0.33 * DSC
Effectiveness	0.2 * ANA + 0.2 * DAM + 0.2 * MOA + 0.2 * MFA + 0.2 * NOP
Extendibility	0.5 * ANA -0.5 * DCC + 0.5 * MFA + 0.5 * NOP

where DSC is design size, NOM is number of methods, DCC is coupling, NOP is polymorphism, NOH is number of hierarchies, CAM is cohesion among methods, ANA is avg. num. of ancestors, DAM is data access metric, MOA is measure of aggregation, MFA is measure of functional abstraction, and CIS is class interface size.

CHAPTER 3 RELATED WORK

3.1 Introduction

In this chapter, we survey the most relevant research work related to this dissertation, which includes anti-patterns detection, refactoring, and the four themes studied in this dissertation: developer's task context, refactoring scheduling problem, testing effort, and energy efficiency of mobile apps.

3.1.1 Anti-patterns detection

Concerning the detection of anti-patterns, we present the following representative works.

Marinescu [70] proposed a metric-based approach to detect anti-patterns capturing deviations from "good design principles" through a set of rules comprised of metrics joined by set operators and relative thresholds. Munro [71] presented a similar rules and metrics-based approach to detect code smells, and evaluated the selected metrics and threshold values through an empirical study.

Moha et al. proposed a domain-specific language to characterize anti-patterns based on a literature review of existing work. They also proposed algorithms and a platform to automatically convert specifications into detection algorithms to apply in a software system. They achieved good precision and a perfect recall [52].

Khomh et al. proposed a Bayesian approach to account for the uncertainty of the loosely specified definitions of anti-patterns. By computing the probability that a class participate in an anti-pattern, this approach allows quality analyst to prioritize the inspection of bad candidate classes [72].

Marinescu et al. proposed InCode, an Eclipse plug-in [73] to assess the quality of a software system in terms of anti-patterns by analyzing the source code on the fly using, the detection strategies proposed in [70]. This tool provides different visualizations and a very comprehensive definition of rules, metrics and design defects. The disadvantage of this approach is that it does not include a way to remove the detected anti-patterns.

Palomba et al. proposed HIST, a datamining approach to detect a set of five anti-patterns based on change history information mined from versioning systems [74]. They evaluate their approach on a testbed of 20 Java systems and a case study with 12 developers of four opens source systems, and found that by including history information, they could detect

more instances of anti-patterns than traditional approaches. Concerning anti-patterns in mobile applications, the same research group proposed a detection tool, called aDoctor [75], to detect 15 android anti-patterns using static analysis code techniques. They test aDoctor on a testbed of 18 android apps and attained a detection precision close to 100%.

3.1.2 Search-based Refactoring

We present a sample of representative works in this category. O'Keeffe and Cinnéide [43] proposed an approach that relies on the QMOOD model [40] to assess the quality of the candidate refactorings. They implemented their approach using local search techniques, namely Simulated annealing (SA), and two versions of hill climbing. They found strong evidence that QMOOD flexibility and understandability attributes are the most suitable attributes to assess the quality of the refactoring solutions. The same authors extended this study in [69] by adding GA. They found Multiple-ascent hill climbing to be the most efficient search technique in terms of speed, quality obtained in different program inputs, and consistence for a different set of parameters; GA performs better with high values of crossover and mutation; the effectiveness of simulated annealing varies in function of the input program.

Seng et al. [44] proposed an approach based on genetic algorithm, that aims to improve the cohesion of the entities through the implementation of the move method refactoring and evaluated the quality of the refactoring sequences with a fitness function that comprise coupling, cohesion, complexity and stability measurements.

Harman and Tratt [45] introduced a multiobjective approach for the problem of refactoring that allows to treat the refactoring problem as a multi-objective problem, where the goal is to find the *Pareto front*, *i.e.,* the set of solutions where there is no component that can be improved without decreasing the quality of another component. Thus, the outcome is not a single solution but a set of optimal solutions to be selected by the developer.

Ouni et al. [26] proposed a multiobjective evolutionary algorithm based on the Non-dominated Sorting Genetic Algorithm-II (NSGA-II) [63]. The two conflicting objectives of their approach are correcting the larger quantity of design defects, while preserving semantic coherence. For the first objective, they input a set of rules to characterize design defects from the literature, and select the rules that detect the most design defects from a set of previous detected defects (example-based approach). The second objective is achieved by implementing two techniques to measure similarity among classes, when moving elements between them. The first technique evaluates the cosine similarity of the name of the constituents, *e.g.,* methods, fields, types. The second technique considers the dependencies between classes.

Moghadam and Cinnéide [76] proposed an automated approach where the goal is to reach a desired design model, described as a UML diagram. The approach takes as input the source code of the *desired program* and the program to be improved; then, it abstracts the corresponding UML design models, and computes the differences between them. Next, it maps the set of differences to source-level refactorings that will be applied in the code. The search problem consists in finding the larger sequence of refactorings that can be legally applied in the program. The metaheuristics algorithms used are the same as those implemented in [69].

Mkaouer et al. [77] proposed an extension of [26], by allowing user's interaction with the refactoring solutions. Their approach consists of the following steps: (1) a NSGA-II algorithm proposes a set of refactoring sequences; (2) an algorithm ranks the solutions and presents them to the user who will judge the solutions; (3) a local-search algorithm updates the set of solutions after n number of interactions with the user or when m number of refactorings have been applied.

3.1.3 Other refactoring approaches

Tsantalis et al. proposed different semi-automated approaches to detect refactoring opportunities like extract method, move method, and removing non-trivial code smells like feature envy and type-checking to improve the design quality of a system. They implemented their techniques as an Eclipse plug-in, named JDeodorant [1] allowing their evaluation on Java projects [37, 78, 79, 80]. Semi-automatic approaches provide an interesting compromise between fully automatic detection techniques that can deviate from human context and manual inspections which are tedious and subjective. However, they require the developer to take decisions about the order of refactorings to be applied. On the contrary, the approaches proposed in this dissertation aim to relieve developers from the time-consuming task of selecting the best sequence of refactorings and evaluating their impact one by one.

3.1.4 Refactoring Scheduling Problem

In this category we present some of the representative works that proposed techniques to schedule a set of candidate refactorings to improve the effect of refactoring according to some defined quality objectives, and to overcome the weakness of search-based approaches that are based on evolutionary algorithms.

Mens et al. [81] formulated a model to analyze refactoring dependencies using critical pair analysis. However, this model lacks automation to schedule refactorings once potential con-

1. http://www.jdeodorant.com

flicts are detected.

Liu et al. [82] proposed an heuristic algorithm to schedule refactorings based on a conflict matrix and the effects of candidate refactorings on the design. They evaluated their approach on a house-made modeling tool using QMOOD [40] model and found that it outperforms a manual approach.

Liu et al. [83] proposed an algorithm to schedule the refactoring of code smells using pair-wise analysis. By using topological sort on graph that represents the type of anti-patterns detected, they reduced the search of sequences by removing redundant edges that correspond to overlapping smells. However, they did not automate the application of the refactorings on the systems.

These previous works require a list of candidate refactorings in advance to schedule a sequence of refactorings.

Bouktif et al. [84] proposed an approach to schedule refactoring actions in order to remove duplicated code using genetic algorithm.

Lee et al. [85] proposed an approach to automatically schedule refactorings to remove method clones using a Competent Genetic Algorithm. The proposed approach was evaluated using a testbed of four open-source systems. They found higher quality improvement compared to manual and greedy search in terms of QMOOD model [40], but the same quality improvement using exhaustive search for the less complex systems.

Zibran and Roy [86] proposed an approach to refactor code clones based on constraint programming. They evaluated their approach on four in-house systems and reported that it outperformed greedy and manual approaches.

Moghadam and Ó Cinnéide [87] proposed an approach for refactoring scheduling where the quality goal is set to be a desired design expressed as a UML model, and the refactoring operators are transformations aimed at achieving that model. They evaluated their approach on an open-source system with a small set of 50 refactorings to be scheduled and found that the produced sequence of refactorings could transform the initial design into the desired design with 100% of success.

3.1.5 Management and prioritization of anti-pattern's correction

In this subsection, we discuss related work on management and prioritization of anti-patterns.

Use of refactoring history. Ouni et al. [27] proposed an approach to leverage development history to prioritize the application of refactorings that have been applied in previous maintenance sessions. They claimed that by measuring the similarity of previous refactorings applied in similar contexts they can improve the quality of the refactorings proposed. They applied their proposed approach on a testbed of 5 open-source systems and reported an improvement of around 10% compared to previous approaches that do not consider refactoring history.

Use of developers' task context. To the best of our knowledge, we propose the first automated refactoring approach that leverages developer's task context to prioritize anti-patterns correction (*cf.* chapter 4). Hence, we present related research works that leverage task context with two objectives: (1) to assist developers during task resolutions (*i.e.,* the context is collected and directly used during the resolution of the current task), and (2) to understand developers' activities.

Kersten and Murphy [88] used the task context to reduce information overhead by filtering and keeping in the developers' environment (*i.e.,* package explorer in the IDE) only the program entities relevant to the developer's task. This prevents the developer from searching for relevant information in a large information space; improving the developer's productivity. Robbes and Lanza [89] also used developer's previously collected contexts to build a code completion tool that reduces developers' scrolling effort. Users found their proposed tool to be more accurate than previous tools. Recently, Lee et al. [90] proposed an approach (named MI) to recommend relevant entities to developers. They used both view and selection activities on the entities from the developer's context and mined association rules to identify relevant entities.

Among the studies that used developers' task context to examine developers' activities is the work of Sanchez et al. [91], who studied developers interruptions and found that work fragmentation is correlated with lower productivity. Ying and Robillard [92] and Zhang et al. [93] studied how developers perform editing activities. Ying and Robillard [92] defined file editing styles (edit-first, edit-last, and edit-throughout) and found that enhancement tasks are associated with a high fraction of edit events at the beginning of the programming session(*i.e.,* edit-first). Zhang et al. [93] characterize how several developers concurrently edit a file and derive concurrent, parallel, extended, and interrupted file editing patterns. They found these file editing patterns to be related to future faults. Soh et al. [94] used Mylyn context to study how developers' navigate through program entities. They found that developers spend more effort on tasks when they exhibit unreferenced exploration (*i.e.,* program entities are almost equally revisited) compared to reference exploration (*i.e.,* revisitation of a set of entities).

3.1.6 Testing Strategies

Testing is an essential but expensive activity to ensure software quality and reliability. Beizer [95] estimates the cost of software testing at 50% of the overall cost of software. Hence, researchers investigate various directions to reduce testing effort and increase its effectiveness. As we propose an automated approach that leverages testing effort to propose refactoring solutions that improve the testability of software systems (*cf.* Chapter 6), we present some related research works on software testing.

Studies related to factors that impact testing effort can be found in [96, 97]; while approaches to automatically generate test data are found in [98, 99]. Finally, other studies defined strategies that can efficiently target specific type of systems or specific kind of faults. In this category, we can cite the different OO strategies that have been proposed to overcome traditional testing strategies limitations regarding the test of OO systems: state based testing [100], pre-and-post conditions testing [101], and MaDUM testing [102]. Another direction to reduce testing effort is to refactor a system specifically for testing. Belonging to this last category are the refactoring as testability transformation works [103, 104]. Refactoring as testability transformation is different from refactoring of anti-patterns in the sense that the system is not changed to improve the design quality of a system, but another version is created just to facilitate the generation of test data that will be used to test the original system. To the best of our knowledge, there is no work that automatically apply refactoring of anti-patterns to reduce testing effort. Sabane et al. [97] present some refactoring actions to reduce testing effort based on the MaDUM strategy. They manually apply some extract method refactorings to reduce the number of transformers of a class under test. These refactorings were performed manually, attempting to reduce the testing effort, not to remove anti-patterns or improve design quality. In fact, some of them were found to decrease the understandability of the class.

3.1.7 Refactoring of Mobile Apps

As we propose an automated approach that considers energy consumption during the refactoring of mobile apps (*cf.* Chapter 7), we present related research works on anti-patterns, and energy efficiency of mobile apps.

Mobile anti-patterns

Linares-Vásquez et al. [105] leveraged DECOR to detect 18 OO anti-patterns in mobile apps. Through a study of 1,343 apps, they have shown that anti-patterns negatively impact the

fault-proneness of mobile apps. In addition, they found that some anti-patterns are more related to specific categories of apps.

Verloop [106] leveraged refactoring tools, such as PMD [2] or JDeodorant [107] to detect code smells in mobile apps, in order to determine if certain code smells have a higher likelihood to appear in the source code of mobile apps. In both works, the authors did not considered Android-specific anti-patterns.

Reimann et al. [108] proposed a catalog of 30 quality smells specific to the Android platform. These smells were reported to have a negative impact on quality attributes like consumption, user experience, and security. Reimann et al. also performed detections and corrections of certain code smells using the REFACTORY tool [109]. However, this tool has not been validated on Android apps [110].

Li et al. [31] investigated the impact of Android developing practices and found that *accessing class fields*, *extracting array length into a local variable* in a for-loop and *inline getter and setters* can reduce the energy consumption of an app in test harness developed specifically for this purpose.

Hecht et al. [110] analyzed the evolution of the quality of mobile apps through the analysis of 3,568 versions of 106 popular Android apps from the Google Play Store. They used an approach, called *Paprika*, to identify three object-oriented and four Android-specific anti-patterns from the binaries of mobile apps. Recently, they also evaluated the impact of removing three types of Android anti-patterns (two of them also studied in this work, *e.g., HashMap usage*, and *private getters and setters*) using a physical measurement setup [111].

Energy Consumption of mobile apps

There are several works on the energy consumption of mobile apps [112, 113, 114, 115, 116, 117]. In this dissertation we focus on those who aimed to understand software energy consumption [114], energy usage [118], or the impact of developer's choices on energy efficiency [119].

Green Miner [117] is a dedicated hardware mining software repositories testbed. The *Green Miner* physically measures the energy consumption of Android mobile devices and automates the reporting of measurements back to developers and researchers. A *Green Miner* web service [3] enables the distribution and collection of green mining tests and their results. The hardware client unit consists of an *Arduino*, a breadboard with an *INA219* chip, a *Raspberry*

2. https://pmd.github.io/
3. https://pizza.cs.ualberta.ca/gm/index.py

Pi running the *Green Miner* client, a USB hub, and a *Galaxy Nexus* phone (running Android OS 4.2.2) which is connected to a high-current 4.1V DC power supply. Voltage and amperage measurement is the task of the *INA219* integrated circuit which samples data at a frequency of $50\,Hz$. Using this web service, users can define tests for Android apps and run these tests to obtain and visualize information related to energy consumption.

Energy models can be provided by a *Software Environment Energy Profile (SEEP)* whose design and development enables the per instruction energy modeling. Unfortunately, it is not common practice for manufacturers to provide *SEEPs*. Because of that, different approaches have been proposed to measure the energy consumption of mobile apps. Pathak et al. [120] proposed *eprof*, a fine-grained energy profiler for Android apps, that can help developers understand and optimize their apps energy consumption. In [121], authors proposed the software tool *eLens* to estimate the power consumption of Android applications. This tool is able to estimate the power consumption of real applications to within 10% of ground-truth measurements. One of the most used energy hardware profilers is the *Monsoon Power Monitor* which has been used in several works.

Da Silva et al. [122] analyzed how the *inline method* refactoring impacts the performance and energy consumption of three embedded software written in Java. The results of their study show that inline methods can increase energy consumption in some instances while decreasing it in others.

Sahin et al. [123] investigated how high-level design decisions affect an application's energy consumption. They discuss how mappings between software design and power consumption profiles can provide software designers and developers with insightful information about their software power consumption behavior. In another work, Sahin et al. [118] investigated the impact of six commonly-used refactorings on 197 apps. The results of their study have shown that refactorings impact energy consumption and that they can either increase or decrease the amount of energy used by an app. The findings also highlighted the need for energy-aware refactoring approaches that can be integrated in IDEs.

Banerjee et al. [32] proposed a technique to identify energy hotspots in Android apps by the generation of test cases containing a sequence of user-interactions. They evaluate their technique using a testbed of 30 apps from *F-Droid*.

Pinto [124] suggested a refactoring approach to improve the energy consumption of parallel software systems. The approach was manually applied to 15 open source projects and reported an energy saving of 12%.

Li et al. [125] proposed an approach to transform web apps to improve energy consumption of

mobile apps and achieved an improvement of 40%, with an acceptance rate of 60% among the users in a testbed of seven web apps. To address the same problem, but using multiobjective technique, Linares-Vásquez et al. [30] proposed an approach to generate *energy-friendly* color palettes that are consistent with respect to the original design in a testbed of 25 apps.

Wan et al. [126] proposed a technique for detecting graphic user interfaces that consumes more energy than desirable. After evaluating their technique on a testbed of 10 apps, they reported that their approach can accurately predict the energy consumption of an app within 14% of the ground truth according to the error estimation rate, which is the accuracy of the power estimate compared to the real measurements.

Bruce et al. [127] leverage *Genetic Improvement* to improve the energy consumption of three *MiniSAT* downstream apps achieving 25% of improvement.

Manotas et al. [128] proposed a framework (*SEEDS*) to automatically select the most energy efficient Java's Collections API and achieved 17% of energy usage improvement on a testbed of seven Java apps.

Hecht et al. [49] conducted an empirical study focusing on the individual and combined performance impacts of three Android performance anti-patterns from two open-source Android apps. These authors evaluated the performance of the original and corrected apps on a common user scenario test. They reported that correcting these Android code smells effectively improves the user interface and memory performance.

CHAPTER 4 USING DEVELOPER'S CONTEXT FOR IMPROVING AUTOMATED REFACTORING

4.1 Introduction

In this chapter we study the use of developer's task context to prioritize classes that need to be refactored. The main problem with current automated refactoring approaches [24, 36, 27, 129] (*cf.* Chapter 3) is that they rely on (1) a collection of *bad design code examples* which adds a new task and responsibility to developers, to collect and manage the aforementioned collection, or (2) a *desired design* [130], where developers are expected to input the model that they want to achieve in advance. In any case, developers have to accept a *global solution*, which might consider classes that are not part of the scope of the maintenance task that they are performing, or in other words, *out of the context*. As a result, developers have to deal with a long sequence of refactorings that often affect classes on which they have no previous knowledge, or lack of ownership. Yet, previous studies have shown that developers prefer approaches that do not disrupt their work flow. In fact, Murphy-Hill et al. report that developers prefer approaches that suggest refactoring operations that can be applied to the group of files that are currently active in their workspace [42] and-or integrated in their preferred Integrated Development Environment (IDE).

This chapter describes an automated refactoring approach that is based on developer's task context. We refer to our proposed approach as Refactoring approach based on task Context (ReCon) for the rest of this dissertation. ReCon has the following advantages over the state-of-the-art approaches: (1) it does not require a set of bad examples, as detection rules are derived from the literature of anti-patterns; (2) it is customizable at a high-level of abstraction, using a domain specific language, and (3) it generates a set of *local refactoring solutions*, *i.e.,* refactoring suggestions over *active classes*, that developers can apply on the fly while performing his development or maintenance task.

To evaluate the performance of ReCon, we mined 1,705 Mylyn Interaction Histories (IH) from three open-source projects (Mylyn, PDE, and Eclipse Platform). From the IH of each task, we computed the relevant classes and entities targeted by the developer when performing the task (this constitutes the task context). Then, we download the code snapshot for each task, from the Version Control System (VCS) of the project and evaluate the quality of the project before and after applying ReCon, considering the removal of anti-patterns and the quality gain in terms of five desirable quality attributes: understandability, flexibility, reusability, effectiveness and extendibility defined in QMOOD [40]. We run our approach using two

different styles, the refactoring of all classes in a system (root-canal), and the incremental refactoring of certain classes explored during maintenance sessions (floss refactoring).

4.2 Prioritizing refactoring of anti-patterns by leveraging Developer's Task Context

In this chapter, we aim to support developers during floss refactoring performed while implementing their daily development and maintenance tasks.

Thus far, automated approaches have focused on providing complete refactoring solutions which cannot be applied in regular developer tasks, but in dedicated refactoring sessions. We, therefore, set out to address the following question:

Central Question: *Is it possible to improve automated refactoring by leveraging developer's task context?*

To guide the search of refactoring opportunities in relevant artifacts (*i.e.,* classes relevant to developers' task), we leverage information provided by interaction traces (captured using the monitoring tools) and suggest refactorings that the developer can apply on the fly while performing his task.

By *developers' task context*, or simply task context, we refer to the program entities that the developers used when resolving a development or maintenance task. In fact, during development or maintenance sessions, developers usually interact with program entities through their IDE. The task context is accessible using monitoring tools, such as Mylyn [131], or MimEc [132]. These tools log all developers' interaction with the program entities (*e.g.,* interaction trace). In the following we use Mylyn as an example of a monitoring tool.

Mylyn is an Eclipse plug-in for task and application lifecycle management, which introduces the concept of task-focused interface [133] (*i.e.,*the IDE realigns the objects in the User Interface (UI) to show only relevant code entities). When developers activate a task, Mylyn automatically build the task context by monitoring developer's activities. Task context is a graph of program elements and their relationships that a developer uses to perform a task. Mylyn builds the task context based on a degree-of-interest model that consist of weighing the relevance of elements to the task.

A developer can create a Mylyn task to track the code changes when handling a change request. The developer's programming activities are monitored by Mylyn to create a task context and predict relevant artifacts for the task. The programming activities monitored

by Mylyn include selection and edition of files. In Mylyn, each activity is recorded as an interaction event between a developer and the IDE. There are eight types of interaction events in Mylyn, as described in Table 4.1. Three types of interaction events are triggered by a developer, *i.e., Command*, *Edit* and *Selection* events.

Each Mylyn log has a task identifier, which often contains the change request ID. A Mylyn log is stored in an Extensible Markup Language (XML) format. Its basic element is *InteractionEvent* that describes the event. The descriptions include: a starting date (*i.e., StartDate*), an end date (*i.e., EndDate*), an event type (*i.e., Kind*), the identifier of the UI affordance that tracks the event (*i.e., OriginId*), and the names of the files involved in the event (*i.e., StructureHandle*). Listing 4.1 presents an example of *InteractionEvent* that was recorded during the correction of the bug #311966 [1] of Eclipse's bug repository.

Table 4.1 Event Types from Mylyn.

Event Type	Description	Developer-Initiated ?
Command	Click buttons, menus, and type in keyboard shortcuts.	Yes
Edit	Select any text in an editor.	Yes
Selection	Select a file in the explorer.	Yes
Attention	Update the meta-context of a task activity.	No
Manipulation	Directly manipulate the degree of interest (DOI) value through Mylyn' user interface.	No
Prediction	Predict relevant files based on search results.	No
Preference	Change workbench preferences.	No
Propagation	Predict relevant files based on structural relationships (*e.g.,* the parent chain in a containment hierarchy).	No

Listing 4.1 Structure of the Mylyn log of bug #311966.

```
<?xml version="1.0" encoding="UTF-8"?>
<InteractionHistory Id="https://bugs.eclipse.org/bugs-311966"
Version="1">
<InteractionEvent
StartDate="2010-06-25 11:27:23.935 EDT"
EndDate="2010-06-25 11:27:27.777 EDT"
Kind="edit"
OriginId="org.eclipse.jdt.ui.CompilationUnitEditor"
StructureHandle="/org.eclipse.mylyn.bugzilla.ui/src/org/eclipse/mylyn/internal/bugzilla/
    ui/tasklist/BugzillaConnectorUi.java"
StructureKind="resource"
Interest="2.0"
Delta="null"
Navigation="null"
/>
...
<InteractionEvent
...
/>
</InteractionHistory>
```

1. https://bugs.eclipse.org/bugs/show_bug.cgi?id=311966

In the rest of this chapter, we refer to the set of program entities relevant to a developer's task as the *context*.

4.3 Approach

In Figure 4.1 we present the workflow of ReCon. A rectangle indicates a process while a rectangle with the corner folded indicates the input/output of a process. ReCon takes as an input an interaction trace generated by a monitoring tool, and a software system. We generate an abstract model by performing a static analysis of the software system. We identified the relevant classes from the interaction trace (task context), and build a map of anti-patterns based on the anti-patterns detection results. Next, we generate a list of candidate refactorings to correct the anti-patterns detected. After that, we use a search algorithm to find the best combination of refactorings that remove the largest number of anti-patterns. The search algorithm can be any metaheuristic technique such as GA, SA, or VNS. It can be argued that in tasks comprised of a few classes, a greedy algorithm, or in the absence of conflicts, applying all the candidate refactorings is more effective than running a metaheuristic. However, it is common to find conflict between refactorings operations, even in the same class. In that case, a search-based algorithm is useful. For that reason, if we detect that there is no conflict between the refactoring operations generated for a determined system, we simply apply all the refactoring operations, without regards of the application order.

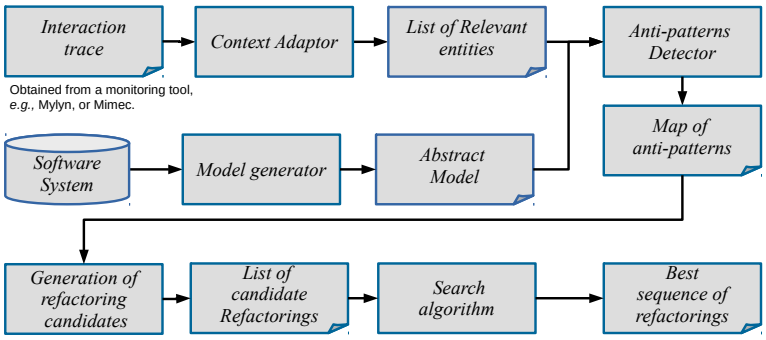

Figure 4.1 Workflow of ReCon.

The search algorithm receives as an input a set of candidate refactoring operations and produces as an output a sequence of refactoring operations to be applied in the system. During the search process, which is non-deterministic, the candidate sequences are evaluated using a blackbox approach, where each candidate sequence is applied to a copy of the abstract model,

and the resultant model is evaluated in terms of anti-patterns occurrences. This process continues until it finds a sequence that removes all anti-patterns, or the search algorithm reaches the maximum number of iterations. The final output is a valid (non-conflicted) sequence of refactorings to improve the design quality of the code entities in the task context.

4.4 Evaluation

The *goal* of this case study is to assess the effectiveness of ReCon in correcting anti-patterns in object-oriented systems (OO) during maintenance tasks. The *quality focus* is the improvement of the design quality of OO systems. The *perspective* is that of researchers interested in developing automated refactoring tools, and developers interested in improving the design quality of their code.

The context consists of *1,705* task contexts, and *1,013* code snapshots from three open-source software systems (Mylyn, PDE, and Platform), and three metaheuristic techniques (GA, SA, VNS). In Table 4.2, we present relevant information about the Eclipse projects studied and the count for each type of anti-pattern.

Table 4.2 Descriptive statistics of the studied Eclipse projects.

Name	Number of classes	Number of tasks	Number of anti-patterns
Mylyn	2,365	183	167
PDE	16,045	129	3,512
Platform	20,259	213	3,558

4.4.1 Dependent and Independent Variables

To assess whether automatic refactoring using context improves the quality of a system, we consider the following dependent and independent variables:

Independent variables: The independent variables define the refactoring approaches that we performed. We use two refactoring styles: automated root-canal refactoring and automated floss refactoring.

Dependent variables: We use the following variables to assess whether a refactoring approach (*i.e.,* automated floss refactoring or automated root-canal refactoring) improves the quality of the system.

— Number of anti-patterns removed after refactoring (#AP): For each refactoring approach, we compute the number of anti-patterns removed. The number of anti-patterns

removed is an indication of the improvement of the design quality of the system. The more anti-patterns are removed, the better is the design quality of the system.

— Design quality improvement. After finding the best refactoring solution for each program using the proposed metaheuristics, we evaluate the resulting design code using 5 quality functions attributes of QMOOD hierarchical model.

4.4.2 Data Collection and Processing

We follow two main steps to collect and process the data of our experiment:

(1) In step one, we collect developers' interaction traces from the Eclipse bug repository [2]. Interaction traces appear as attachments to a bug report. These interaction traces contain program entities that developers interacted with (*i.e.,* the context). When collecting an interaction trace during a bug resolution, developers also perform modifications on the system. These modifications that change the state of the system (and which can improve or degrade the quality of the system) are essential for the completion of the developer's task. Hence, it is important to consider these developers' modifications when looking for refactoring opportunities.

We consider a patch attached to a bug report as the changes performed by a developer during his working session if and only if the interaction trace and the patch are attached by the same developer at the same time [134]. In our experiment, we consider the interaction traces and patches of three Eclipse projects that have most interaction traces. Precisely, we downloaded 663, 132, and 218 couples of interaction traces and patches for Mylyn, PDE and Platform systems, respectively.

(2) In step two, we identify the start timestamp of each interaction trace. We consider that developers checkout the system before they start fixing a bug. Thus, we checkout the snapshot of the system from the appropriate source code repository (*i.e.,* the VCS of the project on which the task was performed) on the master branch and before the start timestamp. In total we checkout 663, 132, and 218 snapshots of Mylyn, PDE and Platform projects, respectively. Snapshots provide the states of the system used by the developers and the patches contain the changes made by the developers.

4.4.3 ReCon implementation

We implement the workflow depicted in Figure 4.1 using Java. We start by extracting relevant code entities in a developer's task from interaction traces, generated by Mylyn,

2. https://bugs.eclipse.org/bugs/

using our context adaptor. Then, we perform the static analysis of the system, using Ptidej tool suite [3]. The result is a Pattern and Abstract-level Description Language (PADL) model, which is an abstract representation of the code entities, such as classes, interfaces, methods and attributes, and their structural relationships, *e.g.,* inheritance, association, etc. Next, we detect anti-patterns in the PADL model using Software Architectural Defects (SAD) tool, which is the implementation of DECOR [52], a well known approach to define and detect anti-patterns, also part of Ptidej tool suite (*cf.* Section 2.2.2).

For this chapter, we consider four types of anti-patterns, namely **Lazy Class (LC)**, **Long Parameter list (LP)**, **Spaghetti Code (SC)** and **Speculative Generality (SG)**. We select these anti-patterns, because (1) they are well defined in the literature, with the recommended steps to remove them [47], (2) they are easy to identify by developers [7], (3) they have been studied in previous works [4, 52, 135, 72].

In Table 2.1, we briefly described each anti-pattern considered in this dissertation, and the proposed refactoring(s) operation to correct them. The proposed refactorings procedures, suggested in the literature [33, 3], aim to support developers, with a previous knowledge of the system functionality, to improve the quality of their systems. They rather exhibit developer's experience, than provide formal rules for characterizing anti-patterns. Hence, to automate this task, we adapted the aforementioned procedures by leveraging the structural information computed from the abstract model and the anti-pattern detection step. We defined a *refactoring strategy* for each anti-pattern, following the recommendations from previous works on semantic preservation [11, 136]. In the next paragraphs, we provided a brief explanation of the refactoring strategies developed used in this dissertation, based on the anti-pattern's definitions described in the literature.

In the case of lazy class, the proposed refactoring is *inline class*, which consist of moving all the features of a LC to another class, and after that remove the LC class from the system. As an example, we present in Figure 4.2 the UML diagram of class `XMLCleaner`, from Eclipse Mylyn Project. This class, which aims to escape "&" characters from XML files, consists of only one public method with less than 20 LOC. Hence, a possible candidate refactoring operation is inlining class `XMLCleaner` to another related class (In the example showed in Figure 4.2, we inlined `XMLCleaner` to `AbstractReportFactory`, as `AbstractReportFactory` calls the method collectResults).

As we can observe, inline class refactoring is comprised of a series of *low level* refactorings that have to be applied in specific order, *e.g.,* move method(s) and/or attribute(s) to another class, update call sites, and delete LC class. Unlike previous refactoring approaches [24, 22]

3. http://www.ptidej.net/tools/designsmells/

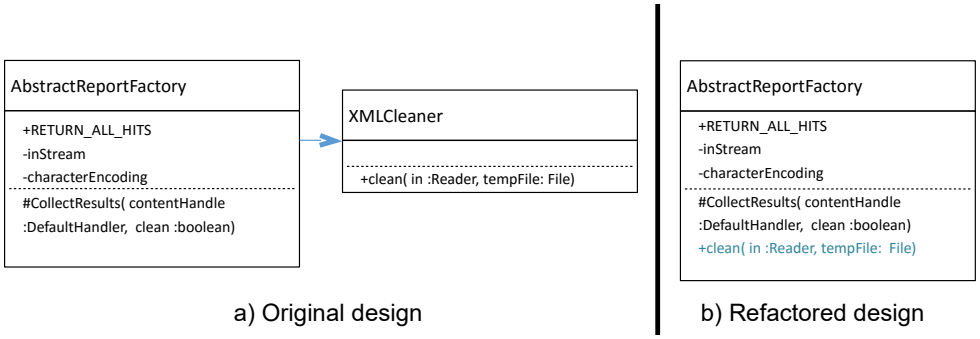

Figure 4.2 An example of Lazy class and its corresponding refactoring.

where low level refactorings are combined without targeting an specific anti-pattern, the sequence of refactorings operations generated by ReCON contains all necessary steps to remove a particular type of anti-pattern. Before applying a refactoring operation, we check if it satisfies a set of pre- and post-conditions to preserve the semantic of the code. For example, one precondition is that we do not inline parent classes, as inlining such classes will introduce regression in the children. An example of post-condition is that after inlining a LC, there is any class in the system with the same signature of the LC. Other quality attributes, such as cohesion are also considered by our approach when applying a refactoring operation. For the inline class example, we select a destiny class that is related to the LC as much as possible. To select such a class, we iterate over all the classes in the systems, searching for methods and attributes that access the LC features directly, or by public accessors (getters or setters). From those classes we choose the one with the large number of access to the LC.

Long parameter list classes are classes that contain one or more methods with an excessive number of parameters, in comparison with the rest of the entities. DECOR defines a threshold to detect when a method have excess of of parameters, based on the computation of boxplot statistics involving all the methods in the system. For example, class `RemoteIssue` from Mylyn project (shown in Figure 4.3) has 21 parameters in its constructor, making it hard to understand and maintain.

The refactoring strategy consists in (1) extracting a new class for each long-parameter-list-method, that will encapsulate a group of parameters that are often passed together, and that can be used by more than one method or classes (improving the readability of the code); (2) updating the signature of each method to remove the migrated parameters, and update the callers and method body in the LP class, to instantiate and replace the parameter with the new parameter object.

Spaghetti code classes are those classes that implement long methods with no parameters

```
1  public RemoteIssue(java.lang.String id,
2          org.eclipse.mylyn.internal.jira.core.wsdl.beans.RemoteVersion[] affectsVersions,
3          java.lang.String assignee,
4          java.lang.String[] attachmentNames,
5          org.eclipse.mylyn.internal.jira.core.wsdl.beans.RemoteComponent[] components,
6          java.    util.Calendar created,
7          org.eclipse.mylyn.internal.jira.core.wsdl.beans.RemoteCustomFieldValue[] customFieldValues,
8          java.lang.String description,
9          java.util.Calendar duedate,
10         java.lang.String environment,
11         org.eclipse.mylyn.internal.jira.core.wsdl.beans.RemoteVersion[] fixVersions,
12         java.lang.String key,
13         java.lang.String priority,
14         java.lang.String project,
15         java.lang.String reporter,
16         java.lang.String resolution,
17         java.lang.String status,
18         java.lang.String summary,
19         java.lang.String type,
20         java.util.Calendar updated,
21         java.lang.Long votes) { ... }
```

Figure 4.3 An example of Long-parameter list constructor detected in Mylyn.

at all, abusing of old procedural programming paradigm, and neglecting the advantages of object-oriented programming. Hence, the proposed refactoring strategy includes the extraction of one or more long methods as new objects. This requires creating a new class for each long method, where the local variables become fields, and a constructor that takes as a parameter a reference to the SC class; the body of the original method is copied to a new method *compute*, and any invocation of the methods in the original class will be referenced through the parameter (stored as final field) to the SC class. Finally, the *original* long method is replaced in the SC class by the creation of the new object, and a call to the *compute* method. Note that we updated the detection rule of spaghetti code defined in SAD to better reflect the definition in the literature [3], where it is stated that spaghetti code is a class with no hierarchy that declares long methods with no parameters. However, the detection condition for *method with many parameters* in SAD is set to number of parameters *inferior* to five. We modified the condition to methods with number of parameters equal to zero, to avoid detecting false positives of this anti-pattern. Note that we did not find instances of this anti-pattern in any of the projects studied, using neither the original nor the suggested fix, and for that reason we cannot provide any example.

In the case of classes affected by speculative generality, the definition states that there is an *abstract class* that is specialized *only by one* class, mainly for handling future enhancements that are not currently required, and thus it is not worthy to keep both classes in the system. We can observe this anti-pattern when we find a subclass and superclass that looks-alike. For example, consider the two classes named `AbstractHandler` from packages `org.eclipse.core.commands` and `org.eclipse.ui.commands` in Platform project (Figure 4.4). We can observe that these classes are practically the same. In addition, there is no other class that inherits from the parent class located on the package `org.eclipse.core.`

`commands`. Hence this case is an ideal candidate to apply CH refactoring.

To perform CH, we first remove the keyword *abstract* from the parent class; next, we pull up the methods and attributes from the child class to the parent class; remove the child class from the system; and replace the type's definitions and call sites from child to the parent class. We discard this refactoring when the child class is defined as inner class inside another class. Inner classes are an integral part of the event-handling mechanism in user interfaces events [137], which is far different from the concept of SG anti-pattern.

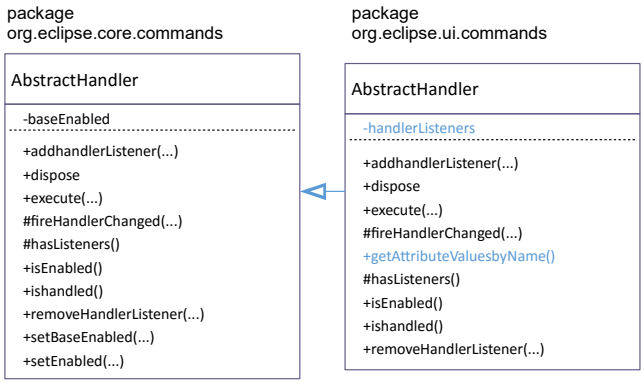

Figure 4.4 An example of Speculative Generality anti-pattern.

The objective function that we use to guide the search for refactoring sequences is presented in Equation (4.1) [36]. It measures the number of anti-patterns removed in comparison with the maximum number of anti-patterns that can exist in a system. We choose this fitness function because it is easy to implement and it is an inexpensive way to measure the effectiveness of our approach with respect to the correction of anti-patterns.

$$DQ = 1 - \frac{NDC}{NC \times NAT}, \tag{4.1}$$

where NDC is the number of classes that contain anti-patterns, NC is the number of classes, and NAT is the number of anti-pattern's types. This objective function increases when the number of anti-patterns in the system is reduced after applying the proposed refactoring sequence. The output value of DQ is normalized between 0 and 1.

Note that the objective function (the number of anti-patterns removed) depends in a non-trivial way on the code of the original and the refactored version. This fact makes it difficult to model the problem using a closed algebraic expression and, thus, limits the kind of algorithms and techniques we can use to solve the problem. In particular, mathematical programming techniques are difficult to apply in this case, since it requires constraints and objective func-

tions given as closed algebraic expressions. Any algebraic model of the objective function should probably take into account too much detail of the source code, which would increase the number of variables and constraints of the potential mathematical program up to the point that it is too large to be solved in a reasonable time. For this reason, we follow a black box optimization approach, where the quality of the solutions proposed is given by an objective function which detects on-the-fly anti-patterns in the refactored code. Metaheuristic algorithms [138] are among the most successful techniques to apply in the context of black box optimization. In the next paragraph we will detail the representation of the solutions, and the parameters of the algorithms used in this chapter to find the best sequence of refactorings.

Solution representation

To represent a candidate solution, we use a vector representation where each element represents a refactoring operation (**RO**). In Table 4.3 we present a synthetic example. We include a *Id* field, which is an integer number assigned to each refactoring operation in our generated list of refactoring opportunities. The optimization algorithm uses this Id to know which refactorings have been applied in the sequence and what ROs can be applied (valid movements in the search space). We also include the anti-pattern's source class, and the type of refactoring. The type of refactoring is used for determining if there is any conflict with any previous RO in the sequence. In addition to this, and according to the refactoring type, we can have more fields providing additional information, *e.g.,* qualified name of long-parameter-list methods, in the case of LP class; children class for a class a class containing Speculative generality anti-pattern, etc.

Table 4.3 Representation of a refactoring sequence.

ID	Source class	Type	Other fields
9	ExtWindowsMenuUI	Introduce Parameter Object	List of long-parameter-list methods
26	RangeSearchFromKey	Inline class	Target class
45	ProjectResource	CollapseHierarchy	Children class
16	ActivityContextManager	Replace method with method object	Long method(s) name

Variation operators

Simulated annealing. It employs one variation operator, *a.k.a.* perturbation operator, which consists of choosing a random point in a sequence, then we remove the refactoring operations from that point to the end, and finally we regenerate the sequence until we cannot add more refactoring operations. To ilustrate this procedure consider the example show in Figure 4.5.

We define this strategy, because an arbitrary transformation of a refactoring operation in the sequence, like the one implemented in binary strings, will lead to semantic inconsistencies given that in refactoring the order is important, *e.g.,* one refactoring could block further refactorings. Moreover, it is cheaper to add operations from a starting point (in the worst case the first refactoring operation) than verifying semantic correctness backwards, and finally, it brings more diversity which is the ultimate goal of perturbing a sequence.

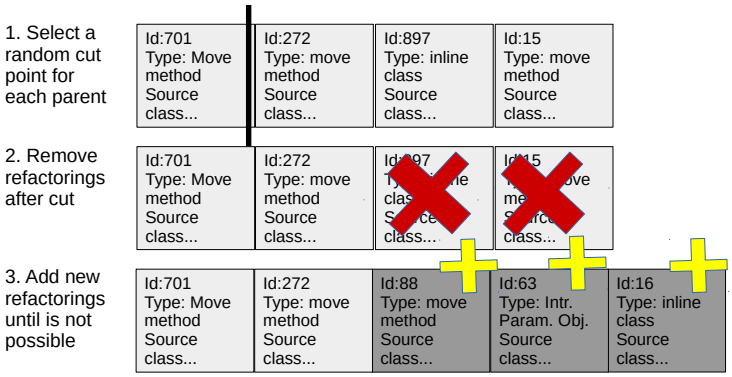

Figure 4.5 Example of perturbation operator

Genetic algorithm. It employs two variation operators, crossover and mutation. To select the best individuals to perform the crossover, we use the "binary tournament" technique. The crossover operator is "Cut and splice" [44, 69] technique, which consists in randomly setting a *cut point* for each parent, and recombining with the rest of elements of the other parent's cut point and vice-versa, resulting in two individuals with different lengths. An example of this operator is shown Figure 4.6. Note that when a refactoring operation is conflicted with a previous one in the sequence, we just drop it.

The mutation operator follows the same strategy than the perturbation process implemented in our version of simulated annealing, rather than the one proposed in in [44, 69] because we found that the former one was unable to find complete solutions, *i.e.,* the ones that removes all anti-patterns in a reasonable amount of time.

Variable neighborhood search. It uses the same perturbation operator of SA to alter a solution in the *kth* neighborhood.

Parameters of the metaheuristics. We use three metaheuristic techniques in our case study. As we mentioned before, they make use of different settings to move through the decision space in the search for an optimal solution. To determine the best parameters for the techniques employed, we run each algorithm with different configurations 30 times, following a *factorial design.*

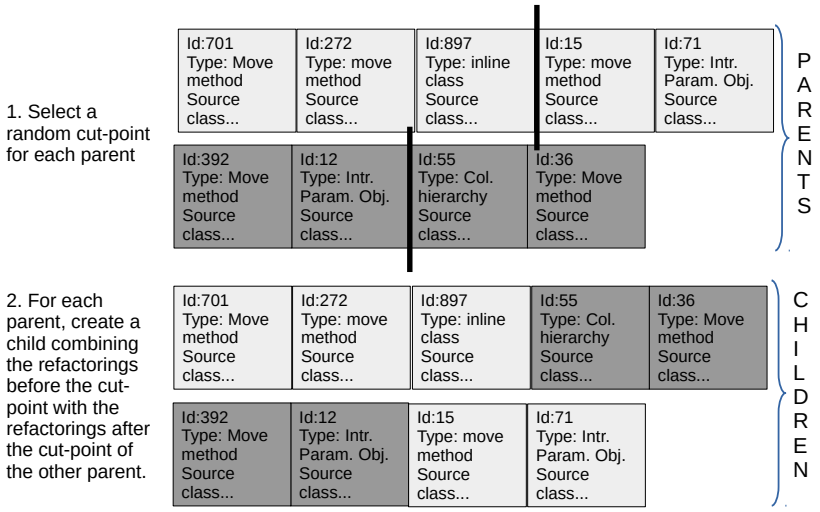

Figure 4.6 Example of crossover operator

In the case of GA we test 16 combinations of mutation probability p_m = (1, 0.8, 0.5, 0.2), and crossover probability $p_c = (1, 0.8, 0.5, 0.2)$, and obtained the best results with the pair (0.8, 0.8). This is not a surprise as in [69] they found high mutation and crossover values to be the best trade for algorithm performance.

For SA, we set the initial temperature to 10,000,000, and tried three different values of cooling factor (CF), $CF = (0.990, 0.993, 0.996, 0.998)$ and found the best results with the latter one.

For VNS we define a maximum number of neighborhoods $maxK = 100$. Note that in our VNs implementation, local search operates at the level of the last j refactoring operations. We tried different values for the local factor $j = (2, 4, 6, 8)$, and found the best results with 2.

For the specific problem of automated refactoring, setting the initial size of the refactoring sequence is crucial to find the best sequence in a reasonable time, especially when we have a huge number of candidate refactorings, because setting a low value will lead to find poor solutions in terms of anti-patterns correction. On the contrary, if the initial size is very large, we may obtain the reverse effect because applying many refactorings not necessarily implies better quality, as refactorings can improve one aspect of quality while worsen others. Hence, we experiment running the algorithms with three relative thresholds: 25%, 50%, 75% and 100%, of the total number of refactoring opportunities. We found that 50% give us the best results in terms of removal of anti-patterns.

Finally, the number of iterations for all the algorithms is set to 1000. The population size for GA is set to 100 individuals as typically used value in other refactoring works [139], and

the selection operator used is binary tournament.

With this information, the map of anti-patterns and the relevant code entities, we automatically generate a list of candidate refactorings. The list of candidate refactorings, and the abstract model are the input of the search algorithm. The search algorithm generate a set of refactoring sequences. The refactoring sequences are evolved using the corresponding variation operators. All the candidate sequences are applied to a copy of the PADL model. Then the number of anti-patterns in the resulting model are computed, and sequence is evaluated using the objective function. The process finish when the stop condition is met. The final output is the best refactoring sequence for the current execution.

4.4.4 Analysis Method

We examine two scenarios: (1) developers perform a dedicated refactoring session after the completion of the task (*i.e.,* root-canal refactoring) and (2) developers intersperse refactorings among other changes during the task activity (*i.e.,* floss refactoring). In the first scenario, we generate refactoring candidates for all existing classes contained in a snapshot. For the second scenario, we only generate refactorings that affect the classes contained in the interaction history (developer's context). The generated refactorings aim to correct any of the four anti-pattern types studied. As we explained before, the interaction histories are associated to a developer's patch aimed to fix a bug. Hence, we apply the corresponding patch to each snapshot, before generating the candidate refactorings, to ensure that they are applied on a stable version of the code.

Due to the random nature of metaheuristic techniques employed in this chapter, it is necessary to perform several independent runs to have an idea of the behavior of the algorithms. We execute 30 independent runs, which is a typically used value in the search-based research community.

We also compare the performance of the metaheuristics employed with random search to make sure that they can find better solutions than a pure random approach.

4.4.5 Results of the Experiment

This section presents and discusses the results of our experiment.

Individual task context versus accumulated task context

After applying their corresponding patch to each task snapshot, we perform floss refactoring (as described in Section 4.4.4) and compare the count of anti-patterns before and after refactoring to assess the benefits of ReCon. In Figure 4.7, we present box plots with the distribution of anti-patterns occurrences for the 657 studied tasks taken from the Mylyn project. The anti-pattern's occurrences correspond to all the classes in the snapshot. But the refactorings applied only considered the relevant classes of each task. Hence, We observe a small reduction in the number of anti-patterns. This result was expected because the number of relevant files for each task (*i.e.,* the developer's context) is small compared to the number of classes in a snapshot, and consequently, the number of refactoring candidates too.

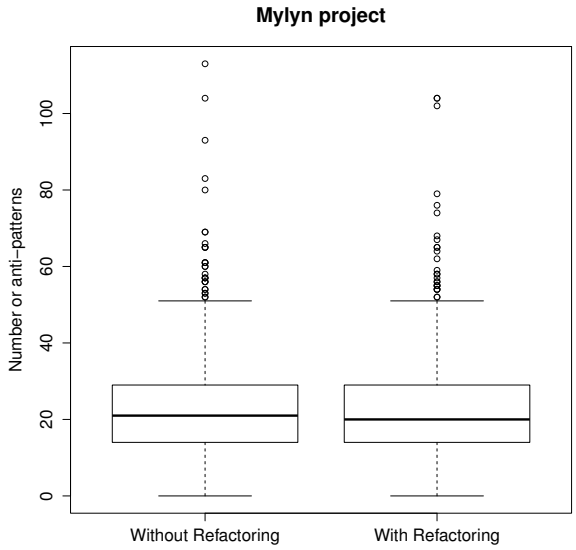

Figure 4.7 Distribution of anti-pattern's occurrences before and after refactoring based on task context.

However, the accumulation of these small improvements (*i.e.,* reductions of anti-patterns occurrences) over a long period of time is likely to result in a significant improvement of the design quality of the system. To verify this hypothesis, we accumulate the contexts of all the individual tasks from the oldest to the most recent (ordered based on commit dates) and apply our automated floss refactoring approach using the accumulated context. This allows us to measure the accumulated impact of floss refactoring. We compute and compare the count of anti-patterns for (1) the source code without refactoring, (2) the source code after applying all the floss refactorings, and (3) the source code after performing a root-canal refactoring. In Figure 4.8, we present a comparison of the the anti-pattern's count for our three projects before and after refactoring (floss and root-canal). To verify if the observed differences

(between the number of corrected anti-patterns for root-canal versus floss refactoring) are statistically significant, we performed a Wilcoxon rank sum test [140] at 95% confidence level (*i.e.,* $\alpha = 5\%$). The test was statistically significant (*i.e.,* p-value<0.05), indicating that the distribution of the results is not the same for both groups. We also evaluated the magnitude of the difference by computing the Cohen's δ effect size (ES) [141]. The results show that the difference is large for the three projects (ES$\geq$ 0.8.).

Overall, we observe that our proposed automated floss refactoring approach can reduce approximately 50% of anti-patterns. This is a significant reduction considering the fact that it does not disrupt the developer's work flow, since it only recommends refactorings that affect files on which the developer is already working (i.e., files from the task context).

On the contrary, relying on root-canal refactoring is expensive. The number of refactoring opportunities detected go from 167 (Mylyn) to 2068 (Platform). However, applying floss refactoring with ReCon can alleviate this cost. From the individual tasks studied in this work we found that the tasks with more refactoring opportunities are: Mylyn task number 87670: 34; PDE task number 84503: 50; and Platform task number 82540: 63. These number of refactorings, which might not be trivial to be generated manually, are feasible to be evaluated and applied for a developer with the help of our approach.

Nevertheless, after applying ReCon during the development and maintenance of a software system, developers can still perform a root-canal refactoring prior to the release of the system to remove the remaining anti-patterns. Figure 4.8 shows that a root-canal refactoring can be very effective at removing anti-patterns in a system. After the root-canal refactoring of Mylyn, PDE, and Platform, only respectively 1, 4, and 8 anti-patterns remained in the projects. We manually inspect these cases, and found that the anti-pattern remaining in Mylyn, that is a LP instance, was not removed because is inside an inner class for which our implementation of introduce parameter object is not suitable; in PDE two instances of lazy class could not be removed due to an issue with a missing package name; in Platform, half of the anti-patterns of SG type were not corrected because they refer to abstract classes belonging to external APIs (`java.util`, and `java.io`). Beside this drawbacks, we consider that the ReCon results are stable, and not biased towards any anti-pattern type.

In the following, we will analyze the performance of the metaheuristics employed in this work, and their corresponding resources consumption.

In Table 4.4 we present the average count of anti-patterns of the 30 independent runs for the three metaheuristic algorithms and random search in the accumulated floss refactoring sce-

Table 4.4 Count of anti-patterns after applying floss refactoring.

Anti-pattern	Original	GA	RS	SA	VNS
MYLYN					
SG	0	0	0	0	0
SC	0	0	0	0	0
LC	27	19	25	19	19
LP	140	49	94	49	49
Total	167	68	119	68	68
PDE					
SG	31	2	3	2	2
SC	0	0	0	0	0
LC	1205	180	193	180	180
LP	2276	1229	1320	1229	1229
Total	3512	1411	1516	1411	1411
PLATFORM					
SG	30	22	23	22	22
SC	0	0	0	0	0
LC	1242	336	341	336	336
LP	2286	1595	1651	1595	1595
Total	3558	1953	2015	1953	1953

nario. As we can observe, the three metaheuristics are capable of removing the same number of anti-patterns, thought with some variations in the amount of memory and execution time required.

With respect to the instances of anti-patterns removed, there is little difference between the refactoring solutions found by each different metaheuristic, especially if we consider that the detection and generation of refactoring operations process is the same. However, the CPU time, and to some extent the memory consumption, that one algorithm takes to find the best combination of the refactorings is where we found more interesting differences. To corroborate this point, we manually compare the refactorings sequences and found that most of the differences are related to the position in which each metaheuristic includes them in the sequence. This is true for the set of refactorings that are not conflicted, and do not required an specific order to be applied.

We also observe that metaheuristics overcome random search in all the projects studied. To corroborate this result, we apply the same statistical test, Wilcoxon rank sum and Cohen's δ effect size (ES), and found that the results are statistically different (p-value$<$0.05), and that difference between the metaheuristics and random search is medium (ES = 0.07) in terms of anti-patterns correction.

The resources usage is depicted in Figure 4.9 for each metaheuristic. We can observe that SA has the fastest execution among the three metaheuristics followed close by GA. We apply Wilcoxon test and Cohen's δ ES, and found that this result is statistically significant in comparison with GA and VNS, and with a large difference (ES = 1.09, 7.31). Concerning memory usage, the difference is also significant, but with a small difference for GA (ES =

0.037) and large for VNS (ES =5.68).

In a scenario where developers are more interested in obtaining a solution fastest, SA is the recommended algorithm. GA consumes less memory but with more variability in the execution time. VNS report the highest values for memory consumption and execution time, given that it has to analyze many neighborhoods before finding an optimal solution. In any case the execution time required to perform floss refactoring using context in each individual task is less than 200 seconds in average (in case someone opts for VNS), which is acceptable when performing a coding task.

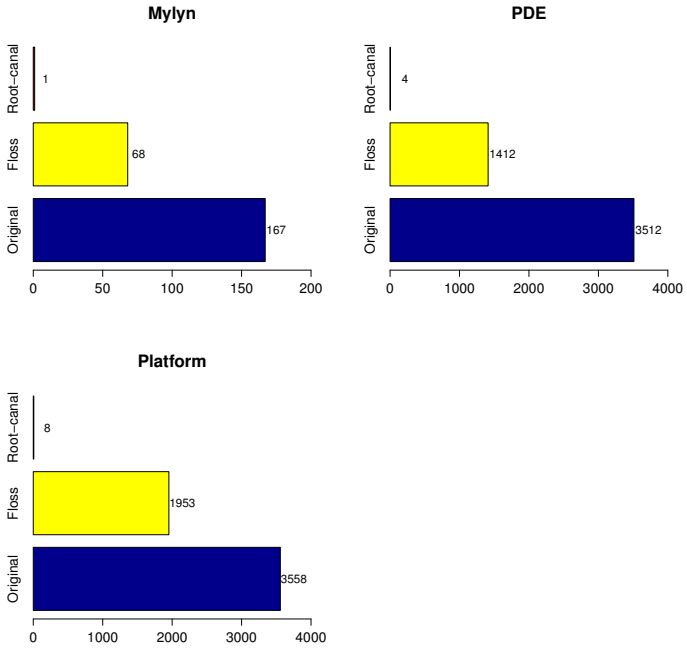

Figure 4.8 Anti-patterns occurrences after applying floss and root canal refactoring.

Performance of the algorithms

Finally, in Table 4.5 we present the resources usage for root-canal using SA metaheuristic, as it is the one to find solutions in the shortest time. As we can expect, the execution time and memory required to perform is bigger for root-canal refactoring, and these values increase proportionally to the number of classes in the studied project. This is expected since we look for refactoring opportunities in all the classes in the system in root-canal refactoring, while in floss refactoring we focus only on classes that are in the developer's context. It is clear that there is a trade-off to make between the quality achieved and resources consumed, as

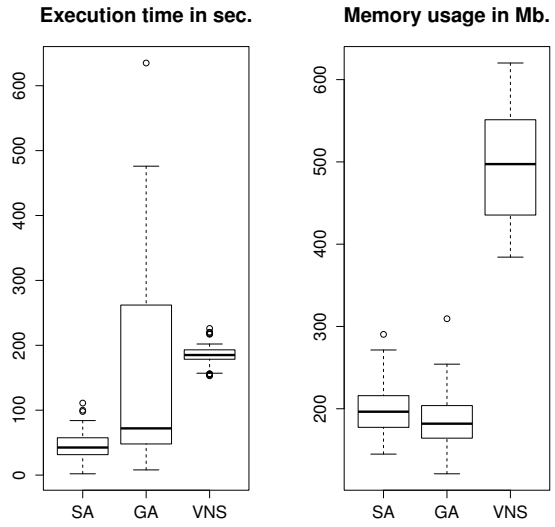

Figure 4.9 Resources consumption for each Algorithm when performing floss refactoring.

the number of anti-patterns removed is less for floss refactoring.

Table 4.5 Resources usage for root-canal using SA.

Program	Memory usage (Mb).	Execution time (hh:mm:ss)
Mylyn	933.78	00:48:58
PDE	4505.83	10:44:15
Platform	5936.74	14:09:01

Quality evaluation

After analyzing our approach in terms of memory usage and execution time, we also consider important to assess the impact on the quality of the programs analyzed. For this purpose, we use the QMOOD [40] (*cf.* Section 2.4) to evaluate the effect of the proposed refactoring sequences on five quality attributes.

To compute the quality gain, we use the formula proposed in [142] where the total gain in quality G for each of the considered quality attributes q_i before and after refactoring is estimated as:

$$G_{qi} = q'_i - q_i, \tag{4.2}$$

where q'_i and q_i represents the value of the quality attribute i after and before refactoring.

In Figure 4.10, we can observe the quality gain obtained for each selected QMOOD attributes after applying root-canal and accumulated floss refactoring using context. In both cases, the

quality increases according to the five attributes. Reusability is the quality attribute that has the highest gain, while effectiveness has the lowest one (0.01,0.009), follow by flexibility (0.27, 0.19). We suggest that the negligible gain in effectiveness is due to the combination of metrics that does not penalize coupling like (DCC), that is impacted by the refactorings proposed in the case study. On the contrary, we observe that extendibility, which penalizes DCC with -0.5, show better results (0.46, 0.34). The low gain in flexibility is presumably due to the fact that a big portion of the weight of that quality attribute is on the *Number of polymorphism methods (NOP)* metric. This metric refers to methods that are overridden by one or more descendent classes. Since the refactorings applied on the programs do not override existing methods, as it is not required by the definition of the anti-patterns analyzed, the increment of this quality attribute is small. On the contrary, the reusability attribute which gives a high weighing to *Design Size* (Number of classes), and *Messaging* (communication between classes) metrics benefits from the decomposition on long parameter list, which is one of the most predominant anti-patterns in the three studied projects. Finally, the substantial increment in understandability reflects a drop in the complexity of the design structure. Understandability is one of the most desired attributes to achieve from the point of view of developers, as it eases the addition of new features and enhancements.

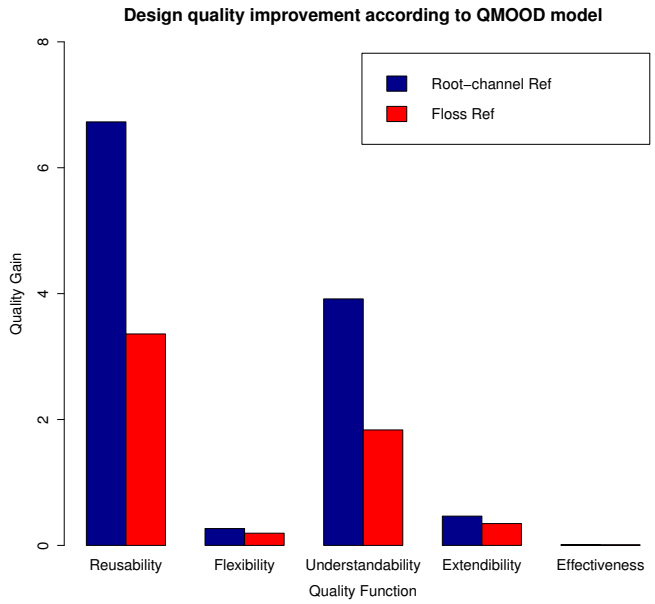

Figure 4.10 The impact of the best refactoring solutions on QMOOD quality attributes.

To summarize this section, we conclude that ReCON can successfully improve the quality of a software system, not only with respect to the number of anti-patterns corrected, but also in terms of reusability, understandability, and to a minor extent, flexibility.

4.5 Discussion

Results from Section 4.4.5 show that our proposed approach ReCon is effective at correcting anti-patterns in software systems. ReCon can find refactoring solutions in a reasonable time using a reasonable amount of resources. The main contribution of ReCon is leveraging task context information to prioritize the refactoring of classes that undergo changes more often. This is especially convenient if we consider that the length of the sequence of refactorings is shorter in a floss scenario than a root canal one. The complexity of the scheduling of the refactorings is also simplified, as the number of possible conflicts is reduced, and finally it does not make too much sense to modify classes that do not change very often.

By contrasting the quality of the resulting design before and after refactoring using Recon in floss and root-canal scenarios give us an insight of the usefulness of the refactorings proposed not only in terms of anti-patterns correction, but in other quality attributes like coupling and design size.

Concerning floss and root-canal scenarios, one interesting finding is the distribution of anti-patterns among the classes that are touch by developers during a task context. For example, for Mylyn project, which has a total of 2365 classes, we covered 72% of them in the floss accumulated scenario (1697) and remove approximately 59% of anti-patterns; that means that 28% of the classes, which were not modified in our collected dataset, contain 41% of anti-patterns. The coverage of classes in PDE and Platform is considerably less, 24% and 11%; however, the remaining anti-patterns in the untouched classes are 40% and 55% respectively. These results suggest that for PDE and Platform, 60% of the anti-patterns studied are concentrated in a small portion of the system.

Finally, ReCon do not require any set of bad code examples to work like previous approaches [24, 36, 130, 142], so it can be used directly out of the box. Another advantage of ReCon is that the thresholds used for detecting the analyzed anti-patterns, can be easily modified according to the user needs through SAD, without modifying any line of code in the implementation of ReCon.

4.6 Threats to validity

We now discuss the threats to validity of our study following common guidelines for empirical studies [143].

Construct validity threats concern the relation between theory and observation. Our modeling approach assumes that each anti-pattern is of equal importance, when in reality, this may not be the case.

Threats to internal validity concern our selection of subject systems, tools, and analysis method. The accuracy of DECOR impacts our results. DECOR is an academic approach which has been reported to achieve high recall and reasonable precision [52]. However, other anti-pattern detection techniques and tools may provide different results. The rational behind using Mylyn's interaction histories is that Mylyn plug-in is the only tool that has been applied to several open-source systems to gather developers' interactions and these are publicly available. Note that the systems analyzed are the top-three systems with more interaction histories.

Conclusion validity threats are related to the violation of the assumptions of the statistical tests and the diversity of our dataset. We used non-parametric tests (Wilcoxon rank sum) that make no assertion about the distribution of the data. We used data from three open-source systems that have different sizes and involve many developers.

External validity threats relate to the generalization of our results. Because our subject projects are open-source and because we used a particular yet representative subset of anti-patterns as proxy for software design quality, we cannot guarantee that the findings of this study can generalize to proprietary software systems and other open-source systems. In the future, we plan to analyze more systems, including proprietary and including different programming languages, to draw more general conclusions.

Reliability validity threats concern the possibility of replicating this study. All the raw data used in this chapter are available in Eclipse Bugzilla. The software systems studied in this chapter are also available online for the public.

4.7 Chapter Summary

In this chapter, we highlighted the main drawbacks of existing refactoring approaches that propose refactoring solutions without considering developer's task context, even though previous studies (*e.g.,* [42]) have found that developers prefer refactoring suggestions that can be applied to files that are active in their workspace.

This lack of consideration for developers' context may explain the poor adoption of automated refactoring approaches in industry. In addition, many of the existing approaches require that developers input a set of bad code examples, to generate detection rules, or a desired model to generate the corresponding refactoring solution(s). These requirements put extra work on developers, slowing the refactoring process, and even rendering it impractical in certain cases. Specifically, we set out to address the following question:

Central Question: *Is it possible to improve automated refactoring by leveraging developer's task context?*

To answer this question, we proposed ReCon, an automated refactoring approach that leverages developer's context and metaheuristic techniques to compute the best sequence of refactoring that affects only entities in the developer's context. We performed a case study using three open-source systems and found that ReCon can successfully correct more than 50% of anti-patterns in a software system using less resources than the traditional approaches from the literature. More importantly, ReCon does not disrupt the developer's work flow, since it only recommends refactorings that affect files on which the developer is already working (*i.e.,* files from the task context).

We also assess the quality of the systems subject of this study before and after applying ReCon, using five quality attributes defined in the QMOOD model [40]. Results show that ReCon can achieve a significant quality improvement in terms of reusability, understandibility, extendibility and to some extent flexibility, while effectiveness reports a negligible increment.

CHAPTER 5 EFFICIENT REFACTORING SCHEDULE

5.1 Introduction

In Section 2.2.3, we formally presented the refactoring scheduling problem, and provided a formulation (Equation (2.1)) to compute the size of the refactoring search space for a set of refactoring candidate solutions generated to be applied in a software system. We also mentioned that most of the existing refactoring approaches rely on metaheuristics to solve the refactoring schedule problem. The reason is that the objective function (typically the number of anti-patterns occurrences) depends in a non-trivial way on the code of the original and the refactored version. This fact makes it difficult to model the problem using a closed algebraic expression and, hence, limits the kind of algorithms and techniques that can be used to solve the problem.

From the plethora of metaheuristic techniques, many SBSE researchers have implemented Evolutionary Algorithms (EAs) [45, 76, 144, 145, 43, 36, 26, 142] to solve the refactoring scheduling problem.

The problem of using EAs is that they typically require large number of evaluations (of solutions) to converge to optimal results. This translates into larger execution times (*cf.* section 4.4.5, root-canal refactoring), and more developer's effort to evaluate and apply the proposed solutions. To address this limitations, some researchers have proposed refactoring approaches that consider conflicts between refactorings to reduce the search-space size. For example, Liu et al. [82] proposed an approach to iteratively select the most promising refactoring operations in terms of design quality, while removing the ones that are in conflict with them, until there are no more refactorings candidates left. In another work, the same research group [83] proposed a refactoring approach for reducing the effort required for removing different type of anti-patterns using pairwise analysis. The idea is to refactor the anti-patterns that can mitigate the negative effects of other types of anti-patterns, (*e.g.,* removing code duplication also affects anti-patterns related to the size of classes/methods). These approaches will maximize the scheduling of refactorings with the higher quality effect. However, they still require developers to find a list of refactoring candidates, and to apply the list to their code.

In this chapter, we aim to close the gap, by providing full-automated refactoring support for developers, that covers all the main steps of the improvement of software design quality through automatic refactoring, i.e., the (1) detection of classes that contain anti-patterns;

(2) the generation of refactoring candidates to improve the design quality of the classes detected in (1); (3) the search for an optimal refactoring order; and (4) the application of the refactoring order from (3).

We, therefore, set out to address the following question:

Central Question: *Is it possible to improve automated refactoring by proposing an efficient scheduling?*

To achieve this goal, we propose a new heuristic approach called RePOR (**Re**factoring approach based on **P**artial **O**rder **R**eduction). Partial order reduction is a popular technique for controlling state space explosion in model checking [146]. The intuition is to reduce the number of refactoring sequences to be explored by removing equivalent sequences (*i.e.,* refactoring sequences that leads to the same design). As a result, less search effort is required than well-known metaheuristic approaches. To evaluate RePOR, we conduct a series of experiments over a testbed of five OSS and compare the results with Genetic Algorithm (GA) [147], Ant Colony optimization (ACO) [148], and the conflict-aware refactoring scheduling approach proposed by Liu et al. [82] (referred to as LIU in this chapter). We show that the solutions obtained by RePOR overcome the ones obtained by the above-mentioned state-of-the-art optimization techniques in terms of performance (i.e., execution time) and effort (i.e., number of refactorings applied).

5.2 Reducing the search-space size of the refactoring scheduling problem

As we discuss in Section 2.2.3, search algorithms start by generating one or more random sequences. Next, the quality of each sequence is computed by applying it to the software system in question, and measuring the improvement in the quality attributes of interest using an objective function (*a.k.a.*fitness function).

We can evaluate the quality of a refactoring sequence SR using the formulation presented in Equation (5.1).

$$Q(SR) = \sum_{k \in K} Q(sr_k);\ \text{with}\ Q(sr_k) = AC(k') - AC(k) \tag{5.1}$$

Where SR is a subset of R; R is the set of refactorings to be applied in a system SYS; K is the set of classes in SYS, $K \in SYS$; sr_k is a subset of SR that modifies class k ($k \in K$).

Each sub-function $Q(sr_k)$ is computed by subtracting the number of occurrences of anti-patterns in class k after applying sr_k to k (*i.e.,* $AC(k')$) and the number of occurrences of anti-patterns before refactoring (*i.e.,* $AC(k)$) . Note that we use the number of occurrences of anti-patterns as a proxy of design quality. The outcome of $Q(SR)$ is related to the presence and the order of refactorings in SR.

Hence, we suggest that refactorings should be clustered depending on the classes that they affect. In this way, they can be optimized separately. Since the order of appearance of refactorings that affect different classes in a sequence is irrelevant, we can reduce the number of refactoring operations that we need to evaluate. Let us reuse the example proposed in Section 2.2.3, that is a set of refactorings: $R = \{A, B\}$ to be scheduled. We recall that applying Equation (2.1) to our example gives us 5 possible sequences: $<>$, $< A >$, $< B >$, $< A, B >$, $< B, A >$, if and only if (iff) we assume that each permutation leads to a different solution. Otherwise, $< A, B >$ and $< B, A >$ are two different representations for the same solution, which leaves us with only 4 different solutions. We also mentioned that having refactorings that affect the same class, the resultant design may vary depending on the order of application of those refactorings.

Let us represent the dependency between refactorings as an undirected graph G_D, where an edge $(r_u, r_v) \in E$ exists iff $r_u, r_v \in R_k$. $k \in K$, where K is the set of classes in a system, and R_k is the set of refactorings that affect class k, $R_k \subset R$.

We use G_D to find the connected components ($CCAP$). A connected component is a maximal subgraph where all the pairs of vertices are connected by a path. Connected components impose a partial order over the refactoring operations. We borrow the idea of *partial order reduction* from model checking [146], to express the removal of sequences of refactorings that lead to the same design.

The set of refactorings R and classes K can be used to form a *bipartite* graph $G_{B=(R,K,E)}$, where each refactoring $r \in R$ is connected to the classes it affects. G_B is linked to the structure of the objective function, where a set of refactorings modify a class, and the application or not of these refactorings affect the number of anti-patterns existing in this class.

Another factor that affects the size of the search-space of the refactoring problem is the occurrences of conflicts. We distinguish to kind of conflicts, sequential dependency conflicts and mutual exclusion conflicts. We elaborate more on these two kind of conflicts in the following.

— Given two refactorings r_i and r_j, r_i has a sequential dependency conflict with r_j iff r_j cannot be applied before r_i. We represent sequential dependency conflicts as follows: $r_1 \rightarrow r_2$, which means that r_1 can be followed by r_2, but r_2 cannot be followed by r_1.

Note that conflicts are directional, i.e., the fact that applying r_j disables r_i does not necessarily means that r_i disables r_j.

— Given two refactorings r_i and r_j, r_i has a mutual exclusion conflict with r_j iff r_i and r_j cannot be applied together in any order. We represent mutual exclusion with the following notation: $r_1 \not\leftrightarrow r_2$.

In the extreme case where no conflicts exist among the pairs of refactoring opportunities (i.e., all pairs commute), only the presence or absence of a refactoring opportunity in a sequence makes a difference in the sequence, and the search space can be reduced to 2^n refactoring sequences.

We model the conflicts between refactorings using a directed graph G_C, where the set of refactoring opportunities R is the set of vertices and an edge $(u, v) \in E$ exists between two refactorings u and v if there is a conflict between refactorings u and v.

To better illustrate the refactoring scheduling problem, and the effect of considering dependencies and conflicts between refactorings has on the size of the search-space, we present a motivating example in Listing 5.1.

Listing 5.1 A refactoring motivating example

```
1  class Geometry{
2   double calcAreaRectangle(Rectangle p1){
3          return p1.Width()*p1.Height();
4   }
5   void longParameterListMethod(int p1, int p2, ..., int p15 ){
6          ...
7   }
8  }
9  class Rectangle{
10     private double width;
11     private double height;
12     public double Width(){
13         return width;
14     }
15     public double Height(){
16         return height;
17         }
18 }
19 class Shape{
20         ...
21 }
```

The refactorings presented in Table 5.1 can be applied to refactor the classes described in Listing 5.1.

Applying Equation (2.1) to the example shown in Listing 5.1, we have that the number of refactoring sequences is $S = \lfloor e \cdot 3! \rfloor = \lfloor 16.3097 \rfloor = 16$. A simple manual enumeration, shown

Table 5.1 List of refactorings candidates for the example from Listing 5.1

ID	Type	Source class	Method	Target Class
r_1	Move method	`Geometry`	`calcAreaRectangle`	`Rectangle`
r_2	Inline Class	`Rectangle`	All fields and methods	`Shape`
r_3	Introduce Parameter Object	`Geometry`	`longParameterListMethod`	`GeometryParamObj` *(new)*

in Table 5.2 confirms this evaluation.

Table 5.2 Enumeration of possible refactoring sequences for the set of refactoring operations {r1, r2, r3}.

sequence	elements	sequence	elements
1.	None	9.	r_3, r_1
2.	r_1	10.	r_3, r_2
3.	r_2	11.	r_1, r_2, r_3
4.	r_3	12.	r_1, r_3, r_2
5.	r_1, r_2	13.	r_2, r_1, r_3
6.	r_1, r_3	14.	r_2, r_3, r_1
7.	r_2, r_1	15.	r_3, r_2, r_1
8.	r_2, r_3	16.	r_3, r_1, r_2

Note that in Equation (2.1) we assume that a permutation of a subset of refactoring operations always leads to a different software design. However, this assumption may not holds in all cases. In Table 5.2 we find pairs of refactorings where the application order is irrelevant, *e.g.,* the application order of r_1 and r_3 in sequences 6, 9. Hence, it is possible to reduce even more the search-space by removing these permutations as they lead to the same design (same solution). This occurs because they affect different code segments (the method and target class is different for r_1 and r_3) , *i.e.,* they are unrelated.

In addition, when a conflict exists between refactorings, it is possible to reduce the size of the search space further. For example, consider the sequential dependency conflict between r_1, r_2, that is r_2 cannot be applied before r_1 (inlining class `Rectangle` invalidates any move method refactoring from/to that class). Hence, by removing redundant solutions, and invalid solutions (solutions with elements that are conflicted) we can reduce the search-space size of the motivating example by half (sequences 1, 2, 3, 4, 5, 6, 8 and 11). Thus, the value obtained after applying Equation (2.1) should be used as an upper bound of the search-space size, as long as we assume that applying a refactoring sequence does not create new refactoring opportunities that were not in the original list. If this happens, the number of possible refactorings can be larger than $\lfloor e \cdot n! \rfloor$. However, in a typical scenario, software maintainers would repeat the process of finding refactoring opportunities until: (1) it is not possible to apply more refactorings, or (2) they are satisfied with the design quality.

5.3 Refactoring approach based on Partial Order Reduction

In this section we present the foundations of RePOR. RePOR is comprised of 7 steps depicted in Algorithm 1

Algorithm 1: RePOR

```
   Input  : System to refactor (SYS), Maximum number of permutations (threshold)
   Output: An optimal sequence of refactoring operations (SR)
 1 Steps RePOR(SYS, threshold)
 2 |   AM=code-design model generation (SYS)
 3 |   A = Detect Anti-patterns(AM)
 4 |   R = Generate set of refactoring candidates(AM, A)
 5 |   G_D = Build Graph of dependencies between refactorings and anti-patterns(AM, R, A)
 6 |   CCAP = Find connected components (G_D)
 7 |   G_C = Build Graph of conflicts between refactorings (AM, LR)
 8 |   SR = Schedule sequence of refactorings(CCAP, G_C, AM)
 9 Procedure Schedule sequence of refactorings(CCAP, G_C, AM):
10 |   SR = 0
11 |   for each ccap ∈ CCAP do
12 |   |   ccap.RemoveInvalidRefactorings(SR)
13 |   |   if ccap.size == 0 then
14 |   |   |   continue
15 |   |   else
16 |   |   |   List permuts = enumeratePermutations(ccap)
17 |   |   |   if permuts ≤ threshold then
18 |   |   |   |   SR.addAll(extractBestPermutation(AM, G_C, permuts))
19 |   |   |   else
20 |   |   |   |   SR.addAll(getFirstValidSequenceFromccap(AM, G_C, ccap, R))
21 |   |   |   end if
22 |   |   end if
23 |   end for
24 |   return SR
25 end
```

5.3.1 Step 1: Code-design model generation

In this step we generate a light-weight representation (a code meta-model) of a software system (SYS), using static code analysis techniques, with the aim of evolving the current design into an improved version in terms of design quality. A code meta-model describes systems at different levels of abstractions. We consider three levels of abstractions to model systems. A code-level model (inspired by UML) which includes all of the constituents found in any object-oriented system: classes, interfaces, methods, and fields. An idiom-level model which is a code-level model extended with binary-class relationships, detected using static analysis. A design-level model that contains information about occurrences of design motifs or of code smells and anti-patterns. A code-meta model must differentiate among use, association, aggregation, and composition relationships. It should also provide methods to manipulate the design model and generate other models. The objective of this step is to manipulate the design model of a system programmatically. Hence, the code meta-model is used to detect anti-patterns, apply refactoring sequences and evaluate their impact in the

design quality of a system. More information related to code meta-models, design motifs and micro-architecture identification can be found in [149, 150].

5.3.2 Step 2: Detect Anti-patterns

In this step we detect anti-patterns in the meta-model using any available detection tool. The output of this step is a set of anti-patterns instances (A), with the qualified name of the classes and constituents that participate in each detected anti-pattern.

5.3.3 Step 3: Generate set of refactoring candidates (R)

After we generate a set of anti-patterns that we want to correct from the previous step, we generate a list of refactoring operations based on the type of anti-patterns. For example, in the case of a Blob class, which is a large controller class surrounded by data classes, we may start by moving functionality to related classes in order to reduce size and improve cohesion using *move method* refactoring. We may have more than one possible targets to move a method from the Blob class, which become refactoring candidates, and our approach should be able to select the move method refactoring that improves the most the design quality of the system after refactoring.

5.3.4 Step 4: Build refactorings dependency graph (G_D)

To avoid evaluating permutations that lead to the same design, it is important to cluster refactorings by the classes they are modifying.

5.3.5 Step 5: Find connected components ($CCAP$)

To guide the search of refactoring operations, once we have built the refactorings dependency graph (G_D), we proceed to find the connected components of G_D.

5.3.6 Step 6: Build refactorings conflict graph (G_C)

As we mentioned before, conflicts arise when two or more refactorings affect the same classes or their constituents (fields, methods, etc.). These conflicts should be considered when generating a refactoring schedule to avoid evaluating invalid sequences.

5.3.7 Step 7: Schedule a sequence of refactorings (SR)

In this final step, we iterate over all connected components, $ccap \in CCAP$, to schedule refactorings that correct more anti-patterns (lines 11-25). At the beginning of the search, the refactorings in sequence SR is empty (line 10). During the search process, we will add refactorings to SR, that can disable other refactorings from R. Hence, we remove refactoring operations that are no longer valid in every iteration of the main loop (line 12). If the number of vertices in a *ccap* is zero after removing invalid refactorings, we continue with the next connected component. Otherwise, we compute all possible permutations of the refactorings in *ccap* (line 16). To enumerate all permutations of *ccap*, we use *Algorithm G (General permutation generator)* from Knuth [151]. This algorithm generates all permutations with the condition that every permutation is visited only once. Depending of the number of elements in *ccap*, the computation time could be too long. The input parameter *threshold* is an integer value which represents the maximum number of permutations that we can enumerate without spending too much time in the enumeration process, and this value is empirically determined according to the architecture of the test computer. If $permuts \leq threshold$, we call *extractBestPermutation* procedure to obtain the best permutation in terms of anti-pattern correction, which is depicted in Algorithm 2. In case the number of permutations is too large to be enumerated (line 19) we call method $getFirstValidSequenceFromccap$ to find the first non-conflicted sequence of anti-patterns from the current *ccap*. We depict the procedure in Algorithm 4.

Algorithm 2: Algorithm to extract the best permutation from a list of a set of integers

```
   Input  : Code design-model (AM), graph of conflicts G_C, list of permutations (permuts)
   Output: A list of refactorings (bestPermutation)
 1 Procedure extractBestPermutation (AM, G_C, permuts):
 2     bestPermutScore = +∞
 3     bestPermutation = new List
 4     for row = 1 to row = permuts.size do
 5         permut = new List
 6         for col = 1 to col = permuts[1].size do
 7             if G_C.isTherePathBetweenNodes(permuts[row][col], permut)==false then
 8                 permut.add(permuts[row][col])
 9             end if
10         end for
11         permutScore = evaluateImpactOfPermutation(permut, AM)
12         if permutScore < bestPermutScore then
13             bestPermutation = permut
14             bestPermutScore = permutScore
15         end if
16     end for
17     return bestPermutation
18 end
```

Algorithm 2 starts by initializing $bestPermutScore$ to positive infinity (as we are performing minimization) and $bestPermutation$ to an empty list. The main *for-loop* (line 4), consists on

iteratively adding refactoring operations from the current permutation to *permut*. If the current refactoring is conflicted with the refactorings already added, it continues to the next operation until the end of the current permutation. Then, it evaluates the impact of the current permutation (*permutScore*), and if is this value is less than the current *bestPermutScore*, it replaces *bestPermut* and *bestPermutScore* with *permut* and *bestPermutScore*. Note that the application of each permutation of refactorings can result in one of the following outcomes: the permutation removes an anti-pattern in the source class; it does not remove the anti-pattern in the source class (*e.g.,* there are not enough move method refactorings to substantially decompose a Blob class); removes the anti-pattern in the source class and introduces an anti-pattern in the target class; or does not remove the anti-pattern in the source class, but adds a new anti-pattern in the target class. The permutation with the best score is returned (line 17).

Algorithm 3: Algorithm to evaluate a permutation in terms of the number of anti-pattern it can remove

```
   Input  : A sequence of integers (permut), code-design model (AM), a set of refactoring candidates (R), bipartite
            graph (G_B)
   Output: An integer value (score)
 1 Procedure evaluateImpactOfPermutation(permut, AM, R):
 2     score = 0
 3     for col = 1 to col = permut.size do
 4         r = R.getRefactoring(permut[col])
 5         Ap = Detect Antipatterns(adj(r, G_B))
 6         if AM.ApplyRefactoring(r)==true then
 7             Ap' = Detect Antipatterns(adj(r, G_B))
 8             score=score+(Ap'-Ap)
 9         else
10             permut.remove(r)
11         end if
12     end for
13     AM.rollbackSequence(permut,R, G_B)
14     return score
15 end
```

In Algorithm 3 we present the procedure to evaluate a permutation in terms of the number of anti-patterns that it corrects. The procedure starts by initializing the variable $score = 0$. In line 3, we have a *for loop* to iterate over all refactorings in the permutation. Next, in line 5, we proceed to detect anti-patterns in the vertices adjacents to r in the bipartite graph, *i.e.,* $adj(r, G_B)$. The outcome of the detection is stored in *Ap*. Next, if the application of r on the code-design model succeeds, we recompute the number of anti-patterns in the related classes, $adj(r, G_B)$ again, this time in the refactored design. Variable *score* is computed by subtracting the count of anti-patterns after refactoring (*i.e., Ap*′) from the count of anti-patterns before refactoring (*i.e., Ap*), and adding this value to the current *score*. If r cannot be applied to the model, we remove r from *permut* (line 10). This is done to reduce the overhead of scheduling invalid refactorings. One may think that validating the existence

of conflicts between r and the refactorings previously scheduled in Algorithm 2 should be enough to warrants a valid sequence. However, we cannot be totally sure until we apply the refactoring sequence on the software system. Applying the refactoring on the software system can be computationally expensive, specially for a search algorithm. As an alternative, we use a code-design model that enables us to simulate the application of a refactoring on the software system. At the end of the loop (line 12), we undo all the refactorings from *permut* that were applied to the code-design model, and return *score* (lines 13-14).

Algorithm 4: Algorithm to obtain the first valid sequence from a set of refactorings

```
   Input  : Code design-model (AM), graph of conflicts G_C, set of connected components (ccap), a set of refactoring
            candidates (R), bipartite graph (G_B)
   Output: A sequence of refactorings (sequence)
 1 Procedure getFirstValidSequenceFromccap (AM, G_C, ccap, R, G_B):
 2     desiredEffect=-1
 3     sequence=new list
 4     tempRefactoringSeq=new list
 5     for each element ∈ ccap do
 6         score=0
 7         r = R.getRefactoring(element)
 8         if G_C.isThereePathBetweenNodes(element, sequence)==true then
 9             continue
10         end if
11         Ap = Detect Antipatterns(adj(r, G_B))
12         if AM.ApplyRefactoring(r)==true then
13             Ap' = Detect Antipatterns(adj(r, G_B))
14             tempRefactoringSeq.add(element)
15             score=score+(Ap' − Ap)
16             if score <= desiredEffect then
17                 removeAntipattern = true
18                 exit for
19             end if
20         end if
21     end for
22     AM.rollback(tempRefactoringSeq)
23     if removeAntipattern = true then
24         sequence = tempRefactoringSeq
25     end if
26     return sequence
27 end
```

Algorithm 4 starts at line 2, when variable $desiredEffect$ is set to -1. This means that the application of the sequence built from a *ccap* removes one anti-pattern (in the source class) and do not add any anti-pattern in any related class. Next, a *for loop* (line 5) iterates the elements in *ccap*. If *element* is conflicted with any of the refactorings already scheduled in *sequence*, we skip to the next *element*. Otherwise, we perform anti-patterns detection on the vertices adjacents to r in G_B. The resulting value is stored in Ap. If the application of r succeeds, we retrieve the participating elements of r from the refactored code-design model, and detect anti-patterns again. Next, we add *element* to $tempRefactoringSeq$ and compute *score*, similar to Algorithm 3. If *score* is less or equal to $desiredEffect$, we set $removeAntipattern$ to *true* and exit the main loop. Finally, we rollback the applied refactor-

ings in the code-design model. If we succeeded in removing at least one anti-pattern instance, we set *sequence* equal to $tempRefactoringSeq$. Otherwise, an empty sequence is returned.

5.4 Case Study

In this section, we conduct a case study to assess the effectiveness of RePOR at improving the design quality of systems. The *quality focus* is the improvement of the design quality of a software system through refactoring. The *perspective* is that of researchers interested in developing automated refactoring tools for software systems, and practitioners interested in improving the design quality of their software systems. The *context* consists of the four meta-heuristics: Ant Colony Optimization (ACO), Genetic Algorithm (GA), LIU, and RePOR, and five open-source systems (OSS). We select Ant Colony Optimization, Genetic Algorithm, LIU to compare the results provided by RePOR as they are well-known techniques successfully used in previous studies for scheduling refactorings [25, 44, 82, 36, 145]. We choose the five OSS according to the following criteria (1): systems belonging to different application domains, (2) availability for replication, (3) use in previous studies concerning refactoring and anti-patterns [52, 36] and (4) non-trivial systems that are likely to present conflict when refactoring.

5.4.1 Research Questions

To better answer the central question of this chapter, we formulate the following research questions:

(RQ1) To what extent can RePOR remove anti-patterns?
This research question aims to assess the effectiveness of RePOR at improving design quality. We use the number of occurrences of anti-patterns as a proxy for design quality, as they have been found to hinder system evolution [8], and to be correlated with the occurrence of bugs [152]. Hence, the more anti-patterns removed the better.

(RQ2) How does the performance of RePOR compare to those of metaheuristics ACO, GA, and the conflict-aware approach LIU from the literature, for the correction of anti-patterns?
This research question aims to assess the performance of RePOR in terms of execution time and effort. The rational of studying the execution time is that developers are advise to perform refactoring regularly along with other coding activities [47]. Hence, the waiting time for an algorithm to produce refactoring solutions should be small to be suitable for working on the loop with developers. The rationale for studying refactoring effort is that

performing a long list of refactorings to achieve high-quality design improvement could lead to an unrecognizable design for developers. It also increases the probability to introduce regression, as it is not suitable to be reviewed by a human pair. Hence, we believe that from developers' perspective [28], it is important to minimize the number of necessary refactorings to obtain a reasonable quality improvement.

5.4.2 Evaluation Method

In the following, we describe the approach followed to answer **RQ1**, **RQ2**.
All statistics have been performed using the R statistical environment [1]. For all statistical tests, we consider a significance level of 5%. For **RQ1**, we measure the effectiveness of RePOR at removing anti-patterns in software systems using the following *dependent variable*:

— Design Improvement (DI). DI represents the delta of anti-patterns occurrences between the refactored system (SYS') and the original system (SYS) and it is computed using the following formulation.

$$DI(SYS) = \frac{|AC(SYS') - AC(SYS)|}{AC(SYS)} \times 100. \tag{5.2}$$

Where $AC(SYS)$ is the number of anti-patterns in a system SYS and $AC(SYS) \geq 0$. The value represents the improvement amount in percentage. High negative values are desired.

The independent variable is the refactoring approach applied to each studied system. We statistically compare the number of remaining anti-patterns after refactoring a system using RePOR with the number of remaining anti-patterns when using other refactoring approaches. Specifically, we test the following hypothesis H_{01}*: There is no difference between the number of remaining anti-patterns of a system refactored using RePOR, and a system refactored using other refactoring approaches.* We test the hypothesis using a non-parametric test, i.e., the Mann-Whitney U test [153]. For estimating the magnitude of the differences of means between the number of remaining anti-patterns in systems refactored by RePOR and systems refactored using other approaches, we use the non-parametric effect size measure Cliff's δ ES, which indicates the degree of overlap between two sample distributions [154]. ES values range from -1 (if all selected values in the first distribution are larger than the second distribution) to +1 (if all selected values in the first distribution are smaller than the second distribution). It is zero when two sample distributions are identical. Cliff's δ effect size is

1. http://www.r-project.org/

considered small when $0.147 \leq \text{ES} < 0.33$, medium for $0.33 \leq \text{ES} < 0.474$, and large for ES$\geq 0.474$ [155].

For **RQ2**, the dependent variables are the execution time and the effort:

— Execution Time (ET). ET represents the total CPU time for the algorithm thread in milliseconds. CPU time is the time that a process is actually running (not waiting on I/O or blocked by other threads that got CPU quantum). We use Oracle's *java.lang.management* library to measure this metric [2].

— Refactoring Effort (RE). We calculate the effort of refactoring by counting the number of refactorings that are scheduled to remove an anti-pattern.

The independent variable is the refactoring approach. We test the following two null hypothesis: H_{02} *: There is no difference between the execution time of RePOR and the execution time of the other studied refactoring approaches.* H_{03} *: There is no difference between the refactoring effort incurred by RePOR and the refactoring effort incurred by other studied refactoring approaches.* To test H_{02}, H_{03}, we use the same statistical tests as in **RQ1**.

Solution representation.

We use a vector representation where each element is a refactoring operation (r) to be applied (*cf.* Section 4.4).

Code Design-Model

The code design-model is generated using Ptidej tool suite [156].

Detection and correction of anti-patterns

To detect anti-patterns, we use DECOR (*cf.* Section 2.2.2). In this chapter, we consider five types of anti-patterns, namely Blob (BL), Lazy Class (LC), Long Parameter List (LP), Spaghetti Code (SC) and Speculative Generality (SG). These anti-patterns are well-recognized by developers [7], and have been studied in previous works [4, 52, 135, 72].

The refactoring strategies that we developed to correct the anti-patterns studied in this chapter were already introduced in Section 4.4, except for Blob Class. To correct Blob anti-pattern, we leverage PADL to determine the number of methods and attributes of a class, and compare it with the rest of the classes in the system (boxplox technique). Then, we estimate the cohesion between its methods and attributes, and determine the existence of

2. https://docs.oracle.com/javase/8/docs/api/java/lang/management/package-summary.html

controlling relationships with other classes. After performing these inter-class analysis, we can propose to move methods to redistribute the excess of functionality from Blob classes to related classes. We follow the strategy proposed by Seng et al. [44] that consists of searching for each method's signature, candidates classes inside the list of parameter types; in case that the parameter type is not primitive and the source code is not a library, we generate a move method refactoring from Blob class to the parameter type. We also consider as candidate classes to move methods, the field types of the defined in the Blob class.

Systems studied

In Table 5.3, we present information about the systems studied: number of classes (NOC), number of lines of code $\times 10^3$ (KLOC), and number of anti-patterns detected by type.

Table 5.3 Descriptive statistics about the studied systems.

System	NOC	KLOC	BL	LC	LP	SC	SG	Total
Apache Ant 1.8.2	697	191	57	40	35	3	6	141
ArgoUML 0.34	1754	183	131	25	281	1	19	457
GanttProject 1.10.2	188	44	47	4	68	5	6	130
JfreeChart 1.0.19	505	98	41	21	62	1	1	126
Xerces 2.7	540	71	56	25	119	2	3	205

In Table 5.4, we present the number of refactoring candidates that were automatically generated by RePOR.

Table 5.4 Number of refactoring candidates automatically generated for each studied system.

CH	IC	IPO	MM	RMWO	Total
Ant					
6	9	35	4269	3	4322
ArgoUML					
19	25	281	2475	1	2800
Gantt Project					
6	4	68	3861	5	3944
JfreeChart					
1	21	62	4228	1	4313
Xerces					
3	25	119	4118	2	4267

5.4.3 RePOR implementation

We instantiate RePOR as an Eclipse plug-in and compared it with three metaheuristics. Design improvement (DI) is measured using Equation (5.2). The parameter *threshold*, described in Section 5.3.7, is set using the following criterion: a value for which the number of

permutations can be stored in memory. To determine this value, we performed 30 independent executions for each of the systems studied in a Windows 10 64-bit, Intel Core 5 at 2.30 GHz, 12 GB of memory machine, and record the number of permutations that we can store for each *ccap*, and found that the maximum number of permutations that we can store in memory is $threshold = 10! = 3,628,800$.

The directed graph of conflicts (G_C) is used for the three metaheuristics to avoid scheduling invalid refactorings. Due to the random nature of the metaheuristics studied (*i.e.,* ACO and GA) it is necessary to perform several independent runs to have an idea of the behavior of the algorithms. Hence, we execute 30 independent runs for all the approaches studied and for each system. This is a typical minimum value (*i.e.,* 30 runs) used in the search-based research community to have enough experimental data to perform a statistical analysis.

With respect to the search of the connected components in the graph of dependencies between refactorings (G_D), we use the implementation proposed by Sedgewick and Wayne [157] which uses a recursive depth-first search algorithm.

The stopping criteria for the metaheuristics studied has to be uniform to provide a fair comparison. While in RePOR and LIU the stopping criteria is determined by the number of vertices in the refactoring dependency and conflict graphs, for ACO and GA, the number of evaluations (transformations applied to the randomly-generated initial solutions) required to find an optimal solution cannot be determined before hand. Typically, researchers use number of evaluations or execution time as stopping criteria. We use number of evaluations as the stopping criterion, with a maximum of one thousand evaluations (for each system). This value was empirically determined in our previous works [144, 145].

The next paragraphs disclose in detail the implementations of ACO, GA, and LIU used in this case study.

5.4.4 Ant Colony Optimization Implementation

Ant Colony Optimization (ACO) [148] is a constructive metaheuristic, inspired by the behavior of real ants, that has been successfully applied in solving NP-hard problems, i.e., problems that in theory cannot be solved in polynomial bounded computation time, such as routing (traveling salesman, vehicle routing), assignment (graph coloring, frequency assignment), scheduling (job shop, flow shop), network routing (connection-oriented network routing), etc. The benefits of using ACO are: rapid discovery of good solutions, distributed computation which avoid premature convergence like in local search, and greedy heuristics which helps to discover acceptable solutions in the early stages of the search process. In

our ACO implementation, the ants are artificial agents that cooperate to build a path in a directed graph $G = (S, T)$ where S is the set of nodes and $T \subseteq S \times S$ is the set of arcs. A finite path over the graph is a sequence of nodes (refactorings operations) $\pi = s_1, s_2, \ldots, s_n$ where $s_i \in S$ for $i = 1, 2, \ldots, n$. We denote π_i the ith node of the sequence and we use $|\pi|$ to refer to the length of the path, *i.e.,* the number of nodes of π.

Our ACO implementation corresponds to a simple ACO [148], where the best ant in the colony updates the pheromone matrix. In Algorithm 5 we describe the main steps of ACO implementation. The steps from line 2 to 5 are the same steps performed by RePOR, and the main algorithm starts in Line 8. In the algorithm, the path traversed by the ith artificial ant is denoted with a^i. We use $|a^i|$ to refer to the length of the path, the jth node of the path is denoted with a^i_j, and the last node with a^i_*. We denote with $T(s)$ to the set of successor nodes of node a^i_*. We use the + operator to indicate concatenation between paths. The maximum value for $|a^i|$ is the number of elements in R (Line 3) *i.e.,* λ_{ant}. The search process starts at line 8 where the pheromone trails are initialized with the same value: a random number between 0 and 1. After the initialization, the ants start the path construction from different nodes, and the algorithm is executed during a given number of steps m (line 10). Inside the loop, each ant builds a path randomly selecting the next node according to the pheromone (τ_{ij}) and the heuristic value (η_{ij}) associated to each arc (i, j) (Line 14). In fact, if the *kth* ant is in node i, it selects node j with probability

$p_{ij} = \frac{[\tau_{ij}]^\alpha [\eta_{ij}]^\beta}{\sum_{k \in N_i} [\tau_{ik}]^\alpha [\eta_{ik}]^\beta}$, where p_{ij} is the probability of an ant to move from node i, to node j. τ_{ij} is the trail intensity which provides information about how many ants have passed through this path. N_i is the set of successor nodes from node i. η_{ij} is an associated heuristic value. k is a mute variable whose domain is the set of successors nodes. The concrete expression is $\eta_{ij} = h(j)$, where $h(j)$ is the score assigned to the candidate refactoring operation by a *heuristic function.* The construction phase is iterated until the ant reaches the maximum length λ_{ant}, or the current node has no successors in the graph (Line 13).

Once an ant has built a path, it is necessary to evaluate it on-the-fly. We generate a clone of the original design (Line 17) and for each node in a^k we apply its corresponding refactoring operation. Then, the algorithm performs anti-patterns' detection (Line 21) in the resulting model. The design quality is evaluated according to the defined objective function. A good solution is a sequence that corrects more anti-patterns.

After the construction phase, the pheromone trails are updated (Line 28) to take into account the quality of the candidate solutions previously built by the ants. The pheromone update follows the expression: $\tau_{ij} \leftarrow \rho\tau_{ij} + f(a^{best}), \forall (i, j) \in a^{best}$, where ρ is the *pheromone evaporation rate* and it holds that $0 \leq \rho \leq 1$. On the other hand, $f(a^{best})$ is the amount of

pheromone that the best-ant-path, ever found, deposits on arc (i, j).

The algorithm is finalized whenever the algorithm reaches one of the following conditions:

1. We reach the maximum number of steps ($msteps$).
2. We reach the optimal state, *i.e.,* The number of classes with anti-patterns is zero ($NDC = 0$).

Algorithm 5: Ant Colony implementation for scheduling refactoring

```
   Input  : System to refactor (SYS)
   Output: An optimal sequence of refactoring operations (SR)
 1 Steps ACO(SYS)
 2     AM=code-design model generation (SYS)
 3     Ap = Detect Anti-patterns(AM)
 4     R = Generate set of refactoring candidates(AM, A)
 5     G_C = Build Graph of conflicts between refactorings and anti-patterns (AM, LR)
 6     SR = Ant Colony Optimization for refactoring(G_C, AM)
 7 Procedure Ant Colony Optimization for refactoring(G_C, AM):
 8     τ = initialize_pheromone()
 9     step = 1
10     while step ≤ msteps AND Ap ≠ 0 do
11         for k = 1 to colsize do
12             a^k = null
13             while |a^k| ≤ λ_ant AND T(a*^k) − a^k ≠ ∅ do
14                 node = select_successor(G_C, T(a*^k), τ, η)
15                 a^k = a^k + node
16             end while
17             AM' = AM.clone()
18             for all node ∈ a^k do
19                 apply_refactorings(AM', node)
20             end for
21             a^k.Ap = detect_antipatterns(AM')
22             if DI(a^k) < DI(a^best) then
23                 a^best = a^k
24                 Ap = a^best.Ap
25             end if
26         end for
27         τ = pheromone_evaporation(τ, ρ)
28         τ = pheromone_update(τ, a^best)
29         step = step + 1
30     end while
31     return a^best
32 end
```

ACO heuristic function

The heuristic value (η_{ij}) is calculated by a function that produces an integer value that defines *how beneficial* is to apply a refactoring r to a class in the system. According to the number of coexisting anti-patterns in the source class, we assign a score that increases with the benefits of applying r on each of the detected anti-patterns in a class. To determine the score, we assign for each refactoring type, an integer value in the range of -2 to 2, where -2 represents a negative effect for a particular anti-pattern, and 2 a very desirable effect,

i.e., complete correction. Let us take the following example: suppose that class A has two coexisting anti-patterns namely LC and LP. The suggested refactorings for correcting those anti-patterns are inline class and introduce parameter-object, respectively. Suppose that we want to evaluate the *goodness* of *node* 1, inline class. For the first defect (LC) we give a score of **2**, as it is the ideal refactoring for correcting LC, and **0** (no benefit or detriment) to LP; then the total score for *node 1* will be 2 (2+0) as well as for *node 2*. On the contrary, suppose that class A has two defects (SC and LP), and we want to prioritize the refactoring of SC over LP. Then, we could assign a heuristic value of 2 to RO type replace method with method object, when a class has SC, and 1 to introduce parameter-object, when a class has LP. In this way the sum of scores for this example will be (2+0), and (1+0) respectively, having more probability to choose the node that corrects SC over the one that corrects LP. The heuristic component η_{ik} cannot accept values equal to zero. Thus, we compute 2^{score} to provide a value in the domain of natural numbers.

In Table 5.5 we show the parameters used for ACO. These parameters are not set in an arbitrary way, but they are the result of running ACO with different configurations 30 times, in a *factorial design*. For example, to select the importance of the heuristic in ACO, we tried the following couples: no heuristic ($\alpha = 1$, $\beta = 0$), same importance ($\alpha = 1$, $\beta = 1$), more importance to pheromone ($\alpha = 2$, $\beta = 1$) and so on.

Table 5.5 Parameters of the Ant Colony Optmization algorithm for refactoring scheduling.

Ant Colony Optimization			
Parameter	**Value**	**Parameter**	**Value**
$msteps$	10	ρ	0.8
$colsize$	100	β	2.0
λ_{ant}	$\|R\|$	α	1.0

5.4.5 Genetic Algorithm implementation

The GA used in this dissertation is a generational genetic algorithm (gGA). In gGA, half of the population is selected and crossed; next, the resulting offspring is mutated and inserted into the population replacing the old individuals.

In Algorithm 6, we describe the main steps of our GA implementation. We define P as a list of refactoring sequences $s \in P$. Lines from 1 to 6 are the initialization steps and the main algorithm starts in line 7. The population size for the experiments is 100 individuals. In Line 8, the population is initialized with randomly generated refactoring sequences, and evaluated in Line 10. In line 15, the refactoring sequences are sorted in descending order by their fitness (number of anti-patterns corrected). The main loop starts in Line 16 until the

stopping criterion is met. For this case study, we use number of evaluations.

Algorithm 6: Genetic Algorithm implementation for scheduling refactorings

```
   Input  : System to refactor (SYS)
   Output: An optimal sequence of refactoring operations (SR)
 1 Steps GA(SYS)
 2     AM=code-design model generation (SYS)
 3     A = Detect Anti-patterns(AM)
 4     R = Generate set of refactoring candidates(AM, A)
 5     G_C = Build Graph of conflicts between refactorings and anti-patterns (AM, LR)
 6     SR = Genetic Algorithm for refactoring(G_C, AM)
 7 Procedure Genetic Algorithm for refactoring(G_C, AM):
 8     nPop = populationSize
 9     P = GenerateInitialPopulation(AM, G_C)
       /* Evaluation of P                                                    */
10     for all s ∈ P do
11         AM' = AM.clone()
12         apply_refactorings(AM', s)
13         s.Ap = detect_antipatterns(AM')
14     end for
       /* the sequences are sorted in ascendent order according to Ap        */
15     P.sort()
16     while not StoppingCriterion do
           /* add the best two individuals of the previous population in O population */
17         O.add(P_0)
18         O.add(P_1)
           /* Reproductive cycle                                             */
19         for 0 to nPop/2 − 1 do
               /* parents is a list of refactoring sequences                 */
20             parents = new List of size 2
21             parents_0 = selection_operator(P)
22             parents_1 = selection_operator(P)
23             offspring = Variation_Operators(parents, G_C)
               /* We generate two offsprings                                 */
24             AM' = AM.clone()
25             apply_refactorings(AM', offspring_0])
26             offspring_0.Ap = detect_antipatterns(AM')
27             AM' = AM.clone()
28             apply_refactorings(AM', offspring_1)
29             offspring_1.Ap = detect_antipatterns(AM')
30             O.add(offspring)
31         end for
32         P=O
33         O=null
34         P.sort()
35     end while
36     best_solution = P_0
37     return best_solution
38 end
```

Parameters of GA

We use the same parameters than those used in Section 4.4.3.

5.4.6 LIU conflict-aware scheduling of refactorings

Liu et al. [82, 83] proposed different heuristics to solve the refactoring scheduling problem. From these approaches, we select the former one [82], as it is the one that could work with

the anti-patterns studied in this chapter. On the other hand, the approach proposed in [83] assumes that the refactoring of certain type of anti-patterns can lead to the resolution of another types (*e.g.,* removing code duplications can affect long method). Hence, they leverage this property to remove redundant edges in the graph of conflicts using topological order. However, the type of anti-patterns that we studied and their corresponding refactorings are independent (*e.g.,* it is not appropriate to apply inline class refactoring from a data class to a Blob class; or collapse hierarchy and inline class cannot be applied at the same time to the same class).

In the following paragraphs we explain the steps that we took to adapt the *conflict-aware scheduling of refactorings* [82] (LIU for short) to our framework, to compare it with RePOR.

LIU uses the QMOOD [40] to assess the effect of applying a refactoring on a software system. Because QMOOD combines weighted design metrics (*e.g.,* design size, hierarchies, polymorphism, etc.) to measure quality attributes like reusability, understandability, flexibility, etc. The values obtained for each quality attribute are only useful when compared to the values obtained from systems of the same domain used by the industry. Hence, in the evaluation of LIU [82] they refactored an in-house-developed-modeling tool, and to calibrate the weights of design metrics, they take as an upper-bound the metrics values obtained from a similar open-source system (*BPEL* from Eclipse foundation). However, in this dissertation we use the occurrence of anti-patterns as proxy for design quality. We believe that the occurrences of anti-patterns is a more appropriate way to asses the quality of a software system, as it does not require to find a good-quality representative system to compare with. Our anti-pattern detection framework relies on DECOR which uses relative threshold values to asses the quality of each class in the system, which makes it more flexible and easier to adapt for an automated approach as it does not require a calibration step.

The steps of our implementation of LIU are summarized in Algorithm 7. The algorithm starts after generating the list of refactoring candidates and building the graph of conflicts (Lines 2-5). In line 9 we start applying all uninjurious refactorings, *i.e.,* refactorings that do not prevent the application of other refactorings. More formally, i is an uninjurious refactoring iff there is not an edge e from v_i to v_j where $\{v_i, v_j \in E\}$ $, E \in G_C$

If there are no more refactorings left in GC, the algorithm ends (Line 13). Otherwise, we iterate over all injurious refactorings and perform the following steps.

Compute synthetical effect. It consists of computing the effect of applying a refactoring i in the system, *i.e.,* the increment/decrement of anti-patterns occurrences after applying i. We denoted the synthetical effect of applying refactoring i as $SynQ_i$.

Compute potential effect. The application of a refactoring may disable other refactorings

Algorithm 7: LIU conflict-aware scheduling of refactorings

```
   Input  : System to refactor (SYS)
   Output: An optimal sequence of refactoring operations (SR)
 1 Steps LIU(SYS)
 2     AM=code-design model generation (SYS)
 3     A = Detect Anti-patterns(AM)
 4     R = Generate set of refactoring candidates(AM, A)
 5     G_C = Build Graph of conflicts between refactorings(AM, LR)
       /* G_C = (V, E)                                                  */
 6     SR = Find sequence of refactorings(G_C, AM)
 7 Procedure Find sequence of refactorings(G_C, AM):
 8     SR = ∅
       /* first applying all uninjurious refactorings                   */
 9     for each v_i|adj(v_i) = 0 do
10         Remove v_i and its edges from G_C
11         SR.add(v_i)
12     end for
13     if |G_C| == 0 then
14         return SR
           /* End algorithm                                             */
15     end if
       /* first applying all injurious refactorings                     */
16     for each v_i|adj(v_i) ≠ 0 do
17         Compute synthetical effect (SynQ_i)
18         Compute potential effect (PQ_i)
19         Selection and application
20         Update potential effect
21     end for
22     return SR
23 end
```

(negative effect), or reduce the possibility of conflicts (positive effect) for those refactorings that are adjacent to v_i. Note that for LIU, there is an edge (asymmetrical conflict) between v, u iff u can be applied before, but not after v. In our motivating example, r_2 presents an asymmetrical conflict with r_1 according to LIU. We denoted the potential effect of applying refactoring i as PQ_i.

Selection and application. Select a vertex v_i from G_C that has the greatest potential effect (PQ) and add it to SR.

Update potential effect. Once refactoring i is applied, we remove the vertex from G_C and update the potential effect of vertices adjacents to v_i ($adj(v_i)$).

5.5 Results

In this section, we answer our two research questions that aim to evaluate RePOR.

5.5.1 (RQ1) To what extent can RePOR remove anti-patterns?

We present in Table 5.6 the Design improvement (DI) in general and for different anti-pattern types, for each studied system. The results are the median of the 30 independent executions.

Table 5.6 Design Improvement (%) in general and for different anti-pattern types.

Metaheuristic	DI	DI_{BL}	DI_{LC}	DI_{LP}	DI_{SC}	DI_{SG}
Ant						
ACO	57.45	68.42	22.5	74.29	66.67	100
GA	58.16	68.42	22.5	74.29	66.67	100
LIU	58.87	54.39	22.5	100	66.67	100
RePOR	60.28	57.89	22.5	100	66.67	100
ArgoUML						
ACO	75.93	51.15	100	83.63	100	100
GA	76.59	51.15	100	84.7	100	100
LIU	81.40	50.38	100	92.88	100	100
RePOR	81.62	38.93	100	98.58	100	100
Gantt Project						
ACO	60	17.02	100	83.82	70	100
GA	60.77	14.89	100	85.29	80	100
LIU	63.85	14.89	100	92.65	60	100
RePOR	66.15	8.51	75	100	100	100
JfreeChart						
ACO	75.4	39.02	100	89.52	100	100
GA	75.4	39.02	100	90.32	100	100
LIU	72.22	31.71	100	88.71	100	100
RePOR	75.4	24.39	100	100	100	100
Xerces						
ACO	56.59	14.29	100	65.55	100	100
GA	57.56	14.29	100	67.23	100	100
LIU	64.39	16.07	100	78.99	50	100
RePOR	73.17	5.36	100	98.32	100	100

With a median DI of 73%, overall, the design improvement (first column) of the solutions generated by RePOR is higher in comparison with the improvements achieved by the other three refactoring approaches. The DI of LIU is close to the one obtained by RePOR except in one system, JfreeChart, where LIU achieved the lowest DI. Concerning ACO and GA, the DI achieved is very similar.

With respect to the type of anti-patterns, RePOR have some difficulty to remove Blob anti-patterns compared to the other metaheuristics (we discuss further in Section 5.6), with one exception, ArgoUML, where it improves more than LIU. For Lazy Class, the results achieved are the same except for Gantt, where it removes less instances than the others. For Long Parameter List, RePOR reports the best results in all the systems studied. For Spaghetti code, RePOR overcomes the rest of the approaches in Gantt, and obtains equivalent results for the rest of the systems studied. Finally, for Speculative Generality the improvement obtained for all the algorithms is the same.

Table 5.7 presents the Mann-Whitney test results and Cliff's δ (ES) effect size obtained when comparing the number of remaining anti-patterns of the systems after being refactored by RePOR and the other refactoring approaches. We observe that all the differences are statistically significant with a large effect size, except for JFreeChart where the difference between ACO and RePOR is small, and the pair GA-RePOR where the effect size is negli-

gible. Therefore we reject H_{01} for the rest of the systems.

Table 5.7 Pair-wise Mann-Whitney U Test test for design improvement.

Pair	$p-value$	Cliff's δ ES	Magnitude
Ant			
ACO-RePOR	2.561349e-12	1	Large
GA-RePOR	1.431438e-11	1	Large
LIU-RePOR	1.685298e-14	1	Large
ArgoUML			
ACO-RePOR	1.176641e-12	1	Large
GA-RePOR	1.143381e-12	1	Large
LIU-RePOR	1.685298e-14	1	Large
Gantt Project			
ACO-RePOR	1.036681e-12	1	Large
GA-RePOR	1.086586e-12	1	Large
LIU-RePOR	1.685298e-14	1	Large
JfreeChart			
ACO-RePOR	0.06868602	0.2333333	Small
GA-RePOR	0.2771456	-0.1333333	Negligible
LIU-RePOR	1.685298e-14	1	Large
Xerces			
ACO-RePOR	1.0618e-12	1	Large
GA-RePOR	9.946555e-13	1	Large
LIU-RePOR	1.685298e-14	1	Large

We reject the null hypothesis H_{01} for Ant, ArgoUML, Gantt, JfreeChart, and Xerces. In these five systems, the number of remaining anti-patterns after refactoring using RePOR is significantly lower than the number of anti-patterns remaining in the systems after refactoring using the other refactoring approaches (i.e., ACO, GA, and LIU). With respect to the magnitude of Cliff's δ, the difference is large for all the systems, except the pairs ACO-RePOR and GA-RePOR in JFreeChart, where it is small and negligible, respectively. Overall, our results suggest that for the set of anti-patterns studied and the systems analyzed, RePOR can correct more anti-patterns, than ACO, GA, and LIU.

5.5.2 (RQ2) How does the performance of RePOR compare to those of meta-heuristics ACO, GA, and the conflict-aware approach LIU from the literature, for the correction of anti-patterns?

We present in Table 5.8 the execution time (ET) and the refactoring effort (EF) incurred for each refactoring scheme. ET is given in seconds, while EF represents the number of refactorings applied. The results are the median of 30 independent runs.

We can observe that RePOR performs better than the other algorithms in terms of execution time and effort, with a remarkable difference, while removing more anti-patterns and using

Table 5.8 Median performance metrics for each system, metaheuristic studied.

Metaheuristic	Execution Time (ET)	Refactoring Effort (EF)
Ant		
ACO	11505.73	1686
GA	11558.97	1676
LIU	260.45	1641
RePOR	82.05	827
ArgoUML		
ACO	5617.51	1119
GA	5664.39	1123
LIU	148.45	1166
RePOR	72.88	438
Gantt Project		
ACO	5924.93	1069
GA	5975.71	1067
LIU	652.45	894
RePOR	133.45	119
JfreeChart		
ACO	11321.81	1748
GA	11369.82	1748
LIU	877.74	1747
RePOR	133.30	297
Xerces		
ACO	5781.67	886
GA	5831.93	887
LIU	389.43	909
RePOR	63.07	178

less resources. In terms of execution time; it takes between one minute and less than three minutes to generate a sequence for a complete system, while the second best scheme (LIU) takes a median of six and half minutes. This is equivalent to a median reduction of 80% of execution compared to the baseline approach.

With respect to refactoring effort, the number of refactorings scheduled are considerably less than the other approaches. The median refactoring's effort reduction is 80% less time than the second best scheme (LIU). We suggest that the less an automated approach deviates from the initial design of a system, the more chances to be accepted by developers and maintainers. Our rational derives from the following conjecture: developers are reluctant to accept large code transformations in favor of potential design quality improvement.

The performance of GA and ACO is poor compared to RePOR, despite using the same solution representation and the conflict graph (to discard invalid refactorings). We attribute this poor performance to their incapability to discard equivalent sequences (*i.e.,* permutations of refactorings that lead to the same design). Despite the fact that LIU has integrated a mechanism to evaluate the potential effect of applying/removing a refactoring from a sequence, it cannot avoid scheduling uninjurious refactorings that do not improve the design quality, incurring additional costs in effort and time.

Concerning RePOR, the overhead occurs when generating a refactoring sequence from a

permutation, in case that it contains a large number of elements. To deal with this issue, RePOR only a consider subset of refactoring operations from the permutation until it reaches the *desiredimpact*, *i.e.,* the correction of an anti-pattern instance without introducing a new one (*cf.* Algorithm 4). However, we do not expect to find many cases where the number of elements in a connected component is to large to be exhaustively explored. In Table 5.9 we provide some statistics about the size of the connected components in G_D generated by RePOR from the studied systems.

Table 5.9 Statistics of the connected components ($CCAP$) in G_D from the studied systems

System	Median size	Size>1	Total $CCAP$
Ant	1	46	99
ArgoUML	1	46	424
Gantt Project	1	25	108
Jfreechart	1	30	106
Xerces	1	36	173

We can observe that the median size of the connected components is one, and the number of connected components with size greater than one goes from 11% to 47% of total number of connected components in the worse scenario.

ACO and GA are algorithms for which it is not possible to predict when an optimal solution will be found. In general, the performance of a metaheuristic can be affected by the correct selection of its parameters. The configurable settings of the search-based techniques used in this chapter correspond to stopping criterion, population size, and the probability of the variation operators. We use the number of evaluations as the stopping criteria. As the maximum number of evaluations increase, we expect the algorithm to obtain better quality results. The increase in quality is usually very fast when the maximum number of evaluation is low. That is, the slope of the curve quality versus maximum number of evaluations is high at the very beginning of the search. But this slope tends to decrease as the search progresses. Our criterion to decide on the maximum number of evaluations is to select a value for which this slope is low enough. In our case *low enough* is when we observe that no more anti-patterns are removed after n number of evaluations, where n is the value that we are testing. We empirically tried different values in the range of 100 to 1500 and found 1000 to be the best value. However, that does not imply that the best solution is to be found at the end of the 1000 iterations, but could happen before. In addition, computing the average of design improvement with respect to time could help to determine if the evolution trend of the solutions could reach its inflexion point, or the algorithm was stopped prematurely.

To study the evolution of the quality of the solutions obtained by each algorithm every time the current best solution is improved, we compute the average quality of each solution with

respect to time, and present the results in Figure 5.1. The quality is expressed as DI, and the time is normalized using the min-max normalization, that is the minimum time value is mapped to 0 and the maximum value to 1. Given that RePOR and LIU produce only one solution in the entire process (instead of producing several solutions and evolving them), there is only one point for these approaches. The interpretation for a point p (t,v) is: from t and until the next sample, the average quality of the metaheuristic is v, where t represents time and v is the DI. We can observe that RePOR produces high-quality solutions in a small fraction of time, in comparison to the other approaches. There are only two cases where differences are small: in ArgoUML, LIU is very close to the results achieved by RePOR in terms of quality with a difference of 0.2%, and incurring only 1.75% additional time, while the difference with the best solutions of ACO and GA is not less than 5%. In JfreeChart, where GA approaches the best solution found by RePOR with a difference of 0.02% in DI, but with a remarkable difference of 99.70% of additional time. This is the only case where GA and ACO are clearly better than LIU. For the rest of the systems, as it is shown in Figure 5.1, both metaheuristics reached their inflexion point far below the optimal solutions found by RePOR and LIU.

With respect to the type of refactorings applied, we present in Table 5.10 the number of refactorings applied by type. We can observe that the number of refactorings applied by RePOR are almost the same, except for move method. That explains the reduction in effort required by RePOR compared to the other metaheuristics. It also explains why the results obtained for the removal of Blob are not so good, since for this type of anti-pattern requires the application of many refactorings to be corrected. Still, this should not be considered as a flaw of our approach, since the main objective is to correct the largest number of anti-patterns without prioritizing the correction of a particular type of anti-pattern, over the others anti-patterns. In this regard, RePOR succeeds well in improving the design quality of the systems studied, in a reasonable amount of time.

Finally, to assess the statistical significance of the results obtained, we compare performance metrics between RePOR and each metaheuristic using the same procedure as **RQ1**. Table 5.11 presents the pair-wise statistical tests for each metaheuristic. We observe that all the differences are statistically significant with a large effect size. Therefore we reject H_{02} for the five studied systems.

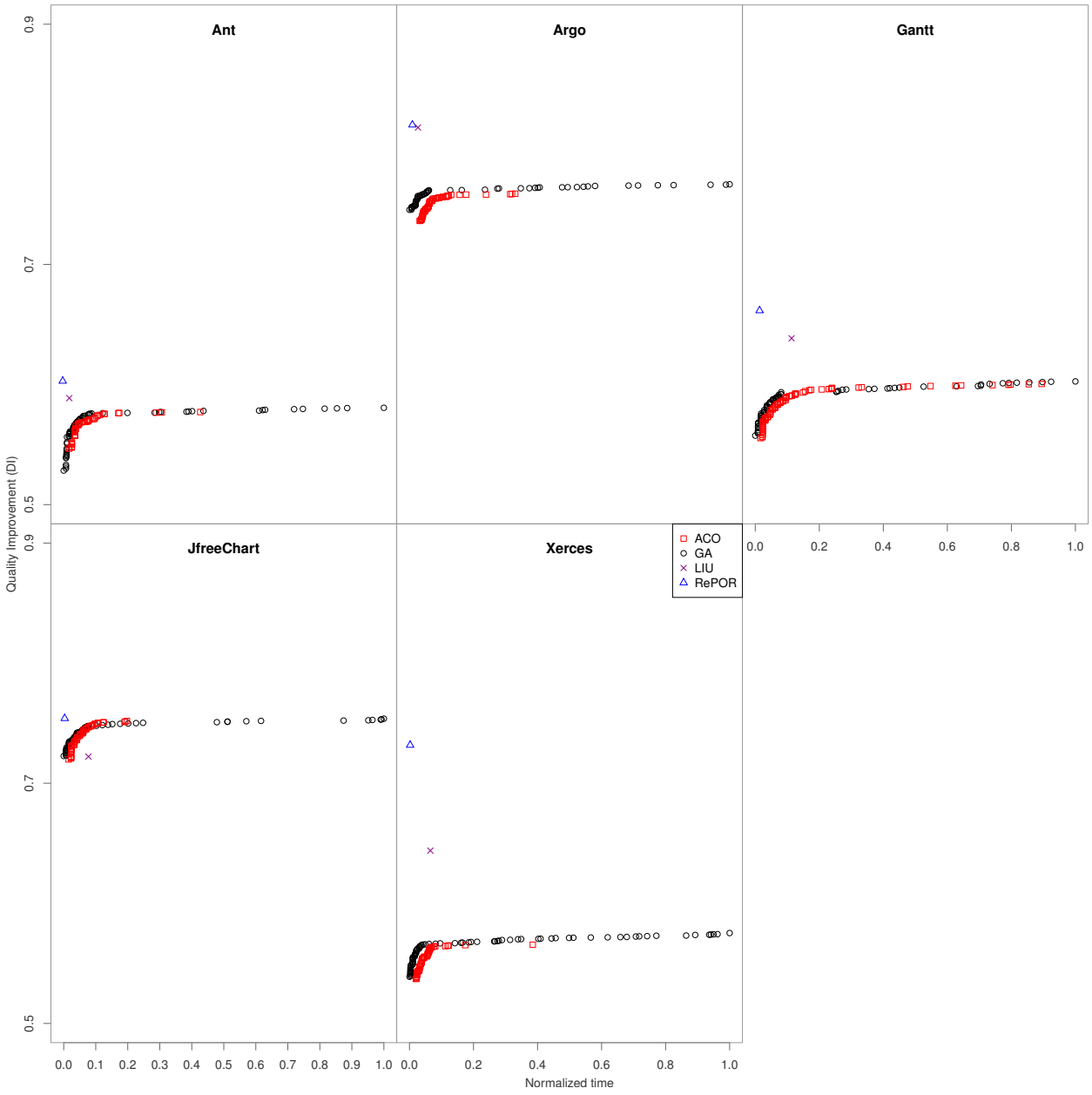

Figure 5.1 Quality evolution of the refactoring solutions with respect to time.

Table 5.10 Median count of refactorings applied for each system, refactoring scheme, by type.

Metaheuristic	CH	IC	IPO	MM	RMWO
Ant					
ACO	6	9	256	1643	3
GA	6	9	27	1629	3
LIU	6	9	35	1589	2
RePOR	6	9	35	774	3
ArgoUML					
ACO	17	24	246	829.5	1
GA	18	23	249	828.5	1
LIU	18	23	281	843	1
RePOR	17	25	280	115	1
Gantt Project					
ACO	6	4	59	996	3
GA	6	4	60	994	3
LIU	6	4	68	812	4
RePOR	6	4	68	37	5
JfreeChart					
ACO	1	21	56	1669	1
GA	1	21	56	1669	1
LIU	1	21	62	1662	1
RePOR	1	21	62	212	1
Xerces					
ACO	3	25	97.5	758.5	2
GA	3	25	99	759	1.5
LIU	3	25	119	761	1
RePOR	3	25	119	29	2

We reject the null hypothesis H_{02} and H_{03}, for Ant, ArgoUML, Gantt, JfreeChart, and Xerces. In these five systems, the execution time and the effort incurred by RePOR are significantly lower than those incurred by the other refactoring approaches. With respect to the magnitude of Cliff's δ, the difference is large for all the systems analyzed. Overall, our results suggest that for the set of anti-patterns studied and the systems analyzed, RePOR can correct more anti-patterns, using less time, and requiring less effort (in terms of refactorings applied) than ACO, GA, and LIU.

5.6 Discussion

In this section we discuss the results obtained by RePOR and their relevance for software maintainers and toolsmiths interested in improving the design quality of a software system through refactoring.

In Section 5.5 we have shown that RePOR is able to correct more anti-patterns using considerably less resources in terms of time and effort than state-of-art refactoring approaches. However, we observed that the number of instances of Blob anti-pattern removed by RePOR was lower than the number of Blobs removed by the other approaches. This could be explained by the large amount of refactorings that are required to remove a Blob anti-pattern,

Table 5.11 Pair-wise Mann-Whitney U Test test for performance metrics.

Metric	Pair	$p - value$	Cliff's δ ES	Magnitude
Ant				
ET	ACO-RePOR	1.691123e-17	1	Large
EF	ACO-RePOR	1.133109e-12	1	Large
ET	GA-RePOR	1.691123e-17	1	Large
EF	GA-RePOR	1.197023e-12	1	Large
ET	LIU-RePOR	1.691123e-17	1	Large
EF	LIU-RePOR	1.685298e-14	1	Large
ArgoUML				
ET	ACO-RePOR	1.691123e-17	1	Large
EF	ACO-RePOR	1.191166e-12	1	Large
ET	GA-RePOR	1.691123e-17	1	Large
EF	GA-RePOR	1.202906e-12	1	Large
ET	LIU-RePOR	1.691123e-17	1	Large
EF	LIU-RePOR	1.685298e-14	1	Large
Gantt Project				
ET	ACO-RePOR	1.691123e-17	1	Large
EF	ACO-RePOR	9.750474e-13	1	Large
ET	GA-RePOR	3.017967e-11	1	Large
EF	GA-RePOR	1.13497e-12	1	Large
ET	LIU-RePOR	1.691123e-17	1	Large
EF	LIU-RePOR	1.685298e-14	1	Large
JfreeChart				
ET	ACO-RePOR	1.691123e-17	1	Large
EF	ACO-RePOR	1.038395e-12	1	Large
ET	GA-RePOR	1.691123e-17	1	Large
EF	GA-RePOR	1.124768e-12	1	Large
ET	LIU-RePOR	1.691123e-17	1	Large
EF	LIU-RePOR	1.685298e-14	1	Large
Xerces				
ET	ACO-RePOR	1.691123e-17	1	Large
EF	ACO-RePOR	1.144319e-12	1	Large
ET	GA-RePOR	1.691123e-17	1	Large
EF	GA-RePOR	1.175678e-12	1	Large
ET	LIU-RePOR	1.691123e-17	1	Large
EF	LIU-RePOR	1.685298e-14	1	Large

in comparison to other types of anti-patterns. Another interesting observation is the fact that Long-parameter List and Lazy class anti-patterns show higher improvement with RePOR. Therefore, there seems to be a trade off between the refactorings that can be scheduled, as it is not possible to improve all types of anti-patterns to the same extent. What we present in this paper is an alternative refactoring approach, which proves to be more efficient than existing refactoring approaches in terms of design improvement, execution time, and effort. We achieved this result by clustering refactorings by the class that they affect in a connected component subgraph (*ccap*), and exhaustively searching (when possible) the best order for the refactorings for each *ccap*, as they are likely to lead to a different software design. In addition, as each *ccap* may contain conflicted refactorings that cannot be scheduled simultaneously, these refactoring operations are removed from the search space too, reducing the length of the sequences to be evaluated. Finally, for the set of refactorings in a *ccap* where the size is too large to explore all permutations exhaustively, we implement in our approach a mechanism to stop the addition of refactorings if we found that the desired effect (*i.e.,* the desired improvement in quality) is achieved, or just simply when the permutation does not lead to any improvement (*i.e.,* does not correct any anti-pattern). In comparison, LIU approach runs until there is no more refactorings left in the graph, so it assumes that all the refactorings that are not conflicted have to be scheduled. An assumption that may lead to the inclusion of unnecessary refactorings in the final sequence. With respect to ACO and GA, they start with random initial solutions that are iteratively transformed until the stopping criteria is achieved. While this proved to be useful for removing Blob anti-patterns, the usage of resources in terms of time and effort seems to be prohibitive for a coding session or when working interactively with a developer, and may be more suitable for refactoring sessions running after-hours as a batch process. Another disadvantage of ACO and GA is that they have to be calibrated in order to perform reasonably well, with the plethora of parameters involved for each algorithm as we show in Section 5.4. One final remark, the refactoring sequences generated by all the approaches studied in this chapter, do not prioritize any code entities that a developer might be interested as we did in Chapter 4. It is possible that developers are interested in refactoring certain specific packages or classes for which they have the ownership; or simply that they just prefer avoiding touching legacy code or critical components. To provide developers with a tool that could be used during daily coding tasks, we integrated RePOR as an Eclipse plug-in [158]. After analyzing a software system (or a subset of classes), our plug-in presents information about the anti-patterns detected, and generates a refactoring sequence from which developers can select the refactorings that they consider appropriate.

5.7 Threats to validity

We now discuss the threats to validity of our study following common guidelines for empirical studies [143].

Construct validity threats concern the relation between theory and observation. Our case study assumes that each anti-pattern is of equal importance, when in reality, this may not be the case. Concerning the scheduling of refactorings, we assume that the potential refactoring operations that can be applied in a software system are determined before the refactoring process begins. This is a big assumption, as new refactoring operations might be found as a consequence of changes in the code, *e.g.,* the application of previous refactorings. However, the search for new refactoring opportunities after applying each refactoring in a sequence is a costly operation. Therefore, most (if not all) the works on automatic refactoring assume that there is a list of refactoring opportunities at the beginning of the search and the optimization algorithm simply selects which of them will be applied and their order until the end of the list/starting of a new refactoring session [55].

Threats to internal validity concern our selection of subject systems, tools, and analysis method. With respect to anti-pattern's detection, DECOR is known to be accurate [52], it is not possible to guarantee that we detect all anti-patterns or that what we detect as anti-patterns are indeed true anti-pattern instances. Other anti-pattern detection techniques and tools should be used to confirm our findings.

Conclusion validity threats concern the relation between the treatment and the outcome. We paid attention not to violate assumptions of the constructed statistical models. In particular, we used non-parametric tests that do not require any assumption on the underlying probability distribution of data.

Reliability validity threats concern the possibility of replicating this study. Every result obtained through empirical studies is threatened by potential bias from data sets [159]. To mitigate these threats we tested our hypotheses over five open-source systems with different size, purpose and years of development. In addition to this, we attempt to provide all the necessary details required to replicate our study. The source code repositories of Apache Ant, ArgoUML, JfreeChart, Gantt and Xerces are publicly available, and have been studied in previous studies related to anti-patterns and code smells. In addition, we made the tool and the data generated publicly-available through our on-line replication web site [158].

Threats to external validity concern the possibility to generalize our results. Our study is focused on five open source software systems having different sizes and belonging to different domains. Nevertheless, further validation on a larger set of software systems is desirable,

considering systems from different domains, as well as several systems from the same domain. In this study, we used a particular yet representative subset of anti-patterns as proxy for software design quality. Future works using different type of anti-patterns are desirable.

5.8 Chapter Summary

In this chapter, we discussed the importance of efficiently scheduling refactorings operations, to reduce refactoring effort. We highlighted some of the major problems of existing search-based approaches. For example, the slow convergence of solutions from EAs, which means that they require too many evaluations to find a solution, that translates in longer execution times. We set out to address the following question:

Central Question: *Is it possible to improve automated refactoring by proposing an efficient scheduling?*

To answer this question, we proposed RePOR, a novel approach for automatically scheduling refactoring operations for correcting anti-patterns in software systems. To evaluate RePOR, we conducted a case study with five open-source software systems and compared the performance of RePOR with the performance of two well-known metaheuristics (GA and ACO) and one conflicting-aware refactoring approach (LIU). Results showed that RePOR can correct more anti-patterns than the aforementioned techniques in just a fraction of the time, and with less effort. Moreover, we integrate RePOR as an Eclipse plug-in to support developers with a fully automated-tool during their development work.

CHAPTER 6 USING TESTING EFFORT FOR IMPROVING AUTOMATED REFACTORING

6.1 Introduction

In previous chapters, we formulated the problem of refactoring as a single objective search problem, using the occurrences of anti-patterns to guide the search process. However, one may be interested in adding new objective functions to improve different design quality aspects at the same time. For example, one could be interested in minimizing the dependency between classes, while preventing the excessive accumulation of responsibilities in a single class. To combine more than one objective, (*cf.* Section 3.1.2) the SBSE research community has formulated the problem of refactoring anti-patterns as a multiobjective combinatorial optimization problem [45, 69, 24, 26], and used EMO to solve it. They have interspersed the correction of anti-patterns with other relevant objectives, like the use of developer's history, the preservation of domain semantics between classes and methods, etc. However, they have ignored another important aspect of the software development process, which is the testing effort.

Testing is an activity that aims to ensure that a system behaves according to its design specifications on a finite, but representative, set of test cases, taken from the infinite execution domain [160]. Researchers have investigated different ways to reduce testing effort and increase its effectiveness at different levels, *e.g.,* unit testing [96], integration testing [161], etc.; as well as for different software artifacts like documentation [162], and source code [96]. Refactoring operations are among the factors that can impact the testing effort of a system. In fact, *move method* or *extract class* refactorings applied to redistribute the responsibilities of a large class, either to its collaborators or to new entities, can allow units of code to be tested separately; reducing the number of scenarios to test. Moreover, writing test cases for the refactored class is simplified as its related components can be easily replaced with mock objects during testing.

However, to the best of our knowledge, despite its importance, testing has been mostly overlooked so far, during automated refactoring. We hypothesize that if we consider the reduction of testing effort (as an additional objective) during automated refactoring, we can obtain refactoring solutions that not only improve the design quality of the system, in terms of anti-patterns correction, but also reduce the testing effort at the same time.

To test our hypothesis, we introduce Testing-Aware ReFactoring approach (TARF), a novel

Multiobjective optimization (MO) approach for the problem of refactoring that minimizes the testing effort while improving the design quality. We perform a case study to assess the effectiveness of TARF using four different metaheuristics (one single and three multiobjective) and a benchmark of four open-source systems.

6.2 Improving automated refactoring of anti-patterns by leveraging testing effort estimation

In this chapter, we aim to improve the refactorings solutions found by an automated refactoring approach by leveraging testing effort estimation in a way that we can both improve design quality, and reduce testing effort of the software systems refactored.

Thus far, automated refactoring approaches have ignored the impact of their solutions on testing effort, which may contribute to developer's reluctance to adopt them at full scale in their regular coding tasks.

We, therefore, set out to address the following question:

Central Question: *Is it possible to improve automated refactoring by considering testing effort?*

To guide the search of refactoring opportunities, we assess both design improvement (measured as the number of anti-patterns' occurrences) and testing effort of the refactored design for each refactoring sequence found by an EMO search algorithm.

6.2.1 Testing effort measurement

We refer to testing effort as the number of test cases required for each class in a software system, according to the Minimal Data members Usage Matrix (MaDUM) testing strategy [100]. MaDUM as well as other object-oriented (OO) testing strategies, *e.g., state-based condition* [163], and the pre-and-post conditions testing [101] have been proposed to overcome the limitations of traditional techniques, *e.g.,* white-box and black-box testing, when testing OO systems. Indeed, as pointed out by many authors [100, 163], the traditional testing strategies used in the context of procedural programming are insufficient to test OO programs because they are conceived to test functions as stand-alone code units, raising the possibility of missing state-based errors occurring during intra-method interactions. Among the testing strategies that consider the OO paradigm, we choose MaDUM because it does not

require any kind of software artifact apart from the source code. Hence, a simple static analysis of the source code is enough to estimate the number of test cases required to find code deviations. Then, that estimation can be leveraged by an automated approach to guide the refactoring process towards a design that minimizes the unit testing effort. Because testing all possible interactions between methods and attributes within a class is expensive, if not impossible, OO testing strategies seek to reduce the number of sequences of methods to test. MaDUM testing uses a divide to conquer strategy to perform unit testing: the class is divided in data slices and its correctness is evaluated in terms of the correctness of all its slices tested separately. A *data slice* is the set of methods that access to a particular attribute (field) in a class. The identification of the data slices is based on the *Enhanced Call-Graph (ECG)* and the *MaDUM*. The ECG represents the type of usages among the members of a class and it is defined as: $ECG(C) = (M(C), F(C), Emf, Emm)$, where $M(C)$ is the set of methods of C, $F(C)$ is the set of fields of C. $Emf = (m_i, f_j)$ indicates that method i accesses field j, and $Emm(m_i, m_j)$ that method i invokes method j.

MaDUM is an $nf \times nm$ matrix where nf and nm are the numbers of fields and methods in the class. It is built using the ECG of the class. MaDUM defines four categories to classify the methods, that are: class constructors (c), transformers (t), *i.e.,* methods that modify the state of a field, reporters (r), *i.e.,* methods that return the value of an attribute, and others (o), *i.e.,* methods that do not fall in the previous categories. Once the MaDUM of a class has been built, the order for testing that class is the following: first reporters are tested to ensure that they do not alter the state of the attribute they are reporting on. Constructors are then tested to ensure that attributes are correctly initialized, and in the right order. The testing of transformers is performed by generating for each slice all permutations of transformers in that slice for each constructor context. For example, let c be the set of constructors and t the set of transformers in a given slice, it is necessary to produce $|c| \times |t|!$ test cases, where the function $|x|$ denotes the cardinality of the set x. Others (o) are tested using traditional black or white-box testing. Note that a method m_j can access a field f_i directly or indirectly through another method m_k invoked by m_j. Although, in the last scenario, if m_j accesses f_i only through m_k, and m_k has been already tested in the f_i slice, there is no need to retest m_j in the slice f_i. The total number of test cases required to test a given class is computed as follows:

$$te(C_i) = |c| + |r| + |o| + \sum_{i=1}^{n} |c_i| * |t_i|! \tag{6.1}$$

Where $|x|$ is the number of methods of type x in the class, n the number of slices in the class, and $|x_i|$ the number of methods of type x in the slice i. When a class in a system presents a high number of transformers in a slice, *i.e.,* methods that modify the state of an attribute,

the probability of having points of failures increases, and consequently a higher number of test cases is required in order to thoroughly test the class. Hence to reduce testing effort and the risk of failures, due to state inconsistencies, the number of transformers within a slice should be kept as low as possible. Considering testing effort in automated-refactoring can alleviate the problem by prioritizing refactorings that reduce the number of transformers, and therefore the number of test cases required. For example, for each slice we need a number of test cases equal to the number of permutations of slice transformers multiplied by each constructor context. Hence, if we apply *move method* refactoring to move one or more of these transformers, from a large class to any other class in the system , which has a low number of transformers, the number of test cases required for the large class will decrease in a significant proportion, while the number of test cases for the small class will slightly increase, making the sum of test cases required for both classes less than before refactoring. Nevertheless, certain refactorings like the introduction of parameter object class increase the number of test cases (one for each parameter extracted from the source class to the new class object, plus one for the new constructor).

6.3 Testing-Aware Automated Refactoring

This section presents the foundations of our proposed approach *TARF* that aims to improve the design quality of OO systems, while minimizing the effort required to test the system.

Algorithm 8 summarizes the main steps of *TARF*. We describe each step in more details in the following paragraphs.

Generation of Abstract Model. This step consists in generating a graph representation of the system under maintenance that contains all the components (classes, methods, and attributes) of the system and the relationships between these components. This graph will be used to detect anti-patterns, compute the testing effort, and apply different refactoring sequences until we find an optimal solution.

Computation of MaDUM. At this step, we compute for each individual class, the corresponding ECG, and MaDUM. Once the MaDUM is built, we can compute the number of test cases required per class, and then by adding these numbers for all classes in the system, we obtain the overall testing effort for the system.

Detection of anti-patterns. This step consists in detecting anti-patterns in the system. We use the count of anti-patterns to assess the design quality of the system.

Generation of refactoring opportunities. This step consists in selecting refactoring operations that can remove anti-patterns detected in the previous step.

Search-based refactoring using EMO. At this step, we apply search-based techniques to

Algorithm 8: TARF Approach

```
   Input  : Software system to refactor (SW)
   Output: Optimal refactoring sequence(s)
1  Pseudocode TARF(SW)
2      AM=Generation of Abstract Model
3      MAD=Computation of the MaDUM matrix
4      AP=Detection of Anti-patterns
5      RS=Generation of refactoring opportunities
6      Search-based refactoring(AM, MAD, RS)
7  Procedure Search-based refactoring(AM,MAD,RS)
8      AM' = AM
9      M = set_of(MaDUM)
10     S = sequence_of(RS)
11     P_0 = GenerateInitialPopulation(S)
12     A = ∅
13     for all S_i ∈ P_0 do
14         apply_refactorings(AM', S_i)
15         compute_Quality(AM')
16         compute_TestingEffort(AM', MAD)
17     end for
18     Evaluation(P_0)
19     A_0 = Update(A_0, P_0)
20     t = 0
21     while not StoppingCriterion do
22         t = t + 1
23         P_t = Variation_Operators(A_{t-1}, P_{t-1})
24         for all S_i ∈ P_t do
25             apply_refactorings(AM', S_i)
26             compute_Quality(AM')
27             compute_TestingEffort(AM', MAD)
28         end for
29         Evaluate(P_t)
30         A_t = Update(A_t, P_t)
31     end while
32     best_solution = A
33     return best_solution
```

find the best sequence of refactorings that achieves a maximum reduction of the number of anti-patterns, while keeping the testing effort as minimum as possible. Since we formulate the problem as MO, we need to use metaheuristic techniques that can lead to fast results in the set of *non-dominated* solutions, *i.e.,* the ones that provide a trade-off between anti-patterns removal and test cases reduction. A generic template for EMO algorithms is presented from line 8 to line 34. The algorithm takes as input the abstract model (AM), the set of MaDUMs for each class (MAD), and the list of candidate refactorings (RS). With this information, it generates the initial population P_0, and updates the set of non-dominated solutions found in this first sample, A (lines 8-20). After generating the initial population, the main search loop starts (line 22). Solutions, *i.e.,* refactoring sequences, included in P and A are varied randomly and a new set of solutions is generated (line 24) and evaluated. The evaluation of the candidate solutions is performed in three steps: (1) the sequence of refactorings is applied to a copy of the abstract model (AM'); (2) a map of anti-patterns is generated from the refactored code and the quality of the system is computed. Next (3) the testing effort is computed after recomputing the MaDUMs for the modified classes. After the evalua-

tion of a set of solutions, solutions that are not non-dominated are retrieved (line 31). The process ends when the algorithm reaches the stop condition. Examples of stop conditions that are commonly used include a predetermined execution time, or a predefined maximum number of evaluations.

6.4 Case Study Design

The *goal* of this case study is to assess the effectiveness of TARF in correcting anti-patterns in OO systems, while reducing the effort required to test the system.

The *quality focus* is the improvement of the design quality of OO systems and the reduction of testing effort through search-based refactoring. The *perspective* is that of researchers interested in developing automated refactoring tools and practitioners interested in improving the design quality of their software system while controlling for testing effort.

The *context* consists of four open-source software systems (ArgoUML, Gantt Project, JHotDraw, and Mylyn) and four evolutionary metaheuristics one single objective algorithm (*i.e.,* Genetic Algorithm (GA)) and three MO algorithms (MOCell, NSGA-II, and SPEA2). Table 6.1 presents relevant information about the systems under study. We select these software systems because (1) they are open-source systems, with different purposes and sizes; and (2) they have been used in previous studies on anti-patterns and refactorings [52, 24, 72].

Table 6.1 Descriptive statistics of the studied systems.

Name	Number of classes	Number of anti-patterns	Number of initial test cases
ArgoUML 0.34	1,754	456	587,220,340
GantProject 1.10.2	188	38	2,510
JhotDraw 5.4	450	89	10,943
Mylyn 3.4	2,365	183	7,303,813

We instantiate our generic approach TARF using four different metaheuristics that we briefly describe below.

GA [57, 58]. This is the same implementation that we use in Section 4.4.3 including the same solution representation, selection and variation operators. The process of selection and recombination is guided by a combined single objective function obtained by multiplying Equation (4.1) by Equation (6.2).

$$STF = \sum_{i=1}^{n} te(C_i), \tag{6.2}$$

where STF is the test effort of the system, te is calculated from Equation (6.1), and n is the total number of classes. We aim to minimize the value of STF.

NSGA-II, SPEA2, and MOCell [1]. These are EMO algorithms where each objective function is evaluated separately, generating typically more than one non-dominated solution. . While each EMO defines their own selection mechanism, the solution representation as well as the variation operators remain the same. We choose these for metaheuristics because they are evolutionary techniques that have been successfully applied to solve combinatorial discrete problems in several contexts. We decide to compare mono-objective with MO metaheuristics to prove that a trade-off between design quality and testing effort exists. Otherwise, GA should perform as good as EMOs in finding an optimal refactoring sequence.

6.4.1 Parameters of the metaheuristics.

We are using four evolutionary metaheuristics in our experiments. As we mentioned before, they make use of variation operators (selection, mutation and crossover) to move through the decision space in the search for an optimal solution. To determine the best parameters for our metaheuristics, we run each algorithm with different configurations 30 times, in a *factorial design* in the following way: we test 16 combinations of mutation probability p_m = (1, 0.8, 0.5, 0.2), and crossover probability $p_c = (1, 0.8, 0.5, 0.2)$, and obtained the best results with the pair (0.8, 0.8). This is not a surprise as in [69] they found high mutation and crossover values to be the best trade for algorithm performance. For the specific problem of automated refactoring, setting the initial size of the refactoring sequence is crucial to find the best sequence in a reasonable time, especially when we have a huge number of candidate refactorings, because setting a low value will lead to find poor solutions in terms of anti-patterns correction. On the contrary, if the initial size is very large, we may obtain the reverse effect because applying many refactorings not necessarily implies better quality, as refactorings can improve one aspect of quality while worsen others. Hence, we experiment running the algorithms with three relative thresholds: 25%, 50%, 75% and 100%, of the total number of refactoring opportunities, and found that 50% give us the best results in terms of removal of anti-patterns and reduction of testing effort. The population size is set to 100 individuals as default value.

We implement all the metaheuristics described before using the jMetal Framework [164], which is a wide-use library for solving optimization problems. Given that we are comparing techniques with different sources of information (population, archive, etc.), we opt for number of evaluations as the stop criteria, and set it to 2500, which is an accepted value for

1. Note that the version of MOCell used in this dissertation is an *asynchronous* version of MOCell called aMOCell4 [66]

optimization problems in general.

6.4.2 Dependent and Independent Variables

To assess whether TARF can improve design quality while reducing testing effort, we consider the following dependent and independent variables:
The *independent variables* are our four selected metaheuristics, *i.e.,* GA, MOCell, NSGA-II, and SPEA2.
The *dependent variables* are the following two metrics used to evaluate the effectiveness of TARF at improving the design quality of systems while reducing the testing effort.

— Difference of anti-patterns occurrences after refactoring (APR): for each metaheuristic and each system, we subtract the number of anti-patterns occurrences after refactoring from the number of anti-patterns occurrences before refactoring . APR is an indication of the improvement/worsening of the design quality of the system. The larger the difference, the better is the design quality of the system.

— Difference of required test cases after refactoring (TCR): for each metaheuristic and each system, we subtract the total number of test cases after refactoring from the total number of test cases before refactoring. TCR is an indication of the improvement/-worsening of testing effort of a system. The larger the difference, the less is the effort required to test the system.

The anti-patterns considered for evaluating TARF are the same used for evaluating RePOR (*cf.* Section 5.3).

6.4.3 Research Questions

To better answer the central question of this chapter, we formulate the following research questions :

(RQ1) To what extent can TARF correct anti-patterns and reduce testing effort?
This research question aims to assess the effectiveness of TARF at improving design quality, while reducing testing effort.

(RQ2) To what extent is design quality improved after refactoring when considering testing effort?
While the number of anti-patterns in a system serves as a good estimation of design quality, there are other quality attributes such as those defined by the QMOOD quality model [40] that are also relevant for developers. This research question aims to assess the impact that the application of TARF has on these aforementioned attributes.

6.4.4 Analysis Method

In order to measure the performance of the MO metaheuristics used in this chapter, we need to consider the quality of their resulting non-dominated set of solutions [165]. We use two quality indicators for that purpose; the Hypervolume (HV) and Spread (Δ). HV provides a measure that considers the convergence and diversity of the resulting approximation set. Higher values of the HV metric are desirable. Spread measures the distribution of solutions into a given front. Lower values close to zero are desirable, as they indicate that the solutions are uniformly distributed. For further details we refer the reader to the source references [166, 63].

To answer RQ2, we use QMOOD [40] to evaluate the impact of the proposed refactoring sequences on five quality attributes as we did for ReCon (*cf.* Sections 2.4, 4.4.5).

We obtain the quality gain of the refactored design (D') by dividing each quality attribute value by the corresponding value for the original design (D) as previous work [22].

6.5 Case Study results

In this section we present the results of our case study with respect to the two research questions of this Chapter.

6.5.1 (RQ1) To what extent can TARF correct anti-patterns and reduce testing effort?

One main feature of MO metaheuristics is that they do not produce a single solution as mono-objective techniques, *e.g.,* GA, but a set of solutions. From this set of solutions we are interested in those that are non-dominated, *i.e.,* the solutions whose objective values cannot be improved without worsening others. Pareto reference Front (RF) is an approximation of the true Pareto Front (*cf.* Section 2.3.4), and similar to other combinatorial optimization studies [45], we assume that the production of the true Pareto front is not feasible, hence we use the reference front, created from the optimal values after 30 independent executions. In Figure 6.1 we present the Pareto reference front for JHotDraw, extracted from the three MO metaheuristics analyzed. The points in Figure 6.1 represent a compromise between quality and testing effort. The $x-axis$ represents the normalized values of quality, measured in terms of number of anti-patterns corrected; $y-axis$ represents the number of required test cases for that solution. The best solutions are found in the right-bottom corner of the plot. Hence, the software maintainer is able to choose a solution according to its preferences, for example if someone is interested mostly in reducing the number of test cases, (s)he can

opt for a solution located in the left-bottom of the plot. However, that solution would not remove as many anti-patterns as the solutions located in the middle or in the extreme right position in the plot. The advantage of considering testing as another objective function is that maintainers obtain the possibility to choose among trade multiple solutions.

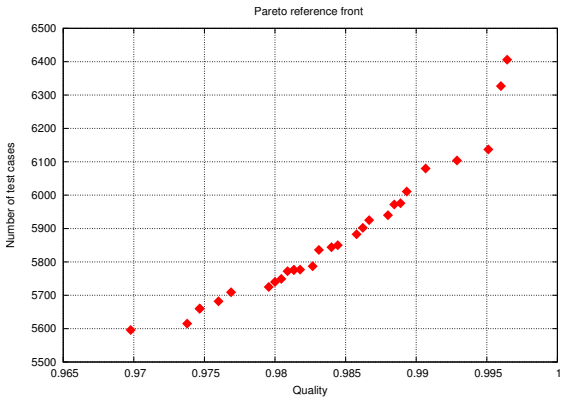

Figure 6.1 The Pareto reference front of JHotDraw.

Note that to compare the three MO with GA, we need to select one solution (**bs**) in the Pareto front. One technique often used to determine the best solution in the Pareto Front of a problem, is selecting the solution that has the minimal distance with a hypothetical *ideal solution* using the Euclidean distance. The *ideal solution* for our approach is the one where we end with zero anti-patterns (1 according to Equation (4.1)) and the number of test cases is close to zero. The complete equation is provided in Equation (6.3).

$$bs = \min_{i=1}^{n} \left(\sqrt{(1 - DQ[i])^2 + (te[i])^2} \right), \tag{6.3}$$

where n is the number of solutions in the Pareto front returned by the MO metaheuristic.

Once we extract the best solution from the Pareto reference front, we are able to compare it with the one of the single objective metaheuristic (GA). In Table 6.2 we present the number of anti-patterns corrected, per anti-pattern, in total, and the resulting number of test cases after applying the four metaheuristics studied. The values presented in Table 6.2 are median values for the 30 independent runs we performed for each algorithm. Columns 2 to 6 are the type of anti-patterns; APR is the difference of the number anti-patterns occurrences after refactoring; TCR is the difference of the number of required test cases after refactoring, and ROs is the number of refactoring operations applied. A negative number in a column indicates the worsening of the metric with respect to the original variable measured, and "−" (columns 2-6) indicates zero anti-patterns of the column type detected. The best results of columns APR and TCR are highlighted with dark grey. We observe that in general

the three MO metaheuristics reduce more the number of test cases than the single approach. Among the four EAs implemented, MOCell achieved the best results concerning testing effort reduction (about 50% median improvement compared to the number of test cases required for the original system version (before refactoring). Moreover, in two systems (ArgoUML, JHotdraw) the mono-objective technique did not reduce, but considerably increase the testing effort. This can be explained due to the large number of long-parameter list classes that this systems contains. The application of IPO refactoring, add new classes to the design causing an increment of the number of test cases.

Concerning anti-pattern correction, GA overcomes the MO metaheuristics in ArgoUML, JHotDraw and Mylyn, the same systems, being the former one the one with highest correction of anti-patterns, but with the highest increase of test cases at the same time. Gantt project is championed by MOCell, and in Mylyn the average of the two metrics favor MOCell as well. The median anti-patterns correction improvement for GA reached about 70% while for SPEA2, which was the second best EAs technique in this metric, reached 46%. This result indicates the existence of conflict between testing effort and anti-patterns' correction. That correcting anti-patterns without considering testing effort, could negatively impact the testing effort for the system after refactoring. With respect to the number of refactorings applied (column ROs), we observe that while longer refactoring sequences seem to reduce more anti-patterns, at least for the three first studied systems, a pattern to characterize the behavior of testing effort is less evident. While in ArgoUML and JhotDraw the metaheuristics with the shortest sequences report the less number of test cases, in Mylyn and Gantt the trend is inverted.

Table 6.2 Median count of anti-patterns removed, and number of test cases after refactoring.

Metaheuristic	BL	LC	LP	SC	SG.	APR	TCR	ROs
ArgoUML								
GA	-1	34	335	1	1	370	-1336	1847
MOCell	0	1	8	0	0	9	140	64
NSGA-II	0	1	9	0	0	10	177	63
SPEA2	0	2	14	0	0	16	218	92
Gantt Project								
GA	0	1	12	–	2	15	-55	93
MOCell	4	2	14	–	4	24	535	381
NSGA-II	3	2	9	–	4	18	519	317
SPEA2	3	2	13	–	4	22	499	346
JHotDraw								
GA	–	1	59	–	–	60	-507	454
MOCell	–	0	21	–	–	21	5321	216
NSGA-II	–	1	28	–	–	29	5283	350
SPEA2	–	0	30	–	–	30	5301	389
Mylyn								
GA	1	19	101	–	–	121	556	1959
MOCell	1	18	77	–	–	96	7266802	2159
NSGA-II	1	17	81	–	–	99	7266748	1974
SPEA2	1	19	83	–	–	103	7266765	2047

We conclude that only considering anti-patterns' correction without taking into account testing effort can negatively impact the testability of a system after refactoring. That a multiobjective formulation which includes testing effort as an objective to minimize when applying automatic refactoring can significantly reduce the number of test cases, while keeping reasonable correction results.

Performance of the three multiobjective metaheuristics. In Table 6.3 we present the mean and the standard deviation of the quality indicators (HV, Spread) values of the metaheuristics for each system on 30 independent runs. A special notation appears in the table: a gray colored background denotes the best (dark gray) and second-best (lighter gray) performing technique. According to the HV indicator, MOCell has been able to approximate the Pareto fronts with the highest accuracy, while the second-best is achieved by NSGAII, except for JHotDraw, where SPEA2 overcomes NSGAII. Concerning the spread indicator, the best spread is divided between SPEA2 and MOCell, and NSGA-II appears to be the less effective metaheuristic. To determine the significance of the obtained results, we compute the Wilcoxon rank-sum test between each pair of metaheuristics. The results are summarized in Table 6.4. In each cell, a ▲ or a ▽ symbol implies a p-value < 0.05, indicating that the null hypothesis (the median difference between pairs of observation is zero) is rejected; otherwise, a – is used. The ▲ denotes that the metaheuristic in the row obtained a better value than the one in the column; the ▽ indicates the opposite. Hence, the only conclusion we can draw from these results is that 1) MOCell overcomes SPEA2 and NSGA-II in more than a half of the systems analyzed in terms of HV, while the performance between NSGA-II and SPEA2 remains unclear. Concerning the spread indicator, we omit the results of the Wilcoxon test because the results were not statistically significant, thus we cannot draw any conclusion about the performance of the metaheuristics using this indicator. Note that the aim of this chapter is not to propose a new multiobjective algorithm to perform automated refactoring, but *reformulate* the problem of refactoring to include testing effort as a goal regardless of the metaheuristic employed.

Although the obtained results point out that MOCell is the most effective technique for the formulation of automatic refactoring considering quality and testing effort, and among the metaheuristics studied, further studies with more systems, and more quality indicators are required to validate this result.

Table 6.3 Quality indicators: Mean and standard deviation

	SPEA2	MOCell	NSGAII
	Hypervolume		
ArgoUML	$4.80e-01_{5.7e-03}$	$5.04e-01_{4.8e-03}$	$4.89e-01_{1.1e-02}$
Gantt	$0.00e+00_{0.0e+00}$	$2.30e-01_{1.6e-01}$	$1.00e-02_{2.0e-02}$
JHotDraw	$5.37e-01_{2.3e-02}$	$5.93e-01_{2.5e-02}$	$5.36e-01_{3.0e-02}$
Mylyn	$1.49e-01_{4.6e-02}$	$2.40e-01_{5.7e-02}$	$2.01e-01_{3.5e-02}$
	SPREAD		
ArgoUML	$5.60e-01_{1.0e-01}$	$5.94e-01_{6.3e-02}$	$6.12e-01_{2.2e-02}$
Gantt	$8.48e-01_{8.1e-02}$	$7.89e-01_{1.7e-01}$	$8.97e-01_{8.3e-02}$
JHotDraw	$6.95e-01_{4.3e-02}$	$6.45e-01_{5.0e-02}$	$7.23e-01_{3.7e-02}$
Mylyn	$9.63e-01_{1.3e-01}$	$9.77e-01_{1.6e-01}$	$1.09e+00_{2.3e-01}$

Table 6.4 Wilcoxon rank-sum test for HV indicator.

	MOCell				NSGA-II			
	ARG	GAN	JHD	MYL	ARG	GAN	JHD	MYL
SPEA2	▽	▽	▽	–	–	–	–	
MOCell					–	▲	▲	–

(RQ2) To what extent is design quality improved after refactoring when considering testing effort?

Although we have shown that automated refactoring can improve the quality of the system and reduce the testing effort, some software maintainers may wonder whether the refactorings applied will produce a new code that is still readable, or if it will be easy to come back later and modify it or extend it. Since concepts such as reusability, or understandability in design quality are quite vague and hard to define, we consider the QMOOD evaluation functions as *examples* of how to correctly characterize good design properties. In Figure 6.2 we present the obtained quality gain values (change quotient) that we computed for each QMOOD quality attribute (QQA) before and after refactoring for each studied system.

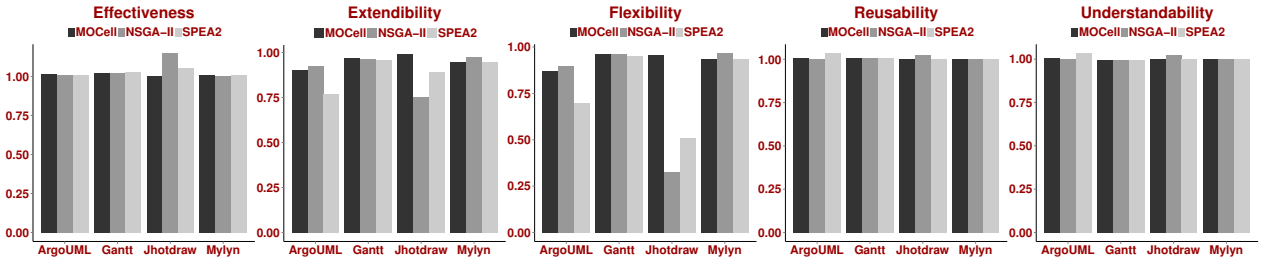

Figure 6.2 The quality gain of the best refactoring solutions on QMOOD quality attributes.

We can observe that the system quality increases across the five QQA in an even manner, that ranges from 1.15 in effectiveness (Jhotdraw, MOCell) to 0.33 in flexibility (JHotDraw, NSGA-II).

We suggest that the low value in flexibility compared to effectiveness is in part the result of

the weight that each of these two quality function assigns to MOA metric (0.2, 0.5 respectively). According to QMOOD, MOA is the number of user-defined classes, and we observe in Table 6.2 that MOCell removed less long parameter list anti-patterns than GA. We remind that the suggested refactoring for LP is introduce parameter-object, which creates new classes to store the long parameter list, hence the increment in the number of user-defined classes is less in MOCell compared to GA. On the other hand, effectiveness assigns a lower weight to this metric (0.2), but integrates other desirable metrics related to OO design like abstraction, encapsulation and inheritance. Finally, understandability, reusability, and extendibility factors are benefited from the extensive application of move method refactorings, and reported an increment similar to effectiveness, because Move method is known to impact metrics like coupling (DCC), cohesion (CAM) and design size (DSC) that serves to calculate these quality attributes.

We conclude that our approach was successful in improving design quality not only in terms of anti-patterns correction, but also in terms of quality attributes such as understandability, reusability, flexibility, effectiveness and extendibility.

6.6 Threats to validity

This section discusses the threats to validity of TARF's case study following common guidelines for empirical studies [143].

Construct validity threats concern the relation between theory and observation. This is mainly due to possible mistakes in the detection of anti-patterns, in the refactorings applied. We based the anti-patterns detection on DECOR [52], and despite the high recall and precision of DECOR, there is no warranty that we detect all the possible anti-patterns, or that those detected are indeed true anti-patterns. Concerning the application of refactorings, we manually validate the outcome of refactorings performed in source code compared to the ones applied to the abstract model to ensure that the output values of the objective functions correspond to the changes performed. However, we rely on the correct representation of the code by the abstract model. In this study we use PADL [150], which has been used in several studies concerning anti-patterns, design patterns, and software evolution.

A second threat is the use of MaDUM as proxy to estimate the testing effort, because other techniques could bring different results. Moreover, MaDUM estimation does not include the effort of writing and running each test case. Instead, it gives an estimate of the number of test cases required to test the class and highlights classes with multiple transformers as difficult

classes to be tested. Finally, MaDUM only works at the level of unitary testing, without considering class interactions. Therefore, we can only claim that, no matter the testing strategy, automated-refactoring approaches should consider the impact of refactorings not only in terms of design quality but in testing effort.

Threats to internal validity concern our selection of anti-patterns, tools, and analysis method. In this study we used a particular yet representative subset of anti-patterns as proxy for design quality.

Conclusion validity threats concern the relation between the treatment and the outcome. We paid attention not to violate assumptions of the constructed statistical models. In particular, we used a non-parametric test, Wilcoxon rank sum, that does not require any assumption on the underlying data distribution.

Threats to external validity concern the possibility to generalize our results. Our study focuses on four open source software systems having different sizes and belonging to different domains. Nevertheless, further validation on a larger set of software systems is desirable, considering systems from different domains, as well as several systems from the same domain. Future replications of this study are necessary to confirm our findings.

6.7 Chapter Summary

In this chapter, we propose TARF, a novel approach that includes testing effort as a mean to improve the testability of software systems during refactoring operations. Specifically, we set out to address the following question:

Central Question: *Is it possible to improve automated refactoring by considering testing effort?*

We answer this question by performing a case study where we validate TARF using a testbed comprised of four open-source systems, and found that the solutions proposed by TARF maintain a compromise between number of anti-patterns corrected, and number of test cases required to test an OO system, according to MaDUM strategy. We implemented TARF using four metaheuristics, and found that MO algorithms lead to better results than single objective GA.

Finally, we assessed the design quality of the solutions proposed using five quality attributes defined in the hierarchical QMOOD model, and found that we can increase the quality in terms of reusability, flexibility, understandability, effectiveness, and extendibility.

CHAPTER 7 IMPROVING AUTOMATED REFACTORING BY CONTROLLING FOR ENERGY EFFICIENCY

7.1 Introduction

During the last five years, and with the exponential growth of the market of mobile apps [167], software engineers have witnessed a radical change in the landscape of software development. From a design point of view, new challenges have been introduced in the development of mobile apps such as the constraints related to internal resources, *e.g.,* CPU, memory, and battery; as well as external resources, *e.g.,* internet access. Moreover, traditional desired quality attributes, such as functionality and reliability, have been overshadowed by subjective visual attributes, *i.e.,* "flashiness" [168].

Mobile applications (apps) play a central role in our life today. We use them almost anywhere, at any time and for everything; *e.g.,* to check our emails, to browse the Internet, and even to access critical services such as banking and health monitoring. Hence, their reliability and quality is critical. Similar to traditional desktop applications, mobile apps age as a consequence of changes in their functionality, bug-fixing, and introduction of new features, which sometimes lead to the deterioration of the initial design [169]. Beside OO anti-patterns, which we studied in previous chapters of this dissertation, resource management is critical for mobile apps. Developers should avoid anti-patterns that cause battery drain. An example of such anti-pattern is *Binding resources too early class* [170]. This anti-pattern occurs when a class *switches on* energy-intensive components of a mobile device (*e.g.,* Wi-Fi, GPS) when they cannot interact with the user. Another example is the use of *private getters and setters* to access class attributes in a class, instead of accessing directly the attributes. The Android documentation [51] strongly recommends to avoid this anti-pattern as virtual method calls are up to seven times more expensive than using direct field access [51].

Previous studies have pointed out the negative impact of anti-patterns on change-proneness [4], fault-proneness [171], and maintenance effort [172]. In the context of mobile apps, Hecht et al. [110] found that anti-patterns are prevalent along the evolution of mobile apps. They also confirmed the observation made by Chatzigeorgiou and Manakos [10] that anti-patterns tend to remain in systems through several releases, unless a major change is performed on the system.

One critical concern of mobile apps development is reducing energy consumption, due to the short life-time of mobile device's batteries. Some research studies have shown that refac-

torings that are applied to remove anti-patterns can impact the energy consumption of a system [118, 173, 122]. Hecht et al. [49] observed an improvement in the user interface and memory performance of mobile apps when correcting Android anti-patterns, like *private getters and setters*, *HashMap usage* and *member ignoring method*, confirming the need of refactoring approaches that support mobile app developers.

We, therefore, set out to address the following question:

Central Question: *Is it possible to improve automated refactoring of mobile apps by considering energy concerns?*

In this chapter we provide mobile developers with a refactoring solution that address energy as well as OO design concerns. We propose a multiobjective refactoring approach called Energy-Aware Refactoring approach for MObile apps (EARMO) to detect and correct anti-patterns in mobile apps, while improving energy consumption.

The results of this chapter can be summarized in the following contributions.

1. An empirical study of the impact of anti-patterns on the energy consumption of mobile apps. We propose a methodology for a correct measurement of the energy consumption of mobile apps. Our obtained results provide evidence to support the claim that developer's design choices can improve/decrease the energy consumption of mobile apps.
2. A novel automated refactoring approach to improve the design quality of mobile apps, while controlling energy consumption. EARMO provides developers with the best trade-off between two conflicted objectives, design quality and energy.
3. The evaluation of the effectiveness of EARMO using three different multiobjective metaheuristics from which EARMO is able to correct a median of 84% anti-patterns.
4. A manual evaluation of the refactoring recommendations proposed by EARMO for 13 apps. The manual evaluation is conducted in two steps. (1) Each refactoring operation in a sequence is validated and applied to the corresponding app. (2) The app is executed in a typical user context and the energy consumption gain is recorded. The sequences generated by EARMO achieve a median precision score of 68%. EARMO precision is close to previously published refactoring approaches (*e.g.,* Ouni et al. [174] reports that Kessentini et al. [24] achieves a precision of 62-63% and Harman et al. [45]. a precision of 63-66%). In addition, EARMO extended the battery life by up to 29 minutes when running in isolation a refactored multimedia app with default settings (no Wi-Fi, no location services, minimum screen brightness).

5. From the manual validation, we provide guidelines for toolsmiths interested in generating automated refactoring tools.
6. We perform the evaluation of the design quality of the refactored apps using a QMOOD [40] and report a median improvement of 41% in extendibility of app's design.
7. We evaluate the usefulness of the solutions proposed by EARMO from the perspective of mobile developers through a qualitative study and achieve an acceptance rate of 68%. These results complement the manual verification in terms of precision and design quality (e.g., extendability, reusability), and serve as external evaluation.

7.1.1 Energy measurement of mobile apps

Energy consumption, a critical concern for mobile and embedded devices, has been typically targeted from the point of view of hardware and lower-architecture layers by the research community. Energy is defined as the capacity of doing work while power is the rate of doing work or the rate of using energy. In our case, the amount of total energy used by a device within a period of time is the energy consumption. *Energy* (E) is measured in *joules* (J) while *power* (P) is measured in *watts* (W). Energy is equal to power times the time period T in seconds. Therefore, $E = P \cdot T$. For instance, if a task uses two watts of power for five seconds it consumes 10 Joules of energy.

One of the most used energy hardware profilers is the *Monsoon Power Monitor*[1]. It provides a power measurement solution for any single lithium (Li) powered mobile device rated at 4.5 volts (maximum three amps) or lower. It samples the energy consumption of the connected device at a frequency of $5\,kHz$, therefore a measure is taken each 0.2 milliseconds. Other works use the LEAP power measurement device [175]. LEAP contains an ATOM processor that runs Android-x86 version 2.x. Its analog-to-digital converter samples CPU energy consumption at a frequency of $10\,kHz$.

In this chapter, energy consumption is measured using a more precise environment. Specifically we use a digital oscilloscope *TiePie Handyscope HS5* which offers the *LibTiePie SDK*, a cross platform library for using *TiePie* engineering USB oscilloscopes through third party software. We use this device because it allows to measure using higher frequencies than the *Monsoon* and *LEAP*. The mobile phone is powered by a power supply and, between both, we connect, in series, a *uCurrent*[2] device, which is a precision current adapter for multimeters converting the input current (I) in a proportional output voltage (V_{out}). Knowing I and the voltage supplied by the power supply (V_{sup}), we use the *Ohm's Law* to calculate the power

1. `https://www.msoon.com/LabEquipment/PowerMonitor/`
2. `http://www.eevblog.com/projects/ucurrent/`

usage (P) as $P = V_{sup} \cdot I$. The resolution is set up to 16 bits and the frequency to $125\,kHz$, therefore a measure is taken each eight microseconds. We calculate the energy associated to each sample as $E = P \cdot T = P \cdot (8 \cdot 10^{-6})s$. Where P is the power of the smart-phone and T is the period sampling in seconds. The total energy consumption is the sum of the energy associated to each sample.

A low sampling frequency can make it very hard to assess the energy consumption of any given method. Consider, for example, the *glTron*[3] application. According to our measurements, the method `com.glTron.Video.HUD.draw` has an execution time (inclusive of called methods) of 91.96 milliseconds. Thus, sampling at $125\,kHz$ (one sample each eight microseconds) or $10\,kHz$ (one sample each 0.1 milliseconds) does not make a big difference as enough data points will be collected. However, if we consider for the same package (`com.glTron`) the method `...Video.GraphicUtils.ConvToFloatBuffer`, its execution lasts only 732 microseconds. Measuring at $10\,kHz$, limits the collection of data points about this method to no more than 7 samples, while measuring at $125\,kHz$ we could collect data points up to 92 samples. In essence, if a method execution last more than one millisecond, such as in `com.glTron.Video.HUD.draw`, the errors will generally averaged out, making the energy estimation error low or even negligible. However, in methods of short duration (less than one millisecond) the error may be higher. Li et al. [176] studied what granularity of measurements is sufficient for measuring energy consumption. They concluded that nanosecond level measurement is sufficient to capture all API calls and methods. This raises another problem, the bottleneck in high-frequency power sampling due to the storage system, which cannot save power samples at the same frequency as the power meter can generate them. However, Saborido et al. [177] found that sampling at $125\,kHz$ just accounts for about 0.7% underestimation error. Therefore we consider that $125\,kHz$ is sufficient to measure the energy consumption of mobile applications.

In our experiments, we used a LG Nexus 4 Android phone equipped with a quad-core CPU, a 4.7-inch screen and running the Android Lollipop operating system (version 5.1.1, Build number LMY47V). We believe that this phone is a good representative of the current generation of Android mobile phones because more than three million have been sold since its release in 2013[4], and the latest version of Android Studio includes a virtual device image of it for debugging.

We connect the phone to an external power supplier which is connected to the phone's motherboard, thus we avoid any kind of interference with the phone battery in our measurements.

3. https://f-droid.org/wiki/page/com.glTron
4. `https://goo.gl/6guUpf`

The diagram of the connection is shown in Figure 7.1. Note that although we use an external power supplier, the battery has to be connected to the phone to work. Hence, we do not connect the positive pole of the battery with the phone.

To transfer and receive data from the phone to the computer, we use a USB cable, and to avoid interference in our measurements as a result of the USB charging function, we wrote an application to disable it [5]. This application is free and it is available for download in the *Play Store* [6].

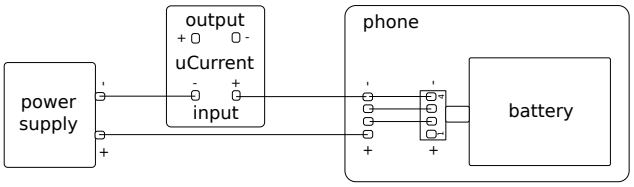

Figure 7.1 Connection between power supply and the Nexus 4 phone.

7.2 Preliminary Study

In this section we present a preliminary study that aimed to measure the impact of anti-patterns on energy consumption. Understanding if anti-patterns affect the energy consumption of mobile apps is important for researchers and practitioners interested in improving the design of apps through refactoring. Specifically, if anti-patterns do not significantly impact energy consumption, then it is not necessary to control for energy consumption during a refactoring process. On the other hand, if anti-patterns significantly affect energy consumption, developers and practitioners should be equipped with refactoring approaches that control for energy consumption during the refactoring process, in order to prevent a deterioration of the energy efficiency of apps.

We formulate the research questions of this preliminary study as follows:

(PQ1) Do anti-patterns influence energy consumption?
The rationale behind this question is to determine if the energy consumption of mobile apps with anti-patterns differs from the energy consumption of apps without anti-patterns. We test the following null hypothesis: H_{0_1}*: there is no difference between the energy consumption of apps containing anti-patterns and apps without anti-patterns.*

(PQ2) Do anti-pattern's types influence energy consumption differently?
In this research question, we analyze whether certain types of anti-patterns lead to more

5. The mobile phone has to be rooted first.
6. `https://goo.gl/wyUcdD`

energy consumption than others. We test the following null hypothesis: H_{0_2}*: there is no difference between the energy consumption of apps containing different types of anti-patterns.*

7.2.1 Design of the Preliminary Study

As mentioned earlier, we consider two categories of anti-patterns: (i) *Object-oriented (OO)* anti-patterns [3, 47], and (ii) *Android anti-patterns (AA)* defined by [48, 51]. Concerning (AA), previous works have evaluated the impact on energy consumption of *private getter and setters* [178, 31, 179] and found an improvement in energy consumption after refactoring.

The anti-patterns considered in this chapter are: **Blob (BL)**, **Lazy Class (LC)**, **Long Parameter list (LP)**, **Refused Bequest (RB)**, **Speculative Generality (SG)**. For Android, we considered **Binding resources too early (BE)**, **HashMap usage (HMU)**, **Private getters and setters (PGS)**. The definitions can be found in Table 2.1.

We select these anti-patterns because they have been found in mobile apps [110, 49], and they are well defined in the literature with recommended steps to remove them [47, 3, 48, 51].

To study the impact of the anti-patterns, we write a web crawler to download apps from *F-droid*, an open-source Android app repository [7]. The total number of apps retrieved by the date of April 14th 2016 is 200. These apps come from five different categories (Games, Science and Education, Sports and health, Navigation, and Multimedia). We filtered out 47 apps which Android version is lower than 2.1 because our transformation environment runs Windows 10 which supports Android SDK 2.1 or higher.

From the remaining 153 apps, we take a random sample that was determined using common procedures in survey design, with a confidence interval of 10% and a confidence level of 95%. Using these values, we obtained that the required sample size is 59 apps. This means that the results we get from our empirical study have an error at most of 10% with probability 0.95.

Next, we filtered apps where libraries referenced are missing or incomplete; apps that required to have *username* and *password* for specific websites; apps written in foreign languages and that we could not fully understand their functionality; apps that does not compile; apps that crashed in the middle of execution, or simply would not run in our phone device. The last filter is that the selected apps should contain at least one instance of any of the anti-patterns studied.

After discarding the apps that do not respect the selection criteria, we end-up with a dataset of 20 apps. Table 7.1 shows the selected apps.

7. `https://f-droid.org/`

Table 7.1 Apps used to conduct the preliminary study on Anti-patterns and Energy consumption.

App	Version	LOC	Category	Description
blackjacktrainer	0.1	3783	Games	Learning BlackJack
calculator	5.1.1	13985	Science & Education	Make calculations
gltron	1.1.2	12074	Games	3D lightbike racing game
kindmind	1.0.0	6555	Sports & Health	Be aware of sad feelings and unmet needs
matrixcalc	1.5	2416	Science & Education	Matrix calculator
monsterhunter	1.0.4	27368	Games	Reference for Monster Hunter 3 game
mylocation	1.2.1	1146	Navigation	Share your location
oddscalculator	1.2	2226	Games	Bulgarian card game odds calculator
prism	1.2	4277	Science & Education	Demonstrates the basics of ray diagrams
quicksnap	1.0.1	18487	Multimedia	Basic camera app
SASAbus	0.2.3	9349	Navigation	Bus schedule for South Tyrol
scrabble	1.2	3165	Games	Scrabble in french
soundmanager	2.1.0	5307	Multimedia	Volume level scheduler
speedometer	1	139	Navigation	Simple Speedometer
stk	0.3	4493	Games	A 3D open-source arcade racer
sudowars	1.1	22837	Games	Multiplayer sudoku
swjournal	1.5	5955	Sports & Health	Track your workouts
tapsoffire	1.0.5	19920	Games	Guitar game
vitoshadm	1.1	567	Games	Helps you to make decisions
words	1.6	7125	Science & Education	Helps to study vocabulary for IELTS exam

7.2.2 Data Extraction

The data extraction process is comprised of the following steps, which are summarized in Figure 7.2.

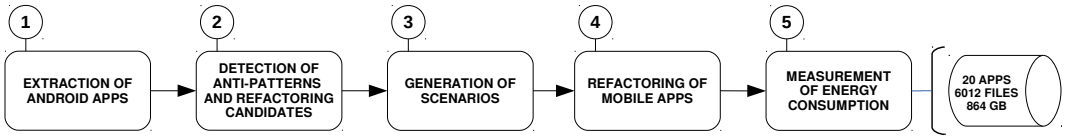

Figure 7.2 Data extraction process.

1. **Extraction of android apps.** We wrote a script to download the apps from *F-droid* repository. This script retrieves the name of the app, the link to the source code, Android API version, and the number of Java files. We use the API version to discriminate apps that are not compatible with our phone, and the number of Java files to filter apps with only one class. After filtering the apps, we import the source code in Eclipse (for the older versions) or Android Studio and ensure that they can be compiled and executed.
2. **Detection of anti-patterns and refactoring candidates**. The detection and gen-

eration of refactoring candidates is performed using our previous automated approach *ReCon* [145]. We use ReCon's current implementation for correcting object-oriented anti-patterns, and add two new OO anti-patterns (*Blob* and *Refused bequest*); we also add three Android anti-patterns based on the guidelines defined by Gottschalk [48], and the Android documentation [51]. As we explained before, in Chapter 4, ReCon supports two styles, root-canal (*i.e.,* to analyze all the classes in the system) and floss refactoring (*i.e.,* to analyze only the classes related to an active task in current developer's workspace provided by a task management integration plug-in). We use the root-canal style as we are interested in improving the complete design of the studied apps.

3. **Generation of scenarios**. For each app we define a scenario that exercises the code containing anti-patterns using the Android application *HiroMacro*[8]. This software allows us to generate scripts containing touch and move events, imitating a user interacting with the app on the phone, to be executed several times without introducing variations in execution time due to user fatigue, or skillfulness. To automatize the measurement of the studied apps we convert the defined scenarios (*HiroMacro* scripts) to *Monkeyrunner* format. Thus, the collected actions can be played automatically from a script using the *Monkeyrunner* [180] Android tool. In Table 7.2 we provide a brief description of each scenario. Note that the scenarios are generated with the main objective of executing the code segment(s) related to the anti-patterns in the original version, and the refactorings applied in the refactored version, and as a disclaimer, many of them may seem trivial, but fit for the purpose of this preliminary study.
4. **Refactoring of mobile apps**. We use Android Studio and Eclipse refactoring-tool-support for applying the refactorings suggested by ReCon. For some anti-patterns instances we have to refactor them manually as the aforementioned IDEs do not provide tool support. To ensure that a refactored code fragment is executed in the scenario, we first set breakpoints to validate that the debugger stops on it. If this occurs, we build the corresponding apk and check that method invocations to the refactored code appeared in the execution trace. To activate the generation of execution trace file, we use the methods provided in *Android Debug Class* [181], for both original and refactored versions. The trace file contains information about all the methods executed with respect to time, that we use in the next step.
5. **Measurement of energy consumption**. As we mention in Section 2.1, we measure energy consumption of mobile apps using a precise digital oscilloscope *TiePie*

8. https://play.google.com/store/apps/details?id=com.prohiro.macro

Table 7.2 Description and duration (in seconds) of scenarios generated for the studied apps in our preliminary study.

App	Scenario	Duration
blackjacktrainer	Press in {...}, then {settings}, and close app.	14.87
Calculator	Make the operation six times five and close app.	17.94
GLTron	Wait until app is loaded and close app.	33.94
kindmind	Press in first category and close app.	21.37
matrixcalc	Fill matrix with number five, press {Calculate}, and close app.	52.47
monsterhunter	Press in {Weapons}, press in first category, select first weapon, press the {+} button, select the {My Wishlist}, press {Ok}, and close the app.	16.39
mylocation	Press the square button, go back, and close app.	15.59
oddscalculator	Wait until app is loaded and close app.	15.72
prism	Wait until app is loaded and close app.	10.84
quicksnap	Wait until app is loaded and close app.	13.8
SASAbus	Wait until DB is downloaded, press {OK} button, wait until maps are downloaded, and close app.	71.72
scrabble	Wait to load board and close app.	35.83
soundmanager	Go to menu, mute/unmute, and close app.	18.74
speedometer	Wait until app is loaded and close app.	13.99
stk	Wait until app is loaded and content downloaded and close app.	35.1
sudoWars	Wait until app is loaded and close app.	10.76
swjournal	Start a workout, filling the two fields, and close app.	28.87
tapsoffire	Press in {Play}, slide down, press over the green color, press {Play}, {API}, {Medium}, and {Play}; close app.	25.96
vitoshadm	Wait until app is loaded and close app.	14.78
words	Wait until app is loaded and close app.	10.75

Handyscope HS5 which allows us to measure using high frequencies and directly storing the collected results to the personal computer at runtime.

In our experiments each app is run 30 times to get median results and, for each run, the app is uninstalled after its usage and the cache is cleaned. A description of the followed steps is given in Algorithm 9, which has been implemented as a Python script. As it is described, all apps are executed before a new run is started. Thus, we aim to avoid that cache memory on the phone stores information related to the app run that can cause to run faster after some executions. In addition, before the experiments, the screen brightness is set to the minimum value and the phone is set to keep the screen on. In order to avoid any kind of interferences during the measurements, only the essential Android services are run on the phone (for example, we deactivate Wi-Fi if the app does not require it to be correctly executed, etc.).

Our script starts the oscilloscope and the app, which we modify to generate the execution trace. Both are different files where the first time-stamp is zero.

When users launch an app, the app goes through an initialization process running the methods `onCreate`, `onStart`, and `onResume`. In Figure 7.3 we present a simplified flow-chart of the state paths of a single-activity Android app. The app is visible after the `onStart` method is executed and the user can interact with the app after the `onResume` method is executed. We consider that an Android app is completely loaded

after method `onResume` ends. The times reported in Table 3 are the times required to completely load each app and run the corresponding scenario. For all scenarios, the last action of the scenario is to manually close the app, which takes between three and five seconds.

Additionally, the generated execution traces contain, for each method call, global execution times relative to the complete load of apps (whose global time is zero). Based on that we consider the global start time of the method `onCreate` as the instant of time when the execution trace is created once the app is launched.

In order to estimate the existing gap between energy and execution traces we do the following. Once we start the oscilloscope we introduce a timer to measure the time needed to launch an Android app. We consider the difference between this time and the time when the method `onCreate` is executed as the gap between energy and execution traces. For instance, if we consider that an Android app is launched in T seconds and the execution trace is created in instant of time N, the existing gap between the energy and execution trace is calculated as $T - N$. Because for each app's run we know the time required to launch the app and when the method `onCreate` is executed, the gap between traces for each app's run is known.

According to our experiments Android apps are launched in the range of [0.76, 0.92] seconds (average 0.83 seconds = 830000 microseconds) and the method `onCreate` is executed, on average, 0.00009 seconds (90 microseconds) after the app is launched. It means that, in average, the existing gap is (830000-90) = 829010 microseconds. For each app's independent run, energy and execution traces are aligned considering the estimated gap shift.

When the oscilloscope is started it begins to store in memory energy measurements which are written to a *Comma Separated Values* (CSV) file when the scenario associated to the app finishes. Once Algorithm 9 finishes, we have two files for each app and run: the energy trace and the execution trace. Using the existing timestamp in energy traces and the starting and ending time of methods calls in execution traces, energy consumption is calculated for each method called and this information is saved in a new CSV file for each app and run. From these files, we filtered out method names that does not belong to the *namespace* of the app. For example, for *Calculator* app, the main activity is located in the package `com.android2.calculator3`, and we only consider the methods included in this package as they correspond to the source code that we analyze to generate refactoring opportunities.The rationale of removing energy consumption of code that is not inside the package of the app is that we did not detect anti-patterns, neither propose refactoring for those classes. Hence, with the aim of

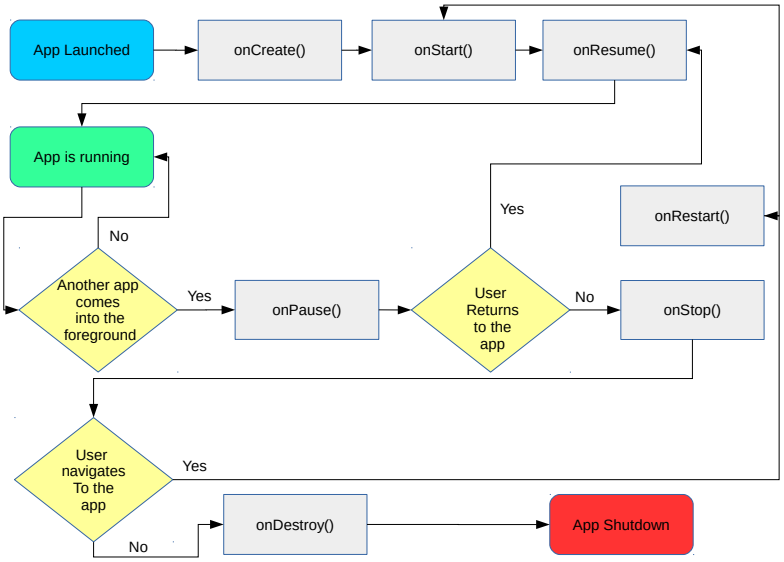

Figure 7.3 Android App flow-chart

removing noise in our measurements (in case that most of an app's energy consumption is on the library or native functions) we focus on the code that contains anti-patterns, to isolate the effect of applying refactoring on energy consumption. Finally, the median and average energy consumption of each app over the 30 runs is calculated.

Algorithm 9: Steps to collect energy consumption.

```
1  forall runs do
2      forall apps do
3          Install app in the phone (using adb).
4          Start oscilloscope using a script from our test PC.
5          Run app (using adb).
6          Play scenario (using Monkeyrunner).
7          Stop oscilloscope.
8          Download execution trace from the phone (using adb).
9          Stop app (using adb).
10         Clean app files in the phone (using adb).
11         Uninstall app (using adb).
12     end
13 end
```

7.2.3 Data Analysis

In the following we describe the dependent and independent variables of this preliminary study, and the statistical procedures used to address each research question. For all statistical tests, we assume a significance level of 5%. In total we collected 864 GB of data from which 391 GB correspond to energy traces, 329 GB to execution traces. The amount of data

generated from computing the energy consumption of methods calls using these traces is 144 GB.

(PQ1): Do anti-patterns influence energy consumption?
For **PQ1**, the *dependent variable* is the energy consumption for each app version (original, refactored). The *independent variable* is the occurrence of any of the anti-patterns studied, and it is true for the original versions of the apps studied, and false for the refactored ones. We statistically compare the energy consumption between the original and refactored design using a non-parametric test, Mann-Whitney U test. Because we do not know beforehand if the energy consumption will be higher in one direction or in the other, we perform a two-tailed test. For estimating the magnitude of the differences of means between original and refactored designs, we use the non-parametric effect size measure Cliff's δ (ES), which indicates the magnitude of the effect size [154] of the treatment on the dependent variable. The effect size is small for $0.147 \leq d < 0.33$, medium for $0.33 \leq d < 0.474$, and large for $d \geq 0.474$ [155].

(PQ2): Do anti-pattern's types influence energy consumption differently?
For **PQ2**, we follow the same methodology as **PQ1**. For each type of anti-pattern, we have three different apps containing an instance of the anti-pattern. We refactor these apps to obtain versions without the anti-pattern. We measure the energy consumption of the original and refactored versions of the apps 30 times to obtain the values of the *dependent variable*. The *independent variable* is the existence of the type of anti-pattern.

7.2.4 Results and Discussion of the Preliminary Study

In Table 7.3 we present the percentage change in median energy consumption after removing one instance of anti-pattern at time, $\gamma(E', E_0)$. This value is calculated using the following expression.

$$\gamma(E', E_0) = \frac{med(E') - med(E_0)}{med(E_0)} \times 100 \tag{7.1}$$

Where the energy consumption of the app after removing an anti-pattern is represented by E', while the energy consumption of the original app is E_0. $med(E)$ is the median of the energy consumption values of the 30 independent runs. Negative values indicate a reduction of energy consumption after refactoring, positive values indicate an increase of energy consumption. In total, we manually correct 24 anti-patterns inside the set of apps that make up our testbed. In seven instances (*i.e.,* 30%) the differences are statistically significant, with Cliff's δ effect sizes ranging from small to large. Specifically, we obtained three apps with large effect size: *speedometer*, *gltron*, and *soundmanager* (two types of anti-patterns); two cases with medium

effect size: *oddscalculator*, *words*; and one with small effect size, *vitoshadm*. Therefore we reject H_{0_1} for these seven apps.

*Overall, our results suggest that different types of anti-patterns may impact the energy consumption of apps differently. Our next research question (i.e., **PQ2**) investigates this hypothesis in more details.*

To answer **PQ2**, on the impact of different types of anti-patterns on energy consumption, we present in Figure 7.4 the percentage change of the energy consumption after removing each type of anti-pattern studied. For the instances where the results are statistically significant ($p - value < 0.05$) we add an "$*$" symbol, the exact value and ES is shown in Table 7.3.

Regarding object-oriented (OO) anti-patterns, on top of Figure 7.4, we observe that removing *lazy class* reduces energy consumption in *blackJacktrainer*. This trend holds for *tapsoffire* and *soundmanager* respectively, with the latter one having statistically significance and magnitude of the difference (*i.e., ES*) is large. In the case of *Refused Bequest*, two out of three apps show that removing the anti-pattern saves energy, and the difference is statistically significant for *vitoshadm*. For the *Blob* anti-pattern, all refactored versions report a decrease in energy consumption, though the differences are not statistically significant.

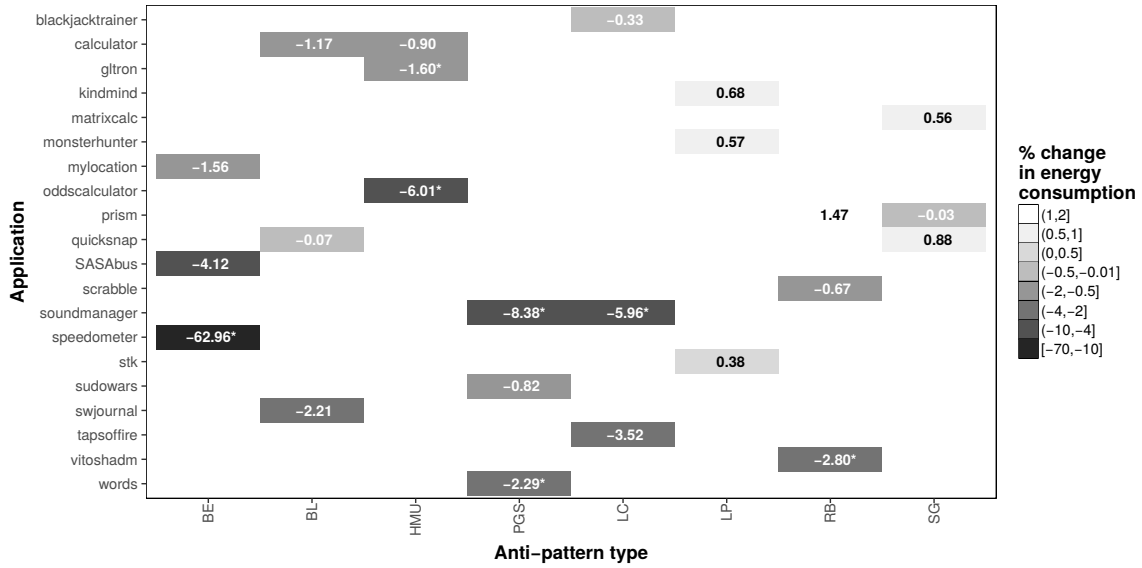

Figure 7.4 Percentage change in median energy consumption when removing different types of anti-patterns

Concerning *Long Parameter list* (LP), and *Speculative Generality* (SG), both report a negative impact on energy consumption after refactoring. While for LP, all the apps point toward

Table 7.3 Percentage change in median energy consumption of apps after removing one instance of anti-pattern at time, Mann—Whitney U Test and Cliff's δ Effect Size (ES).

App	$\gamma(E', E_0)$	$p - value$	ES Magnitude
blackjacktrainer	-0.63	0.2560	-0.15 small
calculator	-1.17	0.1191	-0.25 small
calculator	-0.90	0.4280	-0.10 negligible
gltron	**-1.60**	**2.08E-05**	**-0.70 large**
kindmind	0.68	0.2988	0.16 small
matrixcalc	0.56	0.4898	0.09 negligible
monsterhunter	0.50	0.5602	-0.07 negligible
mylocation	-1.56	0.5699	-0.03 negligible
oddscalculator	**-6.01**	**0.0221**	**-0.34 medium**
prism	1.50	0.0919	0.17 small
prism	-0.03	0.7151	0.03 negligible
quicksnap	-0.07	0.9515	-0.03 negligible
quicksnap	0.89	0.4898	0.04 negligible
SASAbus	-4.12	0.2286	-0.13 negligible
scrabble	-0.67	0.9838	-0.04 negligible
soundmanager	**-8.38**	**0.0001**	**-0.63 large**
soundmanager	**-5.96**	**0.0005**	**-0.53 large**
speedometer	**-62.96**	**3.73E-09**	**-0.97 large**
stk	0.38	0.5028	0.02 negligible
sudowars	-0.82	0.6408	0.04 negligible
swjournal	-2.21	0.2286	-0.23 small
tapsoffire	-3.52	0.3599	-0.22 small
vitoshadm	**-2.80**	**0.0345**	**-0.29 small**
words	**-2.29**	**0.0005**	**-0.44 medium**

more energy consumption, in the case of SG, the energy consumption is increased in two out of three apps after refactoring. We explain the result obtained for LP by the fact that the creation of a new object (*i.e.,* the parameter object that contains the long list of parameters) adds to some extent more memory usage. For SG we do not have a plausible explanation for this trend. For both anti-patterns, the obtained differences in energy consumption is not statistically significant, hence we cannot conclude that these two anti-patterns always increase or decrease energy consumption.

Regarding Android anti-patterns. For *HashMap usage* (HMU) and *Private getters and setters* (PGS), we obtained statistically significant results for two apps. For *Binding Resources too early* (BE), the result is statistically significant for one app. In all cases, apps that contained these anti-patterns consumed more energy than their refactored versions that did not contained the anti-patterns. This finding is consistent with the recommendation of previous works (*i.e.,* [170, 51]) that advise to remove HMU, PGS, and BE from Android apps, because of their negative effects on energy consumption. Note that the amount of energy saved is influenced by the context in which the application runs. For example, *SASAbus*, which is a bus schedule app, downloads the latest bus schedule at start, consuming a considerable amount of data and energy. As a result, the gain in energy for relocating the call method that starts the GPS device is negligible in comparison to the overall scenario. *Mylocation* is a simpler app, that only provides the coordinated position of mobile user. This app

optimizes the use of the GPS device by disabling several parameters, like altitude and speed. It also sets the precision to *coarse* (approximate location [182], and the power requirements to *low*. For this app, we observe a consistent improvement when the anti-pattern is removed, but in a small amount. On the other hand, we have *speedometer*, which is a simple app as well, that measures user's speed, but using *high precision mode*. *High precision mode* uses GPS and internet data at the same time to estimate location with high accuracy. In *speedometer*, we observe a high reduction in energy consumption when the anti-pattern is corrected, in comparison with the previous two apps.

In summary, there is evidence to show that removing ***Binding resources too early***, ***Private getters and setters***, ***Refused Bequest***, *and* ***Lazy class*** *anti-patterns can improve energy efficiency in some cases. We do not find any statistically significant cases were removing an anti-pattern increases energy consumption. Removing* ***Blob***, ***Long Parameter List***, *and* ***Speculative Generality*** *anti-patterns does not produce a statistically significant increase or decrease.*

The impact of different types of anti-patterns on the energy consumption of mobile apps is not the same. Hence, we reject H_{02}.

7.3 Energy-Aware Automated Refactoring of Mobile Apps

After determining in Section 7.2 that the occurrence of anti-patterns impacts the energy consumption of mobile apps, we leverage this knowledge to propose an approach to improve the design quality of mobile apps, while controlling energy consumption. Our proposed approach is based on a search-based process where we generate refactoring sequences to improve the design of an app. This process involves evaluating several sequences of refactoring iteratively and the resultant design in terms of design quality and energy consumption. Measuring in real-time the energy consumption of a refactoring sequence can be prohibitive, because it requires to apply each refactoring element of the sequence in the code, compile it, generate the binary code (APK) and download it into the phone; all of these steps for each time the search-based process requires to evaluate a solution. That is why we define a strategy to estimate the impact of each refactoring operation on energy consumption, based on the results obtained in our preliminary study (Section 7.2) and without measuring during the search process. The strategy consists of the following steps:

1. We compute the energy consumption of an app using the following formulation:

$$EC(a) = \sum_{m \in M} EC(a_m) \tag{7.2}$$

   Where M is the set of methods in a.

2. We prepare two versions of the same app with and without one instance of an anti-pattern type, and we call them a^{ORI}, and a^k. To isolate possible aggregation effects, we remove only one instance of anti-pattern using the same refactoring operations. For example, if we want to remove a Lazy class, we apply inline class to the class that contained that anti-pattern.

3. The energy consumption coefficient of a refactoring applied to remove an anti-pattern of type k, in app a is calculated using the following expression.

$$\delta EC(a^k) = \frac{med(EC(a^k)) - med(EC(a^{ORI}))}{med(EC(a^{ORI}))} \tag{7.3}$$

   Where $med(.)$ is the median value of the 30 independent runs for $EC(a^k)$ and $EC(a^{ORI})$. If the value of $\delta EC(a^k)$ is negative, it means that the refactored version consumes less energy. On the contrary, if this value is positive, it means that the refactored version consumes more energy than the original version.

4. To determine a global refactoring energy coefficient $\delta EC(k)$, we take three apps from our testbed for each type of anti-pattern k. $\delta EC(k)$ is calculated using the following expression.

$$\delta EC(k) = med(\delta EC(a^k)); \forall a^k \in A^k \tag{7.4}$$

   Where A^k is the set of apps that were refactored to remove a single instance of anti-pattern type k.

In the following, we describe the key components of our proposed approach EARMO, for the correction of anti-patterns while controlling for energy consumption.

7.3.1 EARMO overview

EARMO is comprised of four steps, depicted in Algorithm 10. The first step consists in estimating the energy consumption of an app, running a defined scenario. In the second step, we build an abstract representation of the mobile app's design, *i.e., code meta-model*. In the third step, the code meta-model is visited to search for anti-pattern occurrences. Once the list of anti-patterns is generated, the proposed approach determines a set of refactoring

opportunities based on a series of pre- and post-conditions extracted from the anti-patterns literature [3, 47, 170, 51]. In the final step, a multiobjective search-based approach is run to find the best sequence of refactorings that can be legally applied to the code, from the refactoring opportunities list generated in the previous step. The solutions produced by the proposed approach meet two conflicting objectives: 1) remove a maximum number of anti-patterns in the system, and 2) improve the energy consumption of the code design. In the following, we describe in detail each of these steps.

7.3.2 Step 1: Energy consumption estimation

This step requires to provide (1) the energy consumption of the app (E_0). Developers can measure E_0 by setting an energy estimation environment similar to the one presented in Section 7.2, or using a dedicated hardware-based energy measurement tool like GreenMiner [117]. (2) The coefficient $\delta EC(k)$ of each refactoring type analyzed. We derive $\delta EC(k)$ values for each refactoring type based on the results of the preliminary study. EARMO uses this information in the last step to evaluate the energy consumption of a candidate refactoring solution during the search-based process.

7.3.3 Step 2: Code meta-model generation

In this step we follow the same procedure that in previous chapters to generate a light-weight representation (a meta-model) of a mobile app, using static code analysis techniques, with the aim of evolving the current design into an improved version in terms of design quality and energy consumption.

7.3.4 Step 3: Code meta-model assessment

In this step we assess the quality of the code-meta model by (1) identifying anti-patterns in its entities, and (2) determining refactoring operations to correct them. For example, the correction of *Binding resources too early* anti-pattern can be divided in the following steps: detect classes with code statements that initialize energy-intensive components, *e.g.,* GPS or Wi-Fi, before the user or the app can interact with them; move the conflicting statements from its current position to a more appropriate method, *e.g.,* when the app interacts with the user, preventing an unnecessary waste of energy.

Algorithm 10: EARMO Approach

```
   Input  : App to refactor (App), scenario (scen)
   Output: Non-dominated refactoring sequences
1  Pseudocode EARMO(Mobile app)
2      E_0 = Energy consumption measurement (App, scen)
       /* We estimate the energy consumption of an app to estimate the energy improvement during our
          search-based approach                                                                  */
3      AM=Code meta-model generation (App)
       /* From the source code generate a light-weight representation of the code               */
4      RA=Code meta-model assessment (AM)
       /* 1.  Detect anti-patterns in the system and generate a map of classes that contain anti-patterns
          */
       /* 2.  Generate a list of refactoring operations to correct anti-patterns               */
5      Generation of optimal set of refactoring sequences (AM, RA, E_0)
       /* This is a generic template of the EARMO algorithm that finds the optimal set of refactoring
          sequences                                                                             */
6  Procedure Generation of an optimal set of refactoring sequences(AM, RA, E_0)
7      P_0 = GenerateInitialPopulation(RA)
8      X_0 = ∅
       /* X is the set of non-dominated solutions                                              */
       /* Evaluation of P_0                                                                    */
9      for all S_i ∈ P_0 do
           /* S_i is a refactoring sequence                                                    */
10         AM' = clone(AM)
11         apply_refactorings(AM', S_i)
12         compute_Design_Quality(AM', S_i)
13         compute_Energy_Consumption(AM', S_i, E_0)
14     end for
       /* Update the set of non-dominated solutions found in this first sampling              */
15     X_0 = Update(X_0, P_0)
16     t = 0
17     while not StoppingCriterion do
18         t = t + 1
19         P_t = Variation_Operators(P_{t-1})
20         for all S_i ∈ P_t do
21             AM' = clone(AM)
22             apply_refactorings(AM', S_i)
23             compute_Design_Quality(AM', S_i)
24             estimate_Energy_Consumption(AM', S_i, E_0)
25         end for
26         X_t = Update(X_t, P_t)
27     end while
28     best_solution = X_t
29     return best_solutions
```

7.3.5 Step 4: Generation of optimal set of refactoring sequences

In this final step, we aim to find different refactoring sequences that remove a maximum number of anti-patterns, while improving the energy consumption of mobile apps. Hence, we use EMO algorithms to obtain from all the set of possible refactoring combinations, the optimal solutions, *i.e.,* the ones that are not dominated. In the following, we describe the key elements of our MO optimization process.

Solution representation

We represent a refactoring solution as a vector, where each element represents a refactoring operation (RO) to be applied, *e.g.,* a subset of refactoring candidates obtained by EARMO (*cf.* Sections 4.4 and 6.4).

Selection operator

The selection operator controls the number of copies of an individual (solution) in the next generations, according to its quality (fitness). Examples of selection operators are tournament selection or fitness proportionate selection [183].

Variation Operators

The variation operators allow metaheuristics to transform a candidate solution so that it can be moved through the decision space in the search of the most attractive solutions, and to escape from local optima. In EMO algorithms, we often find two main variation operators: crossover and mutation. We implement the same variation operators used in Sections 4.4 and 6.4.

Fitness functions

We use two fitness functions to evaluate the quality and the energy consumption of the refactoring solutions. The function to evaluate the quality of the design is defined in Equation (4.1) [36].

To evaluate the energy consumption of an app (expressed in Joules) after refactoring, we define the following formulation: let E_0 be the estimated energy consumption of an app a, r_i a refactoring operation type in a sequence $S = (r_1, \ldots, r_n)$. We estimate the energy consumption $EC(a)$ of the app resulting from the application of the refactoring sequence S to the app a as follows: $EC(a) = E_0 + \sum_{i=1}^{n} E_0 \times \delta EC(r_i)$, where $\delta EC(r_i)$ is the energy coefficient value of the refactoring operation r_i. We aim to minimize the value of EC during the search process. In Algorithm 10, we present a generic pseudocode for the EMO algorithms used by our approach (lines 6-29). The process starts by generating an initial population of refactoring sequences from the code meta-model assessment step. Next, it applies each refactoring sequence in the code meta-model and measures the design quality (number of anti-patterns) and the energy saved by applying the refactorings included in the sequence (lines 11-13). The next step is to extract the non-dominated solutions (line 15). From line 20

to 25, the main loop of the metaheuristic process is executed. The goal is to evolve the initial population, using the variation operators described before, to converge to the Pareto optimal front. The stopping criterion, which is defined by the software maintainer, is a fixed number of evaluations. Finally, in lines 28-29, the optimal refactoring sequences are retrieved.

7.4 Evaluation of EARMO

In this section, we evaluate the effectiveness of EARMO at improving the design quality of mobile apps while optimizing energy consumption. The *quality focus* is the improvement of the design quality and energy consumption of mobile apps, through search-based refactoring. The *perspective* is that of researchers interested in developing automated refactoring tools for mobile apps, and practitioners interested in improving the design quality of their apps while controlling for energy consumption. The *context* consists of the 20 Android apps studied in Section 7.2, and three MO metaheuristics (MOCell, NSGA-II, and SPEA2). We instantiate our generic EARMO approach using the three multiobjective metaheuristics, described in Section 2.3.4.

As in previous chapters, the code meta-model is generated using *Ptidej Tool Suite* [156].

The anti-patterns considered in the evaluation of EARMO are the ones described in Section 7.2.1. In the following, we describe the strategies implemented in EARMO to correct anti-patterns in Android. Most of the OO anti-patterns strategies were already presented in previous chapters (Section 4.4, Section 5.4), except for Refused Bequest anti-pattern which we describe below.

Replace inheritance with delegation (RIWD). This refactoring is applied when we find a class that inherits a few methods from its parent class (Refused bequest anti-pattern). To apply this refactoring, we create a field of the parent class, and for each method that the child use, we delegate to the field (parent class type), replacing the inheritance by an association. We present an example of this refactoring in Figure 7.5.

Move resource request to visible method (MRM). To determine the appropriate method to initialize a high-power-consumption component, it is necessary to understand the vendor platform. In our case, we illustrate the refactoring based on Android, but the approach can be extended to other operating systems. As previously discussed in Section 7.2.2, when users launch an app, the app goes through an initialization process that ends after the `onStart` method is executed (the app is visible). After the `onResume` method is executed, the user can interact with the app, but not before that. Hence, switching on a high-power-consumption component in the body of `OnCreate` is a terrible idea, in terms of energy

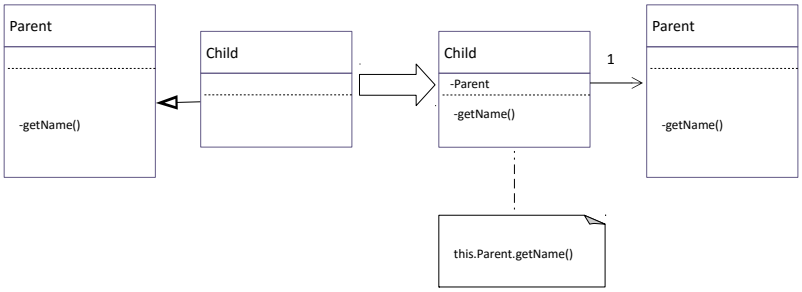

Figure 7.5 An example of applying RIWD in a class. Original class diagram on the left, and refactored class diagram on the right.

consumption. Consequently, the refactoring consists in moving any hardware resource request from `onCreate` to `OnResume`.

Inline private getters and setters (IGS). The use of private getters and setters is expensive in Android mobile devices in comparison to direct field access. Hence, we inline the getters and setters, and access the private field directly. An illustrative example is provided in Figure 7.6.

Replace HashMap with array map (RHA). ArrayMap is a light-weight-memory mapping data structure included since Android API 19. The refactoring consists in replacing the import of `java.util.HashMap` with `android.Util.Arraymap`, and any *HashMap* reference with *ArrayMap*. *ArrayMap* is compatible with the standard Java container APIs (*e.g.,* iterators, etc), and not further changes are required for this refactoring, as depicted in Figure 7.7.

7.4.1 Descriptive statistics of the studied Apps

Table 7.4 presents relevant information about anti-patterns contained in the studied apps. The second column contains the number of classes (NOC), and the following columns contain the occurrences of OO anti-patterns (3-7) and android anti-patterns (8-10). The last two rows summarize the median and total values for each column.

7.4.2 Research Questions

To better answer the central question of this chapter, we formulate the following research questions:

(RQ1) To what extent EARMO can remove anti-patterns while controlling for energy consumption?

This research question aims to assess the effectiveness of EARMO at improving design quality,

```
1
2  private SplashView splashView;
3
4  private SplashView getSplashView() {
5          return splashView;
6      }
7  //This setter is not even used!
8   private void setSplashView(SplashView
       splashView) {
9          this.splashView = splashView;
10     }
11
12   public void initialize() {
13         final boolean firstLaunch =
       isFirstLaunch();
14
15         if (firstLaunch) {
16             getSplashView().showLoading();
17         }
18         ...
19         getSplashView().renderImportError();
20         ...
21         getSplashView().
       renderSplashScreenEnded();
22             }
23        ...
24
25         getSplashView().renderFancyAnimation
       ();
26     }
```

```
1  private SplashView splashView;
2
3  // We inline private getters and setters
4   public void initialize() {
5          final boolean firstLaunch =
       isFirstLaunch();
6
7          if (firstLaunch) {
8             splashView.showLoading();
9          }
10         ...
11         splashView.renderImportError();
12         ...
13         splashView.renderSplashScreenEnded();
14             }
15        ...
16
17         splashView.renderFancyAnimation();
18     }
```

Figure 7.6 Example of *inline private getters and setters* refactoring. Original code on the left, and refactored code on the right.

```
1  package com.glTron.Sound;
2
3  import java.util.HashMap;
4  ...
5  public class SoundManager {
6
7          private static HashMap<Integer,
       Integer> mSoundPoolMap;
8          ...
9          public static void initSounds(Context
        theContext)
10         {
11                 ...
12                 mSoundPoolMap = new HashMap<
       Integer, Integer>();
13                 ...
14         }
15 }
```

```
1  package com.glTron.Sound;
2
3  import android.util.ArrayMap;
4  ...
5  public class SoundManager {
6
7          private static ArrayMap<Integer,
       Integer> mSoundPoolMap;
8          ...
9          public static void initSounds(Context
        theContext)
10         {
11                 ...
12                 mSoundPoolMap = new ArrayMap<
       Integer, Integer>();
13                 ...
14         }
15 }
```

Figure 7.7 Example of replacing HashMap with ArrayMap refactoring. Original code on the left, and refactored code on the right.

Table 7.4 Descriptive statistics showing anti-pattern occurrences in the studied apps.

		O.O. AP					Android AP		
App	NOC	BL	LC	LP	RB	SG	BE	HMU	PGS
Calculator	43	2	3	0	8	5	0	14	0
BlackJackTrainer	13	1	3	0	0	0	0	0	0
GlTron	26	1	3	5	0	0	0	6	1
Kindmind	36	4	0	2	4	0	0	5	0
MatrixCalculator	16	1	0	2	1	2	0	0	0
MonsterHunter	194	11	1	2	32	0	0	3	0
mylocation	9	0	1	0	0	0	1	0	0
OddsCalculator	10	0	6	0	0	0	0	1	0
Prism	17	0	3	0	1	2	0	1	0
Quicksnap	76	3	6	1	1	1	0	10	4
SASAbus	49	0	1	0	0	1	2	7	0
Scrabble	9	0	4	0	0	1	0	2	0
SoundManager	23	0	9	1	0	0	0	6	2
SpeedoMeter	3	0	1	0	0	0	1	0	0
STK	25	0	1	1	0	0	0	4	0
Sudowars	110	26	2	3	21	6	0	9	1
Swjournal	19	0	1	1	0	0	0	0	0
TapsofFire	90	4	5	7	4	1	0	19	1
Vitoshadm	9	0	0	0	1	1	0	0	0
Words	136	10	4	12	6	1	0	15	0
Median	24	1	3	1	1	1	0	4	0
Total	913	63	54	37	79	21	4	102	9

while reducing energy consumption.

(RQ2) What is the precision of the energy improvement reported by EARMO? This research question aims to examine if the estimated energy improvements reported by EARMO reflect real measurements.

(RQ3) To what extent is design quality improved by EARMO according to an external quality model?
While the number of anti-patterns in a system serves as a good estimation of design quality, there are other quality attributes such as those defined by the QMOOD quality model [40] that are also relevant for software maintainers, *e.g.,* reusability, understandability and extendibility. This research question aims to assess the impact of the application of EARMO on these high-level design quality attributes as we did in previuous chapters.

(RQ4) Can EARMO generate useful refactoring solutions for mobile developers? This research question aims to assess the quality of the refactoring recommendations made by EARMO from the point of view of developers. We aim to determine the kind of recommendation that developers find useful and understand why they may chose to discard certain recommendations.

7.4.3 Evaluation Method

In the following, we describe the approach followed to answer **RQ1**, **RQ2**, **RQ3** and **RQ4**.

For **RQ1**, we measure two *dependent variables* to evaluate the effectiveness of EARMO at removing anti-patterns in mobile apps while controlling their energy consumption:

— Design Improvement (DI). DI represents the delta of anti-patterns occurrences between the refactored (a') and the original app (a) and it was introduced in Chapter 5, Equation (5.2).

— Estimated energy consumption improvement (EI). EI is computed using the following formulation.

$$EI(a) = \frac{EC(a') - EC(a)}{EC(a)} \times 100. \tag{7.5}$$

Where $EC(a)$ is the energy consumption of an app a and $EC(a) \geq 0$. EI captures the improvement in the energy consumption of an app a after refactoring operation(s). The sign of EI expresses an increment (+)/decrement (-) and the value represents the amount in percentage. High negative values are desired.

The *independent variables* are the three selected *EMO* metaheuristics, *i.e.,* MOCell, NSGA-II, and SPEA2. We choose them because they are well-known evolutionary techniques that have been successfully applied to solve optimization problems, including refactoring [54, 36]. We implement all the metaheuristics used in this study using the jMetal Framework [164], which is a popular framework for solving optimization problems.

The performance of a metaheuristic can be affected by the correct selection of its parameters. The configurable settings of the search-based techniques used in this chapter correspond to stopping criterion, population size, and the probability of the variation operators. We use number of evaluations as the stopping criteria. As the maximum number of evaluations increase, the algorithm obtains better quality results on average. The increase in quality is usually very fast when the maximum number of evaluation is low. That is, the slope of the curve quality versus maximum number of evaluations is high at the very beginning of the search. But this slope tends to decrease as the search progresses. Our criterion to decide the maximum number of evaluations is to select a value for which this slope is low enough. In our case *low enough* is when we observe that no more anti-patterns are removed after that number of evaluations. We empirically tried different number of evaluations in the range of 1000 to 5000 and found 2500 to be the best value.

For selection operator we use the same operator defined by Deb et al. [63] for NSGA-II, and *binary tournament* for the other EAs, which are the default operators used in *JMetal* for these algorithms.

For population size, we use a default value of 100 individuals; and for the probability of applying a variation operator we selected the parameters using a factorial design in the

following way: we tested 16 combinations of mutation probability $p_m = (0.2, 0.5, 0.8, 1)$, and crossover probability $p_c = (0.2, 0.5, 0.8, 1)$, and obtained the best results with the pair $(0.8, 0.8)$.

The initial size of the refactoring sequence is set to 50% of the size of the total number of refactoring opportunities, based on our previous findings in Section 4.4.3.

With respect to energy estimation, we show in Table 7.5 the energy consumption coefficient $\delta EC(k)$ for each refactoring type, that we use in our experiment. These coefficients were obtained from the formulation described in Section 7.3.

Note that for the *move method* refactoring, we did not use the energy consumption measured for the correction of *Blob*, as correcting a *Blob* requires many *move methods* to be applied. Hence, we measured the same apps used for *Blob* (*i.e., Swjournal*, *Quicksnap* and *Calculator*) with and without moving exactly one method to estimate the effect of this refactoring. The results, which are not statistically significant, show a decrement in energy consumption.

In order to determine which one of our three *EMO* algorithms (*i.e.,* MOCell, NSGA-II, and SPEA2) achieves the best performance, we compute two different performance indicators: *Hypervolume (HV) [166]* and *SPREAD [63].*

We also perform Whitney U Test test pair-wise comparisons between the three algorithms to validate the results obtained for these two performance indicators.

For **RQ2**, we perform an energy consumption validation experiment to evaluate the accuracy of EARMO using our measurement setup described in Section 7.1.1. This is important to observe how close is the estimated energy improvement (*i.e.,* EI) compared to the real measurements. For each selected app we compute refactoring recommendations using EARMO and implement the refactorings in the source code of the app. Then, we measure the energy consumption of the original and refactored versions of the apps using a typical usage scenario, and compute the difference between the obtained values. We compare the obtained result with EI.

For **RQ3**, we use the Quality Model for Object-Oriented Design (QMOOD) [40] to measure the *impact* of the refactoring sequences proposed by EARMO, on the design quality of the apps, as we did in previous chapters.

We compute the quality gain (QG) for each quality attribute using the following formulation.

$$QG(A_y) = \frac{A_y(a') - A_y(a)}{|A_y(a)|} \times 100 \tag{7.6}$$

Table 7.5 Deltas of energy consumption by refactoring type.

Refactoring Type	δEC (ratio)
Collapse hierarchy	0.0056
Inline class	-0.0315
Inline private getters and setters	-0.0237
Introduce parameter object	0.0047
Move method	-0.0020
Move resource request to visible method	-0.0412
Replace HashMap with ArrayMap	-0.0160
Replace Inheritance with delegation	-0.0067

Where $A_y(a)$ is the quality attribute y measurement for an app a, and a' is the refactored version of the app a. The sign expresses an increment (+)/decrement (-) and the value represents the improvement amount in percentage. Note that since the calculation of QMOOD attributes can lead to negative values in the original design, it is necessary to compute the absolute value of the divisor.

For **RQ4**, we conducted a qualitative study with the developers of our studied apps. For each app, we randomly selected some refactoring operations from the refactoring sequence recommended by EARMO, and submitted them to the developers of the app for approval or rejection. We choose three examples for each type of refactoring and for each app.

To measure developers' taking of the refactorings proposed, we compute for each app the *acceptance ratio*, which is the number of refactorings accepted by developers divided by the total number of refactorings submitted to the developers of the app. We also compute the *overall acceptance ratio* for each type of anti-pattern, considering all the apps together.

7.4.4 Results of the Evaluation

In this section we present the answers to our four research questions that aim to evaluate EARMO.

RQ1: To what extent EARMO can remove anti-patterns while controlling for energy consumption?

Because the metaheuristic techniques employed in this work are non-deterministic, the results might vary between different executions. Hence, we run each metaheuristic 30 times, for each studied app, to provide statistical significance. As a result, we obtain three reference Pareto front approximations (one per algorithm) for each app. From these fronts, we extract a global reference front that combines the best results of each metaheuristic for each app and, after that, dominated solutions are removed.

In Figure 7.8, we present the distribution of DI and EI metric values, for each solution in the Pareto reference front. Figure 7.8 highlights a median correction of 84% of anti-patterns and

estimated energy consumption improvement of 48%. To provide insights on the performance of EARMO, we present, in Table 7.6, the number of non-dominated solutions found for each app (column 2), the minimum and maximum values with respect to DI (columns 3-4), and EI metrics (columns 5-6). The number of non-dominated solutions are the number of refactorings sequences that achieved a compromise in terms of design quality and energy consumption. Table 7.6 reports 2.5 solutions on average with a maximum of eight solutions (*words*). Thus, for the studied apps, a software maintainer has approximately three different solutions to choose to improve the design of an app.

In general, we observe that the results for DI and EI metrics are satisfactory, and we find that in nine apps EARMO reach 100% of anti-patterns correction with a maximum EI of 89%. With respect to the variability between apps with more than one solution, for EI metrics the difference between the maximum and minimum value is small, and for DI too, except for the apps with more than two solutions (*i.e., Calculator* and *Words*). We observe that more than 65% of the apps contain more than one solution. To have an insight on those apps, we present in Figure 7.9 the Pareto Front (PF) for each app, where each point represents a solution with their corresponding values, DQ (x-axis) and EC (y-axis). The most attractive solutions are located in the bottom right of the plot.

According to the concept of dominance, every Pareto point is an equally acceptable solution of the multiobjective optimization problem [184], but developers might show preference over the ones that favors the metric they want to prioritize. They could select the refactorings that improve more the energy consumption (*e.g.,* they can chose to correct more Android anti-patterns), or apply more OO refactorings to improve the maintainability of their code. Other developers might be more conservative and select solutions located in the middle of these two objectives. Developers have the last word, and EARMO supports them by providing different alternatives.

Impact of refactoring sequences with respect to the type of anti-patterns. The anti-patterns analyzed in this study affect different quality metrics, and their definitions can be opposed, *e.g., Blob* and *Lazy class*. In Table 7.7, we present the median values of the DI metric for the non-dominated solutions of each type of anti-pattern. The results fall into two different categories.

— **Medium**. *Speculative generality* and *Blob* anti-patterns have median correction rates of 50% and 67%, respectively, while *Long parameter list* reached 75%.

— **High**. For the rest of the studied anti-patterns, the median correction rate is 100%, including the three Android anti-patterns studied and two OO anti-patterns (*i.e., Refused bequest, Lazy class*)

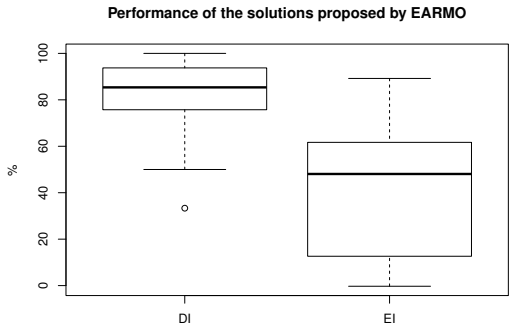

Figure 7.8 Distribution of anti-patterns and energy consumption reduction in the studied apps.

Table 7.6 Minimum and maximum values (%) of DI and EI obtained for each app after applying EARMO.

	Solutions	DI		EI	
App		Min.	Max.	Min.	Max.
blackJacktrainer	1	-75	-75	-6.14	-6.14
calculator	5	-75	-93.75	-48.07	-53.55
gltron	2	-93.75	-100	-25.85	-26.32
kindmind	3	-80	-93.33	-18.42	-18.76
matrixcalculator	3	-33.33	-66.67	0.28	-0.67
monsterhunter	2	-81.63	-83.67	-43.95	-44.42
mylocation	1	-100	-100	-2.05	-2.05
oddscalculator	1	-100	-100	-14.64	-14.64
prism	2	-85.71	-100	-7.94	-9.18
quicksnap	2	-92.31	-96.15	-83.65	-84.88
SASAbus	1	-81.82	-81.82	-27.09	-27.09
scrabble	2	-85.71	-100	-12.36	-12.92
soundmanager	2	-94.44	-100	-35.36	-35.83
speedometer	1	-100	-100	-6.17	-6.17
stk	2	-83.33	-100	-11.05	-11.53
sudowars	8	-60.29	-76.47	-48.77	-63.93
swjournal	1	-100	-100	-5.67	-5.67
tapsoffire	3	-82.93	-87.8	-88.26	-89.21
vitoshadm	1	-100	-100	-3.57	-3.57
words	8	-75	-91.67	-56.83	-63.37

We conclude that including energy-consumption as a separate objective when applying automatic refactoring can reduce the energy consumption of a mobile app, without impacting the anti-patterns correction performance.

Performance of the metaheuristics employed. As mentioned before, EARMO makes use of *EMO* techniques to find optimal refactoring sequences. Therefore, the results can vary from one technique to another. A software maintainer might be interested in a technique that provides the best results in terms of diversity of the solutions, and convergence of the algorithm employed. In the MO research community, the Hypervolume (HV) [166] is a quality indicator often used for this purpose, and higher values of this metric are desirable.

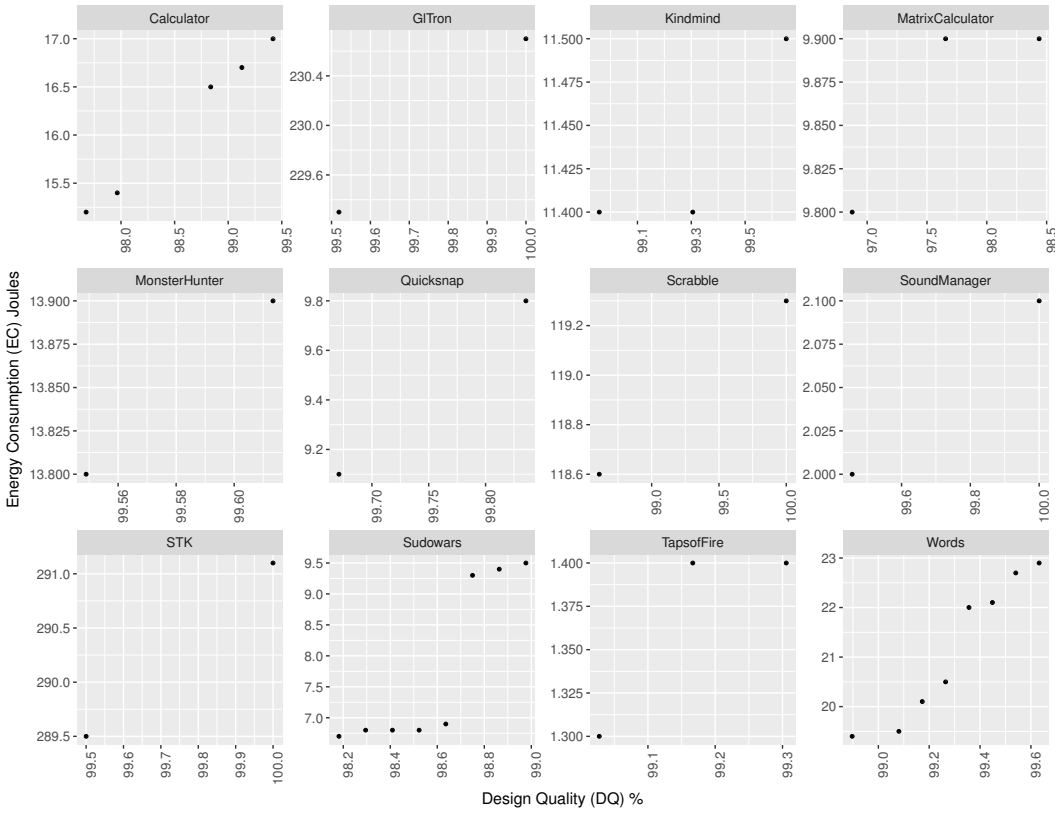

Figure 7.9 Pareto front of apps with more than one non-dominated solution.

Table 7.7 Median values of anti-patterns corrected by type (%).

	O.O. anti-patterns					Android anti-patterns		
App	BL	LC	LP	RB	SG	BE	HMU	PGS
blackjacktrainer	0	-100	NA	NA	NA	NA	NA	NA
calculator	-100	-100	NA	-75	-60	NA	NA	-100
gltron	-100	-100	-90	NA	NA	NA	-100	-100
kindmind	-100	NA	-50	-100	NA	NA	NA	-100
matrixcalculator	0	NA	-50	-100	-50	NA	NA	NA
monsterhunter	-27.27	-100	-75	-100	NA	NA	NA	-100
mylocation	NA	-100	NA	NA	NA	-100	NA	NA
oddscalculator	NA	-100	NA	NA	NA	NA	NA	-100
prism	NA	-100	NA	-100	-75	NA	NA	-100
quicksnap	-66.67	-100	-100	-100	-50	NA	-100	-100
SASAbus	NA	-100	NA	NA	0	-100	NA	-100
scrabble	NA	-100	NA	NA	-50	NA	NA	-100
soundmanager	NA	-100	-50	NA	NA	NA	-100	-100
speedometer	NA	-100	NA	NA	NA	-100	NA	NA
stk	NA	-100	-50	NA	NA	NA	NA	-100
sudowars	-59.62	-100	-66.67	-80.95	-66.67	NA	-100	-94.44
swjournal	NA	-100	-100	NA	NA	NA	NA	NA
tapsoffire	-75	-40	-85.71	-100	0	NA	-100	-100
vitoshadm	NA	NA	NA	-100	-100	NA	NA	NA
words	-85	-100	-91.67	-33.33	50	NA	NA	-100

In Table 7.8 we present the median and interquartile range (IQR) of the *HV* indicator for each metaheuristic and for each app with more than one solution. A special notation has been used in this table: a dark gray colored background denotes the best technique while lighter gray represents the second-best performing technique. For the apps with more than two solutions we observe a draw in *Matrixcalculator*, while MOCell outperforms the other algorithms in two apps. SPEA2 outperforms the rest in *Sudowars*, and gets second best in two more apps. NSGA-II obtains second-best in *Sudowars*. In the cases where the metaheuristics cannot find more than one optimal solution, the value of *HV* is zero. Hence, the outperforming technique according to this quality indicator remains unknown.

Another quality indicator often used is the *Spread* [63]. It measures the distribution of solutions into a given front. Low values close to zero are desirable as they indicate that the solutions are uniformly distributed. In Table 7.9 we present the median and IQR results of the Spread indicator. We observe that MOCell outperforms the other techniques in 92% (12 apps) of cases, while *soundmanager* reports the same value for the three *EMO*s. SPEA2 gets the second best in 69% (nine apps), and NSGAII only in 8% (three apps).

To validate the results obtained by the *HV* and the *Spread* indicators, we perform pairwise comparisons between the three metaheuristics studied, using Whitney U Test, with a confidence level of 95%. The results of these tests are summarized in Table 7.10. We use the same notation and symbols introduced in chapter 6.

Concerning *HV* indicator, only one app (*sudowars*) was statistically significant in the pair MOCell-NSGAII favoring the former one. So we can conclude that in general the performance of the three algorithms is similar. With respect to the *Spread* indicator, MOCell outperforms SPEA2 in seven apps, and NSGA-II in 10. In the pair NSGA-II-SPEA2, there is one app (*Matrixcalculator*) where NSGA-II outperforms SPEA2. Hence, the solutions obtained by MOCell are better spread through the entire Pareto front than the other algorithms. Regarding execution time, we did not observed a significant difference between the execution time of the studied metaheuristics. According to the Whitney U Test test, MOCell is the best performing technique with respect to solution diversity, while regarding HV the performance of the three *EMO* algorithms is similar. Developers and software maintainers should consider using MOCell when applying EARMO.

RQ2: What is the precision of the energy improvement reported by EARMO?
The output of EARMO is a sequence of refactorings that balances anti-pattern correction and energy consumption. Developers select from the Pareto front, the solutions that best fits their needs. To validate the estimations of EARMO, we play the role of a software maintainer who wants to prioritize the energy consumption of his/her app over design quality.

Table 7.8 Hypervolume. Median and IQR.

	MOCell	NSGAII	SPEA2
calculator	$1.32e-1_{8.3e-2}$	$8.92e-02_{1.3e-1}$	$9.47e-2_{1.8e-1}$
gltron	$0.00e+0_{0.0e+0}$	$0.00e+0_{0.0e+0}$	$0.00e+0_{0.0e+0}$
kindmind	$0.00e+0_{1.0e-1}$	$0.00e+0_{0.0e+0}$	$0.00e+0_{0.0e+0}$
matrixcalculator	$2.50e-1_{0.0e+0}$	$2.50e-1_{0.0e+0}$	$2.50e-1_{0.0e+0}$
monsterhunter	$0.00e+0_{0.0e+0}$	$0.00e+0_{0.0e+0}$	$0.00e+0_{0.0e+0}$
prism	$0.00e+0_{0.0e+0}$	$0.00e+0_{0.0e+0}$	$0.00e+0_{0.0e+0}$
quicksnap	$0.00e+0_{0.0e+0}$	$0.00e+0_{0.0e+0}$	$0.00e+0_{0.0e+0}$
scrabble	$0.00e+0_{0.0e+0}$	$0.00e+0_{0.0e+0}$	$0.00e+0_{0.0e+0}$
soundmanager	$0.00e+0_{0.0e+0}$	$0.00e+0_{0.0e+0}$	$0.00e+0_{0.0e+0}$
stk	$0.00e+0_{0.0e+0}$	$0.00e+0_{0.0e+0}$	$0.00e+0_{0.0e+0}$
sudowars	$4.25e-1_{1.3e-1}$	$4.95e-1_{1.2e-1}$	$5.45e-1_{1.2e-1}$
tapsoffire	$0.00e+0_{0.0e+0}$	$0.00e+0_{3.7e-2}$	$0.00e+0_{3.7e-2}$
words	$3.00e-1_{5.3e-2}$	$2.69e-1_{7.3e-2}$	$2.73e-1_{7.0e-2}$

Table 7.9 Spread. Median and IQR.

	MOCell	NSGAII	SPEA2
calculator	$6.89e-1_{3.0e-1}$	$1.12e+0_{4.7e-1}$	$8.73e-1_{5.6e-1}$
gltron	$6.78e-1_{1.8e-1}$	$1.07e+0_{1.8e-1}$	$1.08e+0_{2.7e-1}$
kindmind	$6.93e-1_{1.0e-1}$	$9.71e-1_{2.2e-1}$	$7.66e-1_{3.0e-1}$
matrixcalculator	$5.00e-1_{0.0e+0}$	$1.39e+0_{0.0e+0}$	$1.49e+0_{3.5e-3}$
monsterhunter	$8.97e-1_{4.3e-1}$	$9.70e-1_{2.1e-1}$	$9.27e-1_{1.1e-1}$
prism	$0.00e+0_{0.0e+0}$	$1.94e+0_{3.8e-2}$	$1.92e+0_{4.6e-2}$
quickSnap	$1.95e-1_{4.1e-1}$	$1.29e+0_{6.0e-1}$	$1.00e+0_{1.4e+0}$
scrabble	$5.00e-1_{1.0e+0}$	$1.50e+0_{3.8e-1}$	$1.62e+0_{7.8e-1}$
soundmanager	$1.00e+0_{1.7e-1}$	$1.00e+0_{0.0e+0}$	$1.00e+0_{0.0e+0}$
stk	$0.00e+0_{0.0e+0}$	$1.95e+0_{2.9e-02}$	$1.91e+0_{1.5e-1}$
sudowars	$7.96e-1_{1.3e-1}$	$8.53e-1_{1.4e-1}$	$8.41e-1_{1.3e-1}$
tapsoffire	$7.53e-1_{5.4e-1}$	$1.00e+0_{1.7e-1}$	$1.00e+0_{8.6e-2}$
words	$6.84e-1_{2.5e-1}$	$9.42e-1_{2.2e-1}$	$7.07e-1_{1.6e-1}$

Table 7.10 Pair-wise Whitney U Test test for HV and Spread indicators.

EMO Pair	Quality Indicator	▲	▽	–
MOCell, SPEA2	HV	0	0	13
	Spread	7	0	6
MOCell, NSGA-II	HV	0	1	12
	Spread	10	0	3
NSGA-II, SPEA2	HV	0	0	13
	Spread	1	0	12

The process of validation consists in manually applying the sequence of refactorings to their corresponding source code, for each of the studied apps. We ran the scenario after applying each sequence to ensure that we are not introducing code regression. Finally, we compile and generate the APK file to deploy it in the mobile device and measure their energy consumption using our experimental setting described in Section 7.2. With this *EC* validation, we want to estimate EARMO's median error with respect to real measurements.

Concerning the scenarios used for *EC* validation, we defined new ones for the apps where we consider that the scenario used in the preliminary study do not reflect a typical usage. The reason is that in the preliminary study we were only interested in executing the code segment

related to an anti-pattern instance in the original version and its corresponding refactored code segment. The scenarios of Table 7.2 were just designed to check if a correlation exists between energy consumption and anti-pattern occurrences. Some scenarios designed for the preliminary study just required to start the app, wait certain seconds, and close it to execute the refactored code segment. For the *EC* validation we want to reflect the actions that a user typically will perform with an app, according the purpose of their creators, instead of scenarios designed to maximize other metrics like coverage which do not reflect the daily use of normal users. To validate EARMO (and perform optimization) we replace the scenarios in Table 7.2, *i.e.,* the ones that only load and close an app, by the ones presented in Table 7.11. Note that in some cases we have to modify the code to remove any sources of randomness that may alter the execution path between different runs. For example, *Sudowars* is a sudoku game where the board is randomly generated. Because in the scenario we introduce fixed numbers in fixed positions of the board, we need to ensure that the same board is always displayed to produce the same execution path over the 30 independent runs. Hence, we fixed the random seed used in the app to force to display always the same board. A similar case happens to another board game (*scrabble*).

Table 7.11 Description of scenarios generated for the *EC* validation and duration (in seconds).

App	Scenario	Duration
Calculator	Same scenario as preliminary study.	17.94
GLTron	Tap screen to start the game and wait until the moto crashes.	40.08
kindmind	Select each category, wait for the relaxation message, and close app.	80.06
monsterhunter	Same scenario as preliminary study.	16.39
oddscalculator	Select two players, {7 heart}, {8 heart}, {9 heart}, tap {calculate}, wait five seconds, and close app.	45.83
quicksnap	Take a picture and close app.	16.30
SASAbus	Same scenario as preliminary study.	71.72
scrabble	Assign the first four letters to the first cells, tap {confirm}, and close app.	65.11
soundmanager	Same scenario as preliminary study.	18.74
stk	Wait until content is downloaded, tap {karts}, tap first row, back, back, tap {tracks}, tap first row, and close app.	86.55
sudowars	Wait until app is loaded, tap {manual}, tap {single player}, tap {tick} button, select first square and write values 1, 2, 3, 4, 5, and 6, tap {...}, tap {assistant}, give up, tap yes, tap back, close app.	53.13
tapsoffire	Same scenario as preliminary study.	25.96
words	Select a category, tap {play}, tap {flash card}, tap {green hand}, tap {flash card}, tap {red hand}, tap {back}, and close app.	57.34

For the manual application of the sequence of refactorings, two Ph.D. candidates with more than 5 years of experience in Java, and a MSc. Student with two years of programming

experience, worked together. After each team member finished to apply a refactoring sequence to an app, we shared the control version repository with the other team members for approval. In case of disagreement, we vote for either apply or exclude a refactoring operation(s) from a sequence. Additionally, whenever we observed an abnormal behavior in the app after applying a refactoring, we rolled back to the previous code version and discarded the conflicting refactoring. We provide a link to the git repositories containing the refactored versions available online at `http://swat.polymtl.ca/rmorales/EARMO/`.

It is important to mention that we applied the refactorings using the Android Studio tool support, and we do not find cases where refactorings violate any semantic pre- and post-condition. However, there are many cases, specially in *move method* refactoring, and in *replace inheritance with delegation*, where it is possible to introduce regression despite the fact that the refactoring is semantically correct. Due to the absence of a test suite, we execute the defined scenario on the phone after applying each refactoring, to validate the correct execution of it. This is crucial, because an app could be executed even if the refactoring applied introduces regression until we exercise the functionality related to the code fragment touched by the refactoring. When we notice that the refactoring is not exercised in the defined scenario, we separately test that functionality.

In Table 7.12 we present the results of the manual refactoring application. The column *Discarded ref.* is the number of refactorings discarded from the sequence; *Applied ref.* the refactorings applied, and *Total* is the sum of both columns. *Precision* is the ratio of refactorings generated over the valid refactorings. Overall, EARMO shows a good precision score (68%) for all apps. In fact, only in 20% of the apps, the precision is less than 50%. From these apps, we discuss Prism, which is the app with lowest precision score. We found one out of five refactorings to be valid, and that one is the *IGS* type; three refactorings attempt to inline autogenerated classes from Android build system (*e.g.,* `R`, `BuildConfig`); one attempts to inline a class that extends from `android.app.Activity` class which is not invalid. From the four refactorings discarded of *Prism*, three can be consider valid but useless, and only one will introduce regression. Later, we provide guidelines for toolsmiths interested in developing refactoring tools for Android. With respect to the total number of refactorings applied, we observe that in seven cases we apply more than 20 refactorings, and from this subset two of them require more than 100. This validate our idea, that an automated approach can be useful for developers and software maintainers interested in improving the design of their apps, but with limited budget time to perform a dedicated refactoring session for all classes existing in their app.

In Table 7.13 we present EARMO median execution time $Exec.Time$, estimation values

Table 7.12 Summary of manual refactoring application for the EC validation.

App	DI%	EI%	Discarded ref.	Applied ref.	Total	Precision (%)
Calculator	75	54	19	45	64	70
BlackJackTrainer	75	6	3	1	4	25
GlTron	94	26	19	13	32	41
Kindmind	80	19	7	23	30	77
MatrixCalculator	33	1	0	1	1	100
MonsterHunter	82	44	29	83	112	74
mylocation	100	2	1	1	2	50
OddsCalculator	100	15	0	6	6	100
Prism	86	9	4	1	5	20
Quicksnap	92	85	69	119	188	63
SASAbus	82	27	3	8	11	73
Scrabble	86	13	0	6	6	100
SoundManager	94	36	3	5	8	63
SpeedoMeter	100	6	1	1	2	50
STK	83	12	2	3	5	60
Sudowars	71	64	38	75	113	66
Swjournal	100	6	13	6	19	32
TapsofFire	83	89	21	139	160	89
Vitoshadm	100	4	0	2	2	100
Words	75	63	23	76	99	77
			Total	614	Median	68

of energy consumption EC, median energy consumption of an app before (E_0) and after (E') refactoring. The difference between EC and E', $\gamma(EC, E')$ is calculated by subtracting $EC - E'$ and dividing the result by E' and the result is multiplied by 100. Similarly, we calculate the difference between E' and E_0, $\gamma(E', E_0)$. From the statistical tests between E_0, E', the $p-$value, and *effect size* (ES). The last column is the median difference of battery life duration, in minutes, between the original and the refactored version ($Diff.Batterylife$). This is of special interest for software maintainers to assess if the impact of applying a refactoring sequence would be noticeable to end users. We provide details of how to compute the last column below. This procedure has been used in previous works [185].

For each app we calculate its battery usage (in %) using Equation 7.7 to estimate the percentage of battery charge that is consumed by an app when running the defined scenario. E is the energy consumption in Joules of an app (derived from the median of the 30 independent runs), and V and C are the voltage and electric charge (in mAh), respectively, of the phone battery. For Nexus 4, $V = 3.8$ and $C = 2100mAh$.

$$Battery_{usage} = \frac{E}{V} \times \frac{1000}{C \times 3600} \times 100 \tag{7.7}$$

After obtaining the battery usage for both versions (original, and refactored) of each app, we use it to compute the battery life (in hours) using Equation 7.8 where ET is the execution time of the app (in seconds). We consider the battery life of an app to be the time that it takes to drain the battery if the scenario associated to the app is continuously run.

$$Battery_{life} = \frac{(ET \times 100)/Battery_{usage}}{3600} \tag{7.8}$$

Finally, we calculate the average battery life for each app (original and refactored) and subtracted these values to obtain the difference of battery life ($Diff.Batterylife$). Positive values are desired, as they mean that the battery life is longer using the refactored version, while negatives values mean the opposite effect.

Note that we did not consider apps in the validation where the number of refactorings applied is one, that accounts for six apps. The reason is that for these apps the energy improvement estimation EI is inferior to 10% before the manual application of refactorings, so we do not expect a measurable energy consumption change. In addition, we also omit *Swjournal*, in which we applied six refactorings out of 13, but given its low EI of 6% it is unlikely to report a noticeable change either.

For the remaining 13 apps, we observe that the median execution time for generating the refactoring sequences is less than a minute (56 seconds). Concerning energy estimation (EC), the direction of the trend holds for all the apps in the testbed according to the results measured E'. Concerning the accuracy of the estimation, EARMO values are more optimistic than the actual measurements but in an acceptable level. There are some remarkable exceptions, like *Tapsoffire* where the difference is 50%. In this app, most of the refactorings are *move method* type (120). If we multiply 120 by E_0, and the result by $\delta EC(move\,method)$ we have an energy consumption decrease of -1.64 J; 12 refactorings of *inline private getter and setters* type that account for -1.92 J. These two refactorings consume in total 3.56 J. The rest of the energy is divided between six *IPO* and one *replace HashMap with ArrayMap*. However, the impact on energy for this app is far from this value, probably because the scenario does not exercise (enough) the code that is modified by the refactorings to report a considerable gain. On the other side, *STK* reports the most close prediction with a difference of 3%. The refactorings applied are three *inline getter and setters*. If we compare the results obtained by EARMO compared with the preliminary study, the energy consumption trend holds for all the apps. However it is hard to make a fair comparison because in the Preliminary study we measure the effect of one instance of each anti-pattern type at a time, but in the energy consumption validation of EARMO we apply few to several refactorings. Although we assume that the effect of refactoring is aggregated, it is difficult to prove it with high precision, since we could not exercise all possible paths related to the refactored code in the proposed scenarios. Yet, the median error of $\gamma(EC, E')$ is in acceptable level of 12%, like the one reported by Wan et al. [126], when estimating the energy consumption of graphic user interfaces in a testbed of 10 apps.

Concerning the difference in energy consumption after refactoring, we observe that for three apps we obtain statistical significant results, with large effect size (results are in bold). This corroborates the findings in the preliminary study, for these apps. Although, for the rest of the apps the results are not statistically significant, we still we believe that the results are sound with respect to the energy consumption improvements reported. A recent work by Banerjee reported an energy consumption improvement from 3% to 29% in a testbed of 10 *F-Droid* apps with an automated refactoring approach for correcting violations of the use of energy-intensive hardware components [186]. With respect to battery life, EARMO could extend the duration (for the apps where the difference is statistically significant) of the battery from a few minutes up to 29 minutes (see the remarkable increment reported for *Words*). Note that to obtain a similar outcome in battery life, the proposed scenarios should be executed continuously, draining the battery from full to empty, which is not impossible, but rather unlikely. Yet, the benefits of improving design quality of the code, and potentially reducing the energy consumption of an app should not be underestimated. Not only because battery life is one of the main concerns of Android users and every small action performed to keep a moderate energy usage in apps is well appreciated. But, even if there is not a noticeable gain in energy reduction, software maintainers are safe to apply refactoring recommendations proposed by EARMO without fearing to introduce energy leaks.

Guidelines for toolsmiths designing refactoring recommendation tools.

We discuss some issues that should be considered for toolsmiths interested in designing refactoring recommendation tools for Android based on our experience applying the suggestions generated by EARMO. We should note that the tool that we use for detecting anti-patterns, which is DECOR, is not developed for Android platform. Hence, it does not consider the control flow depicted in Figure 7.3 and the OS mechanisms of communication between apps. This could generate false positives and consequently impact the generation of refactoring opportunities. Toolsmiths interested to develop refactoring tools for mobile platforms, based on anti-pattern detection tools aimed to target OO, should adapt the detection heuristics to avoid generating invalid refactoring operations. We discuss some strategies to consider below.

Excluding classes autogenerated by android build system. The classes `<app package>.R`, and `<app package>.BuildConfig` should not be considered for analysis of anti-patterns as they are automatically generated when (re)building an app .

Classes extending classes from `android.content` *package and its corresponding subpackages.* This package provides classes for accessing and publishing data on a mobile device and messaging between apps. As an example, consider `android.content.BroadcastReceiver`,

Table 7.13 EARMO execution time (seconds), EC estimation (J), median energy consumption E_0 and E' (J), γ values, statistical tests, and difference in battery life (minutes).

App	$Exec.Time$	EC	E_0	E'	$\gamma(EC, E')$	$\gamma(E', E_0)$	$p-value$	ES	$Diff.Batterylife$
calculator	154.90	17.40	21.28	19.49	-11%	-8%	**1.86E-09**	-0.94	2.55
gltron	55.98	242.27	256.44	252.15	-4%	-2%	**8.01E-08**	-0.77	0.42
kindmind	34.59	17.10	18.72	18.9	-10%	1%	0.1294	0.21	-4.61
monsterhunter	237.10	13.63	16.07	16.05	-15%	0%	0.6263	-0.03	-0.82
oddscalculator	8.98	29.25	30.61	30.94	-5%	1%	0.1094	0.22	-2.06
quicksnap	418.82	11.52	15.33	15.29	-25%	0%	0.9193	-0.04	3.33
SASAbus	32.39	3.79	4.61	5.49	-31%	19%	0.7922	0.09	-2.03
scrabble	18.55	88.68	94.56	94.14	-6%	0%	0.9193	-0.03	2.45
soundmanager	25.70	1.75	1.96	2.00	-13%	2%	0.3492	0.16	1.88
stk	24.58	240.82	252.81	249.29	-3%	-1%	0.1403	-0.16	0.99
sudowars	203.60	46.21	54.27	53.99	-14%	-1%	0.0879	-0.20	1.07
tapsoffire	281.00	3.30	6.80	6.59	-50%	-3%	0.9354	-0.02	1.97
words	119.65	25.16	27.01	25.13	0%	-7%	**0.0384**	-0.27	29.71

which allows an app to receive notifications from relevant events beyond the app's flow, e.g., a user activating the airplane mode. An app can receive broadcasts in two different ways. (1) declaring a *broadcast receiver* in the app's manifest; (2) creating an instance of class `BroadcastReceiver`, and register within a context [187]. We focus in the first method, as is the one that could lead developers to introduce regression (even using IDE's refactoring tool support). In *manifest-declared* receivers, the receiver element is registered in the app's manifest, and a new class is extended from `BroadcastReceiver` which requires to implement `onReceive(context, Intent)` method, to receive the contents of the broadcast. Let us briefly discuss the main issue when generating refactoring opportunities for classes extending from android.content packages (in this example we focus in `BroadcastReceiver`) depending on the type of refactoring to be applied. Collapse hierarchy refactoring is not considered as `BroadcastReceiver` does not belong to the app's package. Replace inheritance with delegation will introduce regression when removing the hierarchy relationship with `BroadcastReceiver`. We observe the same issue with inline class when trying to move the methods and attributes to other potential class. Move method will introduce regression too, when trying to move inherited methods like `onReceive` to another class.

Collapsing hierarchy of classes registered as Android activity. When a refactoring operation consists of applying Collapse hierarchy refactoring to a class that extends from `Activity`, it is also necessary to update the app's manifest with the name of the parent class.

Replacing Hashmap with ArrayMap. It is necessary to replace the imports for `android.support.v4.util` when Android API is less than 19, or `android.util` otherwise. It is important to mention that `ArrayMap` is defined as final, so it limits the possibility to derive a new implementation from this class, contrary to `HashMap` and its derived classes (e.g., `LinkedHashMap`).

RQ3: To what extent is design quality improved by EARMO according to an external quality model?

In **RQ1**, we have shown that EARMO is able to find optimal refactoring sequences to correct anti-patterns while controlling for energy consumption. Although anti-patterns occurrences are good indicators of design quality, a software maintainer might be interested in knowing whether the applied refactorings produce code that is for example readable, easy to modify and–or extend. To verify such high-level design quality attributes, we rely on the QMOOD quality model. Table 7.14 presents the maximum and minimum quality gain achieved after applying the refactorings suggested by EARMO, for each app studied and for each QMOOD quality attribute.

— *Reusability, understandability and flexibility.* In general, the refactored apps report a slight decrease that ranges from 0.9% to 4% for these attributes. In the case of reusability, the *prism* app is an outlier, with a medium deterioration of reusability between 31% and 44%. EARMO finds two refactoring sequences (or two non-dominated solutions in the Pareto front) that are comprised of five refactoring operations. These refactorings are three *inline* operations, which have negatively impacted the *reusability* value because of the weight (*i.e.,* 0.5) that reusability assigns to the number of entities in the system (DSC metric). The fourth refactoring is *Inline private getters and setters*, which negatively affects the cohesion among methods (CAM) because one getter is inlined in the system. The last refactoring of the first refactoring sequence is *replace inheritance with delegation* which negatively impacts the coupling between classes (DCC), leading to a drop of 44.36% (minimum value) of reusability. In the second refactoring sequence, the last refactoring is *collapse hierarchy* which negatively impact DSC metric as well. Concerning understandability, we observe little variation through all the apps, making it the least impacted attribute among the five attributes studied. Finally, for flexibility we report a median of -4.07%. One remarkable case is *Mylocation*, with 100% gain for this attribute. It has one solution comprised of two refactorings, inline class and move resource request to visible method. While the former one does not have a direct impact on the design, the inline of a class positively impacted this attribute because the number of classes is small (only nine classes). Similarly, *Oddscalculator* contains one solution with seven inline class refactorings, and one inline private getter. On the other hand, *Swjournal* has one solution composed mainly by *move method* refactorings (19), and one inline class. The inline class operation is likely responsible for the drop of the value of the attribute to 45%.

— *Effectiveness.* We report a small gain of 3.14%, with two outliers (*Oddscalculator* and *Soundmanager*). As we discussed before, *Oddscalculator* is mainly composed of inline

Table 7.14 Quality gain achieved by EARMO on QMOOD quality attributes.

	Reu.		Und.		Fle.		Eff.		Ext.	
App Name	Min.	Max.	Min.	Max.	Min.	Max.	Min.	Max.	Min.	Max.
blackjacktrainer	-3.96	-3.96	-4.05	-4.05	-11.13	-11.13	9.29	9.29	94.86	94.86
calculator	-1.06	-0.58	-1.00	0.11	-14.52	6.73	1.85	3.18	13.51	21.07
gltron	-8.19	-2.83	-4.25	-2.39	-10.54	-4.93	3.79	6.12	38.01	40.79
kindmind	-1.10	-0.67	0.87	0.93	-0.12	1.78	-0.25	0.36	58.08	58.62
matrixcalculator	0.00	2.16	0.05	0.33	0.34	35.64	-0.51	-0.25	89.87	100.36
monsterhunter	0.08	0.10	0.00	0.10	0.43	0.73	0.42	0.48	57.22	57.69
mylocation	-1.56	-1.56	1.49	1.49	100.00	100.00	7.39	7.39	1.25	1.25
oddscalculator	-5.31	-5.31	-5.28	-5.28	70.86	70.86	28.93	28.93	42.15	42.15
prism	-44.36	-31.27	-8.14	-6.10	-14.46	-10.60	7.53	10.22	65.17	78.30
quicksnap	-2.74	-2.72	-3.77	-3.51	-39.15	-37.23	1.89	2.25	4.15	4.91
sasabus	-0.24	-0.24	-0.07	-0.07	-0.41	-0.41	1.11	1.11	64.57	64.57
scrabble	-8.41	-7.30	-0.80	-0.05	-13.41	-10.20	9.79	12.77	-1.67	1.60
soundmanager	-7.39	-5.67	-5.02	-3.40	-14.65	-5.92	24.11	26.17	32.32	44.32
speedometer	-0.93	-0.93	-1.22	-1.22	55.56	55.56	9.72	9.72	-124.16	-124.16
stk	-0.01	0.53	0.18	0.34	1.21	3.74	1.35	1.35	55.05	55.96
sudowars	-2.71	-0.76	-2.10	-1.12	-12.42	-5.43	-0.94	0.24	25.16	30.52
swjournal	-4.14	-4.14	-2.45	-2.45	-45.33	-45.33	0.87	0.87	6.88	6.88
tapsoffire	-0.39	-0.07	-2.97	-2.90	-13.36	-12.24	4.87	4.98	18.38	19.13
vitoshadm	-0.21	-0.21	0.10	0.10	8.71	8.71	3.79	3.79	153.06	153.06
words	2.11	3.92	0.44	0.81	4.19	8.11	-6.27	-3.70	72.88	74.27
Median values for all PF solutions	**-1.24**		**-0.94**		**-4.07**		**3.14**		**40.78**	

class refactorings. *Soundmanager* has two solutions, both contain nine inline classes, six *inline getters/setters*, and two replace HashMap usage operations. In addition, the second solution includes introduce parameter-object refactoring, which adds a new class to the design, has the highest effectiveness value for this app.

— *Extendibility.* For this attribute we report a considerable improvement of 41%. We attribute this increment to the removal of unnecessary inheritance (through inline class, collapse hierarchy and replace inheritance with delegation refactorings). In fact, the extendibility function assigns a high weight to metrics related to hierarchy (*i.e.,* MFA, ANA). These are good news for developers interested in improving the design of their apps through refactoring, as the highly-competitive market of Android apps requires adding new features often and in short periods of time. Hence, if they interleave refactoring before the release of a new version, it will be easier to extend the functionality of their apps.

We conclude that our proposed approach EARMO can improve the design quality of an app, not only in terms of anti-patterns correction, but also their extendibility, and effectiveness.

RQ4: Can EARMO generate useful refactoring solutions for mobile developers? We conducted a qualitative study with the developers of the 20 apps studied in this chapter to gather their opinion about the refactoring recommendations of EARMO. The study took

place between August 17th and September 17th 2016. 23 developers, identified as authors in the repository of the apps, were contacted but only 8 responded providing feedback for a total of 8 apps. Table 7.15 provides some background information on the developers that took part in our qualitative study. Each developer has more than 3 years of experience and their primary programming language is Java. Half of the developers use Android Studio to program. 100% of them considered refactorings to be useful but only 12% said that they perform refactoring frequently. We asked each developer to name the three refactorings that they perform the most. As we can see in Table 7.15, the most frequent refactorings performed by the developers are: to remove dead code, move method, inline class, extract class/superclass, collapse hierarchy, and extract interface. They also mentioned to extract repetitive code into new functions (extract method), and adjusting data structures.

Table 7.15 Background information on the surveyed developers.

App Name	Interval Age	Experience	Prog. Language	IDE	Top refactorings
Calculator	18 to 24	5-9 years	Java	Android Studio	Extract method, remove dead code, extract or remove new class/interface
OddsCalc	35 to 44	3-4 years	Java	Eclipse	Move type to new file, move method/field.
Kindmind	25 to 34	<1 year	Java	Android Studio	Renaming variables and classes, extract method/class
GLTron	35 to 44	3-4 years	Swift	XCode	Adjusting data structures, move method, extract class/-superclass, Inline class, Collapse hierarchy and extract interface
Scrabble	35 to 44	3-4 years	python	vim	Extract method, remove dead code, add encapsulation
Prism	45 to 54	10 years or more	Java	Eclipse	Extract variable, extract method, rename
Matrixcalc	18 to 24	3-4 years	Java	Android Studio	Refactoring duplicate code, renaming classes/methods and variables, remove dead code
STK	18 to 24	1-2 years	Java	Android Studio	Extract method, extract class

For each app, we randomly selected three refactorings for each refactoring type, from the refactoring sequence in the Pareto front with the highest energy gain. We submitted the proposed refactorings to the developers of the app. We asked the developers if they accept the solution proposed by EARMO, and if not, to explain why. We also asked if there were any modification(s) that they would like to suggest to improve the proposed refactoring recommendations. In Figure 7.10, we present the acceptance ratio of the refactoring solutions proposed by EARMO, by app (left), and by anti-pattern (right).

We can observe that for four apps (*prism, scrabble, stk, matrixcalculator*), 100% of the refactorings suggested by EARMO were accepted. For three other apps (*calculator, kindmind, oddscalculator*) the acceptance ratio range from 40% to 57%. The developer of the *GLTron* app rejected all the refactorings recommended for the app. However, some of the reasons behind her/his rejections are not convincing as we will discuss in the following paragraph.

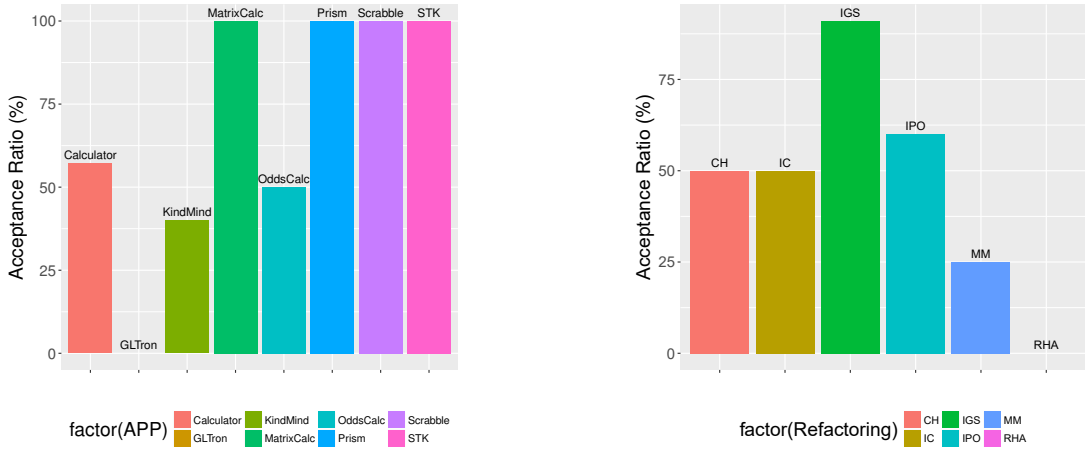

Figure 7.10 Acceptance ratio of the refactorings proposed by EARMO.

Overall, 68% of recommendations suggested by EARMO were accepted by developers.

The refactoring with the highest acceptance ratio is inline private getters and setters, while the one with the lowest acceptance ratio is replace HashMap with ArrayMap. The only app for which replace HashMap with ArrayMap was recommended is *GLTron*. The argument provided by the developer of *GLTron* to justify his disapproval of the refactoring is that because "GLtron runs on many platforms, introducing too many Android specific APIs would be a bad idea from a portability point of view". He also mentioned that because the HashMap contains few objects, the impact on performance is minimal. However, the Android documentation [50] emphasizes the advantages of using ArrayMap when the number of elements is small, in the order of three digits or less. In addition to this, the performance in energy consumption should not be ignored.

Move method refactoring has an acceptance rate of 25%. The following reasons were provided by developers to justify their decision to reject some move method refactorings suggested by EARMO. For the *calculator* app, the developer rejected two suggested *move method* refactorings, arguing that the candidate methods' concerns do not belong to the suggested target classes. However, s/he agrees that the source classes are Blobs classes that should be refactored. We obtained a similar answer from the developer of *Kindmind*, who also agrees that the classes identified by EARMO are instances of Blob, but proposes other target classes as well. To justify her/his rejection of all the three move method refactorings that were suggested for her/his app, the developer of *GLTron* argued that there are more important issues than moving a single method. However, (s)he didn't indicate what were those issues.

Introduce parameter object. We found long-parameter list instances in *matrixcalculator*,

STK and *GLTron*, and its only in *GLTron* that the developer rejected the two refactorings proposed, claiming that the new object will bloat the calling code of the method; and for the second one, that the method has been already refactored in a different way.

Collapse hierarchy. We found two instances of speculative generality, one in *Prism* (which was accepted) and another in *Calculator*; the latter one was rejected because the collapsed class (which is empty) implements a functionality in the paid version. The developer wanted to keep the empty class to maintain compatibility between the two versions of the app (*i.e.,* free and paid versions). However, the developer agrees that the solution proposed by EARMO is correct, and will consider to remove the empty class in the future.

Inline class. Two inline class refactorings were proposed by EARMO, one in *Scrabble* and another in *Oddscalculator*. The former one was rejected by the developer because (s)he considers that inlining the lazy class will change the idea of the design.

Inline private getters and setters. EARMO recommended Inline private getters and setters refactorings in 7 out of the 8 apps for which we received developers' feedback. From a total of 11 Inline private getters and setters operations that were suggested by EARMO, only one was rejected, and this was in *GLTron*. The developer of *GLTron* argued that a method that is called only once require no performance optimizations.

The majority of recommendations made by EARMO were received favorably. For those that were rejected, it was not because they were incorrect or invalid, but because they affected certain aspects of the design of the apps that developers did not wanted to change. The recommendations made by EARMO raised the awareness of developers about flaws in the design of their apps. This was true even when the suggested fixes (*i.e.,* the refactorings) for these design flaws were rejected by the developers.

Hence, we conclude that EARMO recommendations are useful for developers. We recommend that developers use EARMO during the development of their apps, since it can help them uncover design flaws in their apps, and improve the design quality and energy consumption of their apps.

7.5 Threats to validity

This section discusses the threats to validity of this chapter's case studies following common guidelines for empirical studies [143].

Construct validity threats concern the relation between theory and observation. This is mainly

due to possible mistakes in the detection of anti-patterns, when applying refactorings. We detected anti-patterns using DECOR [52] and the guidelines proposed by Gottschalk and the Android performance tips [48, 51]. However, we cannot guarantee that we detected all anti-patterns, or that all those detected are indeed true anti-patterns. Concerning the application of refactorings, for the preliminary study, we use the refactoring tool support of Android Studio and Eclipse, to minimize human mistakes. In addition, we verify the correct execution of the proposed scenarios and inspect the ADB Monitor to avoid introducing regression after refactoring was applied. Concerning the correction improvement reported by EARMO, we manually validated the outcome of refactorings performed in the source code with respect to the ones applied to the abstract model, to ensure that the output values of the objective functions correspond to the changes performed. However, we rely on the correct representation of the code generated by *Ptidej Tool Suite* [156], that has been used in several studies of anti-patterns, and software maintenance. Considering energy measurements we used a phone model used in previous works. Plus our measurement apparatus has the same or higher number of sampling bits than previous studies, and our sampling frequency is one order of magnitude higher than past studies. Overall, we believe our measurements are more precise or at least as precise as similar previous studies. As in most previous studies we cannot exclude the impact of Android operating system. What is measured is a mix of Android and application actions. We mitigate this by running the application multiple times and we process energy and execution traces to take into account only the energy consumption of method calls belonging to the app. Because interpreted code runs slowly when profiling is enabled, it is probable that the energy consumption associated with each method call is higher. However, given that the profiling was enabled in all the experiments, we can assume that the instrumentation overhead introduced by the production of execution traces is constant between different runs of the same scenario.

Threats to internal validity concern our selection of anti-patterns, tools, and analysis method. We used a particular yet representative subset of anti-patterns as a proxy for design quality. Regarding energy measurements, we computed the energy using well known theory and the scenarios were replicated several times to ensure statistical validity. From the set of anti-patterns studied, we target one that is related to the use of device sensors, that is Binding Resources too early. Because our setup is measured inside a building, device location might be computed using Wi-Fi instead of GPS if the reception is not good enough. In that case, it is likely to be less than the cost of using GPS sensor outdoors. This also applies to network connections, where the cost incurred for connecting through Wi-Fi is likely to be less than the one incurred for using a cellular network. Additionally, in the evaluation of EARMO we use MonkeyRunner to communicate with apps though simulated signals rather

than signals triggered through real sensors (for example, touchscreens or gravity sensors) on mobile devices, that could be regarded as not realistic. In case that a more realistic measurement is required, we can substitute intrusive methods, like Monkeyrunner, with a robot arm that uses the same cyber-physical interface as the human user [188].

Conclusion validity threats concern the relation between the treatment and the outcome. We paid attention not to violate assumptions of the constructed statistical models. In particular, we used a non-parametric test, Mann-Whitney U Test, Cliff's δ ES, that does not make assumptions on the underlying data distribution.

Reliability validity threats concern the possibility of replicating this study. The apps and tools used in this study are open-source. To obtain the same results, the same model of phone and version of Android operating system should be used. Moreover, the scenarios defined for each application are only valid for the apps versions selected for our testbed. The reason is that the scenarios were build based on absolute coordinates of the screen. If another model of phone is used, or the app's interface changes, the scenarios will not be valid.

Threats to external validity concern the possibility to generalize our results. These results have to be interpreted carefully as they may depend on the specific device where we ran the experiments, the operating system and the virtual machine (VM) used by the operating system. For the former one, it is well known that in ART (Android Run Time used in this work) the apps are compiled to native code once, improving the memory and CPU performance, while the previous VM for Android (Dalvik) runs along with the execution of an app, and may perform profile-directed optimizations on the fly. To validate this threat, we execute the energy consumption validation using Dalvik and ART VMs and found $\pm 1\%$ of difference in the median of $\gamma(E', E_0)$ values for the apps used in the energy consumption validation. Hence, we suggest that our results area valid for both VMs, for the set of anti-patterns, apps, and scenarios used in this work.

Our study focuses on 20 android apps with different sizes and belonging to different domains from *F-Droid*, which is one of the largest repositories of open-source Android apps. Still, it is unclear if our findings would generalize to all Android applications. Yet, more studies and possibly a larger dataset is desirable. Future replications of this study are necessary to confirm our findings. External validity threats do not only apply to the limited number of apps, but also to the way they have been selected (randomly), their types (only free apps), and provenance (one app store). For this reason this work is susceptible to the App Sampling Problem [189], which exists when only a subset of apps are studied, resulting in potential sampling bias. Nevertheless, we considered apps from different size and domains, and the anti-patterns studied are the most critical according to developers perception [7, 110].

7.6 Chapter Summary

In this chapter, we have proposed EARMO, a novel approach that takes into account energy efficiency when refactoring mobile apps. Specifically, we set out to address the following question:

> **Central Question:** *Is it possible to improve automated refactoring of mobile apps by considering energy concerns?*

We answer this question by performing two case studies.

1. A preliminary study to assess the impact of well-known OO and mobile anti-patterns on energy consumption.
2. A full-scale study to evaluate EARMO using three evolutionary multiobjective techniques on a benchmark of 20 free and open-source apps.

The results of our empirical evaluations show that for the set of anti-patterns studied, they do impact the energy efficiency of mobile apps. That EARMO can propose solutions to remove a median of 84% of anti-patterns, with a median execution time of less than a minute. We also quantify the battery energy gain of EARMO and found that in a multimedia app, when the proposed scenario is executed continuously until the battery is drained out, it could extend the battery life by up to 29 minutes.

We also demonstrated that in the instance of search space explored by the metaheuristics implemented, different compromise solutions are found, justifying the need for a multiobjective formulation.

Concerning the quality of the solutions proposed, we manually evaluated the precision of the sequences generated by EARMO and obtained a median of 68% precision score. We study the cases where some of the refactorings in a sequence are not valid and provide valuable guidelines for toolsmiths to improve the precision of automated refactoring approaches.

We also evaluated the overall design quality of the refactored apps in terms of five high-level quality attributes assessed by an external model, and reported gains in terms of understandability, flexibility, and extendibility of the resulting designs.

Finally, we conducted a qualitative study to assess the quality of the refactoring recommendations made by EARMO from the point of view of developers. Developers found 68% of refactorings suggested by EARMO to be very relevant.

CHAPTER 8 CONCLUSION AND RECOMMENDATIONS

Refactoring is an important maintenance task aimed to support the evolution of software systems. Manually refactoring is a laborious and effort consuming task. Many researchers (*cf.* section 3.1.2) have proposed (semi)automated approaches to perform refactoring. These approaches have proven to be efficient in improving the design quality of the refactored systems. However, their analysis focus on a set of limited attributes (semantic coherence, development history) to guide the refactoring search process. The inclusion of developer's task context, refactoring effort, testing effort, and energy efficiency, could improve automated refactoring. Hence, in this dissertation, our thesis was:

Search-based refactoring can be readily used for automatically improving software design quality while (1) cutting the cost of anti-patterns detection and correction; (2) making more efficient use of computational resources than existing approaches; and (3) providing useful solutions from developer's perspective

To prove our hypothesis, we proposed to consider developer's task context, refactoring order and conflict awareness, testing effort, and energy efficiency of mobile apps to better guide the automated refactoring process.

8.1 Advancement of knowledge

In the following paragraphs, we summarize our main contributions to the advancement of the knowledge.

8.1.1 Improving automated refactoring using developer's task context

The first contribution of this dissertation is using developer's task context to prioritize the refactoring of relevant classes during regular coding activities. Instead of recommending a large list of refactorings in a root-canal refactoring style, we propose to put developers on the loop, and when they write their code, a list of candidate refactorings are generated based on their task context (floss refactoring style). This is a more natural way of refactoring, and help developers to improve the quality of their system, and to assess their design choices.

By performing a large-scale study over 1,705 interaction histories associated to bug fixing activities, we showed that the accumulated effect of refactoring only the classes contained in

developer's context corrected more than 50% of the total number of anti-patterns in a testbed of three OSS. We also observe that at least 41% of the anti-patterns detected using root-canal refactoring style, affect classes that were not present in developer's context, suggesting that the refactoring activity for these classes could be postponed it, or was not very critical. Hence, the solutions proposed by our refactoring approach prioritize the most important classes according to the developer's environment.

8.1.2 Improving automated refactoring through efficient scheduling

The second contribution is about reducing the search-space of the refactoring scheduling problem, by modeling conflict and dependencies among refactorings, and by taken inspiration from model checking techniques (partial order reduction) to eliminate redundant refactoring solutions. Our proposed approach, RePOR, was able to correct a median of 73% of anti-patterns, while reducing both refactoring effort, and execution time to 80% in comparison to the values achieved by three metaheuristics (GA, Ant Colony and LIU), on a testbed of five open-source systems. We also implemented RePOR as an Eclipse Plug-in, which can be freely downloaded from the author's personal web site [158], to perform automated refactoring during software development and maintenance activities.

8.1.3 Improving automated refactoring by considering testing effort

Our third contribution is considering testing cost while performing refactoring. First, we empirically showed that developer's design choices can affect the testability of a system, and that a compromise between design quality and testing effort exists. To control the effect of refactoring on testing effort, we formulate refactoring as multiobjective problem considering quality improvement and testing effort and solve it using three state-of-the-art EMO algorithms. On a testbed of four open-source systems, we show that our EMO algorithms could correct a median of 46% of anti-patterns while reducing testing effort by 48% compared to the initial system design. We also observe that considering only anti-patterns removal, yield to larger number of test cases. We also assessed the design quality of the refactored systems using QMOOD, and found that the effectiveness of the refactored systems was increased by up to 15%.

8.1.4 Improving automated refactoring of mobile apps by controlling for energy efficiency

The fourth contribution of this dissertation is using energy consumption of anti-patterns to improve the energy efficiency of mobile apps during refactoring activity. First, we showed that by removing anti-patterns (OO and Android) we can affect the the energy efficiency of mobile apps. Next, we proposed multiobjective refactoring approach, called EARMO, that controls for energy consumption while refactoring. We evaluate EARMO by performing a set of comprehensive case studies on a testbed of 20 android apps, and found the following results. EARMO corrected a median of 84% of anti-patterns, and achieved a median improvement of 41% of the extendibility of the app's design according to QMOOD model. Moreover, we found that 68% of the solutions proposed by EARMO were accepted by the original developers of the apps studied. Concerning energy efficiency of the refactored apps, we found that EARMO could extend the battery life by up to 29 minutes for a multimedia app when continuously running a typical scenario.

8.2 Recommendations and future work

In this dissertation, we have verified our thesis and proved that search-based refactoring can be enhanced by considering several dimensions (task context, refactoring effort, testing effort and energy efficiency) to improve the design quality of software systems. This results opens interesting new research directions that we describe below.

8.2.1 Automated Refactoring of testing artifacts

As we described in Section 2.1, it is important to keep consistency between software design and other software artifacts. Testing is crucial in software development cycle and we empirically proved that refactoring have an impact on testing effort. Indeed, we propose an approach to control for testing effort. However, testing artifacts could become obsolete after a major software architectural change (including refactoring), which requires developers to invest time to update and validate their integrity. In addition, previous works have report the existence of test anti-patterns, and their negative effects on software development activities [190, 191]. As part of our future work, we propose the development new mechanisms to automatically refactor testing artifacts in conjunction with production code.

8.2.2 Improving automated refactoring by considering code lexicon

One of the current problems in automated approaches is the production of sound identifiers names when transforming the code. Ouni et al. [26] proposed an approach to prioritize refactorings that maintain the semantic of a software system's naming conventions. However, their approach didn't leverage the the linguistic context of the systems, when naming the new entities added in the software systems (classes, methods, etc.) during refactoring. Therefore, we suggest that learning linguistic patterns from a software system could help to improve automated refactoring, as linguistic patterns capture the meaning of the code entities and the intention of the methods. Linguistic patterns help developers to better understand the code they are working on, and to ease software evolution and maintenance tasks. Moreover, as systems grown in size and complexity, the cost of learning and adopting the system's naming conventions for new developers integrating a team is not negligible. However, advancements in machine learning techniques now makes it possible to learn these linguistic patterns.

8.2.3 Evaluating the usefulness of automated approaches

The results of this dissertation emphasize the benefits of the refactoring solutions in terms of anti-patterns correction and desirable design quality attributes (reusability, understandability, flexibility, etc.). However, it is important to evaluate the usefulness of the solutions proposed by a refactoring approach from the perspective of developers. In chapter 7, we surveyed developers of mobile apps about the relevance of the refactorings proposed by EARMO in our testbed, and found 68% of the refactorings to be very relevant to developers. Yet, it is necessary to perform further studies with developers from open- and close- source systems to improve even more the usefulness of automated refactoring techniques.

As a result of the research performed in this dissertation, we came with the idea of perform several user studies to answer some of the following questions about the usefulness of automated refactoring. For example, how much does developers like automated refactorings in comparison to manual refactoring changes? Which refactorings can be safely automated without affecting the comprehensibility of the code? Can an automated approach perform better than a developer in performing refactoring? Beside naming conventions, which other code attributes are important to be considered in a refactoring approach to be useful for developers?

8.3 Final remarks

One of the main complains that we received from developers from the automated refactorings solutions proposed in this dissertation was not about the correctness of the solutions proposed, but the pertinence of the solutions with respect to developer's design quality priorities. With this respect, existing refactorings catalogs (Fowler, Brown, etc.) need to be updated to reflect the latest technology advancements and the emergence of new programming languages, and technologies (*e.g.,* cloud, mobile, Internet of things, etc.). Despite the large body of work on refactoring and automated approaches (*cf.* Chapter 3) and the approaches proposed in this dissertation, we foresee more work on the following topics: (1) the transition from legacy systems (*e.g.,* client-sever architectures) to emergent technologies through refactorings; (2) refactorings of software developed using new emergent technologies.

REFERENCES

[1] R. S. Pressman and W. S. Jawadekar, *Software engineering - A Practitioner's Approach*, 5th ed. McGraw-Hill Higher Education, 2001.

[2] M. M. Lehman, "Feedback in the software evolution process," *Information and Software Technology*, vol. 38, no. 11, pp. 681–686, November 1996.

[3] W. J. Brown, R. C. Malveau, W. H. Brown, H. W. McCormick III, and T. J. Mowbray, *Anti Patterns : Refactoring Software, Architectures, and Projects in Crisis*, 1st ed. John Wiley and Sons, March 1998.

[4] F. Khomh, M. D. Penta, Y.-G. Gueheneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change- and fault-proneness," *Empirical Softw. Engg.*, vol. 17, no. 3, pp. 243–275, Jun. 2012.

[5] J. O. Coplien and N. B. Harrison, *Organizational Patterns of Agile Software Development*, 1st ed. Prentice-Hall, Upper Saddle River, NJ (2005), 2005.

[6] A. J. Riel, *Object-oriented design heuristics*. Addison-Wesley Reading, 1996, vol. 335.

[7] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, and A. D. Lucia, "Do they really smell bad ? a study on developers' perception of bad code smells," in *Software Maintenance and Evolution (ICSME), 2014 IEEE Int'l Conference on*. IEEE, 2014, pp. 101–110.

[8] M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in *Software Maintenance and Reengineering (CSMR), 2011 15th European Conf. on*, March 2011, pp. 181–190.

[9] S. Vaucher, F. Khomh, N. Moha, and Y.-G. Guéhéneuc, "Tracking design smells : Lessons from a study of god classes," in *Reverse Engineering, 2009. WCRE'09. 16th Working Conf. on*. IEEE, 2009, pp. 145–154.

[10] A. Chatzigeorgiou and A. Manakos, "Investigating the evolution of bad smells in object-oriented code," in *Quality of Information and Communications Technology (QUATIC), 2010 7th Int'l Conf. on the*. IEEE, 2010, pp. 106–115.

[11] W. F. Opdyke, "Refactoring object-oriented frameworks," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1992.

[12] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring : Improving the Design of Existing Code*. Addison-Wesley, 1999.

[13] B. van Rompaey, B. Du Bois, S. Demeyer, J. Pleunis, R. Putman, K. Meijfroidt, J. C. Dueas, and B. Garcia, "Serious : Software evolution, refactoring, improvement of operational and usable systems," in *Software Maintenance and Reengineering, 2009. CSMR '09. 13th European Conf. On*, 2009, Conference Proceedings, pp. 277–280.

[14] Q. D. Soetens and S. Demeyer, "Studying the effect of refactorings : A complexity metrics perspective," in *Quality of Information and Communications Technology (QUATIC), 2010 7th Int'l Conf. On the*, 2010, Conference Proceedings, pp. 313–318.

[15] T. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, dec 1976.

[16] B. D. Bois, S. Demeyer, J. Verelst, T. Mens, and M. Temmerman, "Does god class decomposition affect comprehensibility ?" in *IASTED Conf. on Software Engineering*, 2006, Conference Paper, pp. 346–355.

[17] M. Kim, T. Zimmermann, and N. Nagappan, "A field study of refactoring challenges and benefits," in *Proc. of the ACM SIGSOFT 20th Int'l Symposium on the Foundations of Softw. Eng.*, ser. FSE '12. ACM, 2012, pp. 50 :1–50 :11.

[18] S. A. Slaughter, D. E. Harter, and M. S. Krishnan, "Evaluating the cost of software quality," *Commun. ACM*, vol. 41, no. 8, pp. 67–73, Aug. 1998. [Online]. Available : http://doi.acm.org/10.1145/280324.280335

[19] J. R. Cordy, "Comprehending reality-practical barriers to industrial adoption of software maintenance automation," in *Program Comprehension, 2003. 11th IEEE International Workshop on.* IEEE, 2003, Conference Proceedings, pp. 196–205.

[20] F. Khomh, T. Dhaliwal, Y. Zou, and B. Adams, "Do faster releases improve software quality ? : an empirical case study of mozilla firefox," in *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories.* IEEE Press, 2012, pp. 179–188.

[21] T. Mariani and S. R. Vergilio, "A systematic review on search-based refactoring," *Information and Software Technology*, vol. 83, pp. 14–34, 2017.

[22] M. O'Keeffe and M. Ó Cinnéide, "Search-based refactoring for software maintenance," *Journal of Systems and Software*, vol. 81, no. 4, pp. 502–516, 2008.

[23] O. Seng, J. Stammel, and D. Burkhart, "Search-based determination of refactorings for improving the class structure of object-oriented systems," *GECCO 2006 : Genetic and Evolutionary Computation Conference, Vol 1 and 2*, pp. 1909–1916, 2006.

[24] M. Kessentini, W. Kessentini, H. Sahraoui, M. Boukadoum, and A. Ouni, "Design defects detection and correction by example," in *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on.* IEEE, 2011, pp. 81–90.

[25] F. Qayum and R. Heckel, "Local search-based refactoring as graph transformation," in *Search Based Software Engineering, 2009 1st International Symposium on*. IEEE, 2009, Conference Proceedings, pp. 43–46.

[26] A. Ouni, M. Kessentini, H. Sahraoui, and M. S. Hamdi, "Search-based refactoring : Towards semantics preservation," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 2012, Conference Proceedings, pp. 347–356.

[27] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, and M. S. Hamdi, "Improving multi-objective code-smells correction using development history," *Journal of Systems and Software*, vol. 105, no. Supplement C, pp. 18–39, 2015. [Online]. Available : http://www.sciencedirect.com/science/article/pii/S0164121215000631

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

[29] K. Beck, *Extreme programming explained : embrace change*. addison-wesley professional, 2000.

[30] M. Linares-Vásquez, G. Bavota, C. E. B. Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, "Optimizing energy consumption of guis in android apps : A multi-objective approach," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 143–154.

[31] D. Li and W. G. J. Halfond, "An investigation into energy-saving programming practices for android smartphone app development," in *Proceedings of the 3rd International Workshop on Green and Sustainable Software*, ser. GREENS 2014. New York, NY, USA : ACM, 2014, pp. 46–53. [Online]. Available : http://doi.acm.org/10.1145/2593743.2593750

[32] A. Banerjee, L. K. Chong, S. Chattopadhyay, and A. Roychoudhury, "Detecting energy bugs and hotspots in mobile apps," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 588–598.

[33] M. Fowler, *Refactoring : improving the design of existing code*. Pearson Education India, 1999.

[34] T. Mens and T. Tourwé, "A survey of software refactoring," *Software Engineering, IEEE Transactions on*, vol. 30, no. 2, pp. 126–139, 2004.

[35] T. Tourwé and T. Mens, "Identifying refactoring opportunities using logic meta programming," in *Software Maintenance and Reengineering, 2003. Proceedings. Seventh European Conference on*. IEEE, 2003, pp. 91–100.

[36] A. Ouni, M. Kessentini, H. Sahraoui, and M. Boukadoum, "Maintainability defects detection and correction : a multi-objective approach," *Automated Software Engineering*, vol. 20, no. 1, pp. 47–79, 2013.

[37] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "JDeodorant : Identification and removal of type-checking bad smells," in *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*. IEEE, 2008, pp. 329–331.

[38] T. DeMarco, *Controlling software projects : Management, measurement, and estimates*. Prentice Hall PTR, 1986.

[39] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[40] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *Software Engineering, IEEE Transactions on*, vol. 28, no. 1, pp. 4–17, 2002.

[41] F. Khomh and Y.-G. Guéhéneuc, "Dequalite : Building design-based software quality models," in *Proceedings of the 15th Conference on Pattern Languages of Programs*, ser. PLoP '08. New York, NY, USA : ACM, 2008, pp. 2 :1–2 :7. [Online]. Available : http://doi.acm.org/10.1145/1753196.1753199

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

[43] M. O'Keeffe and M. O. Cinnéide, "Search-based software maintenance," in *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on*, 2006, Conference Proceedings, pp. 10 pp.–260.

[44] O. Seng, J. Stammel, and D. Burkhart, "Search-based determination of refactorings for improving the class structure of object-oriented systems," *GECCO 2006 : Genetic and Evolutionary Computation Conference, Vol 1 and 2*, pp. 1909–1916, 2006.

[45] M. Harman and L. Tratt, "Pareto optimal search based refactoring at the design level," in *Proceedings of the 9th annual conference on Genetic and evolutionary computation*. ACM, 2007, Conference Proceedings, pp. 1106–1113.

[46] A. Ouni, M. Kessentini, H. Sahraoui, and M. S. Hamdi, "The use of development history in software refactoring using a multi-objective evolutionary algorithm," in *Proceedings of the 15th annual conference on Genetic and evolutionary computation*. ACM, 2013, Conference Paper, pp. 1461–1468.

[47] M. Fowler, *Refactoring – Improving the Design of Existing Code*, 1st ed. Addison-Wesley, June 1999.

[48] M. Gottschalk, "Energy refactorings," Master's thesis, Carl von Ossietzky University, 2013.

[49] G. Hecht, N. Moha, and R. Rouvoy, "An empirical study of the performance impacts of android code smells," in *Proceedings of the International Workshop on Mobile Software Engineering and Systems*, ser. MOBILESoft '16. New York, NY, USA : ACM, 2016, pp. 59–69. [Online]. Available : http://doi.acm.org/10.1145/2897073.2897100

[50] "Android API : ArrayMap," https://developer.android.com/reference/android/support/v4/util/ArrayMap.html, [Online ; accessed 18th-May-2017].

[51] "Android performance tips," https://developer.android.com/training/articles/perf-tips.html, June 2016.

[52] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A. Le Meur, "Decor : A method for the specification and detection of code and design smells," *Software Engineering, IEEE Transactions on*, vol. 36, no. 1, pp. 20–36, 2010.

[53] J. M. Chambers, *Graphical Methods for Data Analysis*, 1st ed. Wadsworth International Group, 1983.

[54] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering : Trends, techniques and applications," *ACM Comput. Surv.*, vol. 45, no. 1, pp. 11 :1–11 :61, Dec. 2012. [Online]. Available : http://doi.acm.org/10.1145/2379776.2379787

[55] R. Morales, F. Chicano, F. Khomh, and G. Antoniol, "Exact search-space size for the refactoring scheduling problem," *Automated Software Engineering Journal*, 2017. [Online]. Available : http://dx.doi.org/10.1007/s10515-017-0213-6

[56] S. Kirkpatrick, "Optimization by simulated annealing : Quantitative studies," *Journal of statistical physics*, vol. 34, no. 5-6, pp. 975–986, 1984.

[57] D. Whitley, "A genetic algorithm tutorial," *Statistics and computing*, vol. 4, no. 2, pp. 65–85, 1994.

[58] ——, "An overview of evolutionary algorithms : practical issues and common pitfalls," *Information and software technology*, vol. 43, no. 14, pp. 817–831, 2001.

[59] N. Mladenović and P. Hansen, "Variable neighborhood search," *Computers & Operations Research*, vol. 24, no. 11, pp. 1097–1100, 1997.

[60] D. F. Lochtefeld and F. W. Ciarallo, "Multi-objectivization via decomposition : An analysis of helper-objectives and complete decomposition," *European Journal of Operational Research*, vol. 243, no. 2, pp. 395 – 404, 2015. [Online]. Available : http://www.sciencedirect.com/science/article/pii/S0377221714009916

[61] J. Knowles, L. Thiele, and E. Zitzler, "A tutorial on the performance assessment of stochastic multiobjective optimizers," *Tik report*, vol. 214, pp. 327–332, 2006.

[62] K. Deb, *Multi-objective optimization using evolutionary algorithms*. John Wiley & Sons, 2001, vol. 16.

[63] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm : Nsga-ii," *Evolutionary Computation, IEEE Transactions on*, vol. 6, no. 2, pp. 182–197, 2002.

[64] E. Zitzler, M. Laumanns, L. Thiele, E. Zitzler, E. Zitzler, L. Thiele, and L. Thiele, "SPEA2 : Improving the strength pareto evolutionary algorithm," 2001.

[65] A. J. Nebro, J. J. Durillo, F. Luna, B. Dorronsoro, and E. Alba, "MOCell : A cellular genetic algorithm for multiobjective optimization," *International Journal of Intelligent Systems*, pp. 25–36, 2007.

[66] ——, "Design issues in a multiobjective cellular genetic algorithm," in *Proceeding of the Conference on Evolutionary Multi-Criterion Optimization, volume 4403 of LNCS*. Springer, 2007, pp. 126–140.

[67] B. W. Boehm, J. R. Brown, and H. Kaspar, *Characteristics of software quality*. North-Holland, 1978.

[68] R. G. Dromey, "Cornering the chimera [software quality]," *IEEE Software*, vol. 13, no. 1, pp. 33–43, 1996.

[69] M. O'Keeffe and M. O. Cinneide, "Getting the most from search-based refactoring," in *Gecco 2007 : Genetic and Evolutionary Computation Conference, Vol 1 and 2*, 2007, Journal Article, pp. 1114–1120.

[70] R. Marinescu, "Detection strategies : Metrics-based rules for detecting design flaws," in *IEEE Int'l Conference on Software Maintenance, ICSM*. IEEE Computer Society, 2004, Conference Proceedings, pp. 350–359.

[71] M. Munro, "Product metrics for automatic identification of "bad smell" design problems in java source-code," in *Software Metrics, 2005. 11th IEEE International Symposium*, Sep. 2005, pp. 15–15.

[72] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "A bayesian approach for the detection of code and design smells," in *Quality Software, 2009. QSIC'09. 9th International Conference on*. IEEE, 2009, pp. 305–314.

[73] R. Marinescu, G. Ganea, and I. Verebi, "Incode : Continuous quality assessment and improvement," in *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*, March 2010, pp. 274–275.

[74] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Mining version histories for detecting code smells," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 462–489, 2015.

[75] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, "Lightweight detection of android-specific code smells : The adoctor project," in *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*. IEEE, 2017, pp. 487–491.

[76] I. H. Moghadam and M. O. Cinneide, "Code-imp : A tool for automated search-based refactoring," in *Proceedings of the 4th Workshop on Refactoring Tools*. IEEE Computer Society, 2011, Conference Proceedings, pp. 41–44.

[77] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, and M. Ó Cinnéide, "Recommendation system for software refactoring using innovization and interactive dynamic optimization," in *Proceedings of the 29th ACM/IEEE Int'l Conf. on Automated software engineering*. ACM, 2014, pp. 331–336.

[78] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Jdeodorant : identification and application of extract class refactorings," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 1037–1039.

[79] M. Fokaefs, N. Tsantalis, and A. Chatzigeorgiou, "Jdeodorant : Identification and removal of feature envy bad smells," in *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, 2007, pp. 519–520.

[80] N. Tsantalis and A. Chatzigeorgiou, "Identification of extract method refactoring opportunities for the decomposition of methods," *Journal of Systems and Software*, vol. 84, no. 10, pp. 1757–1782, 2011.

[81] T. Mens, G. Taentzer, and O. Runge, "Analysing refactoring dependencies using graph transformation," *Software and Systems Modeling*, vol. 6, no. 3, pp. 269–285, 2007.

[82] H. Liu, G. Li, Z. Y. Ma, and W. Z. Shao, "Conflict-aware schedule of software refactorings," *IET Software*, vol. 2, no. 5, p. 446, 2008.

[83] H. Liu, Z. Ma, W. Shao, and Z. Niu, "Schedule of bad smell detection and resolution : A new way to save effort," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 220–235, 2012.

[84] S. Bouktif, G. Antoniol, E. Merlo, and M. Neteler, "A novel approach to optimize clone refactoring activity," in *Proceedings of the 8th annual conference on Genetic and evolutionary computation*. ACM, 2006, pp. 1885–1892.

[85] S. Lee, G. Bae, H. S. Chae, D. Bae, and Y. R. Kwon, "Automated scheduling for clone-based refactoring using a competent ga," *Software : Practice and Experience*, vol. 41, no. 5, pp. 521–550, 2011.

[86] M. F. Zibran and C. K. Roy, "A constraint programming approach to conflict-aware optimal scheduling of prioritized code clone refactoring," in *2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*, 2011, Conference Proceedings, pp. 105–114.

[87] I. H. Moghadam and M. O. Cinnéide, "Resolving conflict and dependency in refactoring to a desired design," *e-Informatica Software Engineering Journal*, vol. 9, no. 1, 2015.

[88] M. Kersten and G. C. Murphy, "Using task context to improve programmer productivity," in *Proceedings of the 14th ACM SIGSOFT/FSE*, 2006, pp. 1–11.

[89] R. Robbes and M. Lanza, "Improving code completion with program history," *Automated Software Engineering*, vol. 17, no. 2, pp. 181–212, June 2010.

[90] S. Lee, S. Kang, S. Kim, and M. Staats, "The impact of view histories on edit recommendations," *Software Engineering, IEEE Transactions on*, vol. 41, no. 3, pp. 314–330, March 2015.

[91] H. Sanchez, R. Robbes, and V. M. Gonzalez, "An empirical study of work fragmentation in software evolution tasks," in *Proceedings SANER*, 2015, pp. 251–260.

[92] A. Ying and M. Robillard, "The influence of the task on programmer behaviour," in *Proceedings ICPC*, june 2011, pp. 31–40.

[93] F. Zhang, F. Khomh, Y. Zou, and A. E. Hassan, "An empirical study of the effect of file editing patterns on software quality," in *Proceedings WCRE*, 2012, pp. 456–465.

[94] Z. Soh, F. Khomh, Y.-G. Gueheneuc, G. Antoniol, and B. Adams, "On the effect of program exploration on maintenance tasks," in *Reverse Engineering (WCRE), 2013 20th Working Conference on*, Oct 2013, pp. 391–400.

[95] B. Beizer, *Software Testing Techniques 2nd edition.* International Thomson Computer Press, 1990.

[96] M. Bruntink and A. van Deursen, "An empirical study into class testability," *J. Syst. Softw.*, vol. 79, no. 9, pp. 1219–1232, Sep. 2006. [Online]. Available : http://dx.doi.org/10.1016/j.jss.2006.02.036

[97] A. Sabane, M. Di Penta, G. Antoniol, and Y.-G. Guéhéneuc, "A study on the relation between antipatterns and the cost of class unit testing," in *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on.* IEEE, 2013, pp. 167–176.

[98] P. McMinn and M. Holcombe, "Evolutionary testing of state-based programs," in *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '05. New York, NY, USA : ACM, 2005, pp. 1013–1020. [Online]. Available : http://doi.acm.org/10.1145/1068009.1068182

[99] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Trans. Software Eng.*, vol. 39, no. 2, pp. 276–291, 2013.

[100] I. Bashir and A. L. Goel, *Testing Object-Oriented Software : Life-Cycle Solutions*, 1st ed. Secaucus, NJ, USA : Springer-Verlag New York, Inc., 2000.

[101] C. Boyapati, S. Khurshid, and D. Marinov, "Korat : Automated testing based on java predicates," in *ACM SIGSOFT Software Engineering Notes*, vol. 27, no. 4. ACM, 2002, pp. 123–133.

[102] T. S. Chow, "Testing software design modeled by finite-state machines," *IEEE transactions on software engineering*, no. 3, pp. 178–187, 1978.

[103] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper, "Testability transformation," *IEEE Trans. Softw. Eng.*, vol. 30, no. 1, pp. 3–16, Jan. 2004. [Online]. Available : http://dx.doi.org/10.1109/TSE.2004.1265732

[104] M. Harman, "Refactoring as testability transformation," in *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, ser. ICSTW '11. Washington, DC, USA : IEEE Computer Society, 2011, pp. 414–421. [Online]. Available : http://dx.doi.org/10.1109/ICSTW.2011.38

[105] M. Linares-Vásquez, S. Klock, C. McMillan, A. Sabane, D. Poshyvanyk, and Y.-G. Guéhéneuc, "Domain matters : bringing further evidence of the relationships among anti-patterns, application domains, and quality-related metrics in Java mobile apps," in *Proceedings of the 22nd International Conference on Program Comprehension*, C. K. Roy, A. Begel, and L. Moonen, Eds. ACM, 2014, pp. 232–243.

[106] D. Verloop, *Code Smells in the Mobile Applications Domain*. TU Delft, Delft University of Technology, 2013.

[107] M. Fokaefs, N. Tsantalis, and A. Chatzigeorgiou, "Jdeodorant : Identification and removal of feature envy bad smells," in *ICSM*, 2007, pp. 519–520.

[108] J. Reimann, M. Brylski, and U. Aßmann, "A tool-supported quality smell catalogue for android developers." *Softwaretechnik-Trends*, vol. 34, no. 2, 2014.

[109] J. Reimann, M. Seifert, and U. Aßmann, "On the reuse and recommendation of model refactoring specifications," *Software and System Modeling*, vol. 12, no. 3, pp. 579–596, 2013. [Online]. Available : http://dx.doi.org/10.1007/s10270-012-0243-2

[110] G. Hecht, B. Omar, R. Rouvoy, N. Moha, and L. Duchien, "Tracking the Software Quality of Android Applications along their Evolution," in *30th IEEE/ACM International Conference on Automated Software Engineering*, ser. Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE 2015), L. Grunske and M. Whalen, Eds. Lincoln, Nebraska, United States : IEEE, Nov. 2015, p. 12. [Online]. Available : https://hal.inria.fr/hal-01178734

[111] A. Carette, M. A. A. Younes, G. Hecht, N. Moha, and R. Rouvoy, "Investigating the energy impact of android smells," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Feb 2017, pp. 115–126.

[112] K. Aggarwal, A. Hindle, and E. Stroulia, "GreenAdvisor : A tool for analyzing the impact of software evolution on energy consumption," in *ICSME*. IEEE, 2015, pp. 311–320.

[113] I. Polato, D. Barbosa, A. Hindle, and F. Kon, "Hybrid HDFS : decreasing energy consumption and speeding up hadoop using ssds," *PeerJ PrePrints*, vol. 3, p. e1320, 2015. [Online]. Available : http://dx.doi.org/10.7287/peerj.preprints.1320v1

[114] C. Pang, A. Hindle, B. Adams, and A. E. Hassan, "What do programmers know about the energy consumption of software ?" *PeerJ PrePrints*, vol. 3, p. e886, 2015.

[115] C. Zhang, A. Hindle, and D. M. Germán, "The impact of user choice on energy consumption," *IEEE Software*, vol. 31, no. 3, pp. 69–75, 2014. [Online]. Available : http://dx.doi.org/10.1109/MS.2014.27

[116] K. Rasmussen, A. Wilson, and A. Hindle, "Green mining : energy consumption of advertisement blocking methods." in *GREENS*, 2014, pp. 38–45.

[117] A. Hindle, A. Wilson, K. Rasmussen, E. J. Barlow, J. C. Campbell, and S. Romansky, "Greenminer : A hardware based mining software repositories software energy consumption framework," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014, 2014, pp. 12–21.

[118] C. Sahin, L. L. Pollock, and J. Clause, "How do code refactorings affect energy usage ?" in *International Symposium on Empirical Software Engineering and Measurement, ESEM*, 2014, pp. 36 :1–36 :10.

[119] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, "Mining Energy-greedy API Usage Patterns in Android Apps : An Empirical Study," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA : ACM, 2014, pp. 2–11. [Online]. Available : http://doi.acm.org/10.1145/2597073.2597085

[120] A. Pathak, Y. C. Hu, and M. Zhang, "Where is the energy spent inside my app ? : fine grained energy accounting on smartphones with eprof," in *EuroSys*, P. Felber, F. Bellosa, and H. Bos, Eds. ACM, 2012, pp. 29–42.

[121] S. Hao, D. Li, W. G. Halfond, and R. Govindan, "Estimating mobile application energy consumption using program analysis," in *Software Engineering (ICSE), 2013 35th International Conference on*. IEEE, 2013, pp. 92–101.

[122] W. G. P. da Silva, L. Brisolara, U. B. Correa, and L. Carro., "Evaluation of the impact of code refactoring on embedded software efficiency," in *In Proceedings of the 1st Workshop de Sistemas Embarcados*. Bonn : Gesellschaft für Informatik, 2010, pp. 145–150.

[123] C. Sahin, F. Cayci, I. L. M. Gutiérrez, J. Clause, F. E. Kiamilev, L. L. Pollock, and K. Winbladh, "Initial explorations on design pattern energy usage." in *GREENS*. IEEE, 2012, pp. 55–61.

[124] G. Pinto, *A Refactoring Approach to Improve Energy Consumption of Parallel Software Systems*. Informatics Center, Federal University of Pernambuco, 2015.

[125] D. Li, A. H. Tran, and W. G. J. Halfond, "Making web applications more energy efficient for oled smartphones," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA : ACM, 2014, pp. 527–538. [Online]. Available : http://doi.acm.org/10.1145/2568225.2568321

[126] M. Wan, Y. Jin, D. Li, and W. G. J. Halfond, "Detecting Display Energy Hotspots in Android Apps," in *Proceedings of the 8th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, Apr. 2015.

[127] B. R. Bruce, J. Petke, and M. Harman, "Reducing energy consumption using genetic improvement," in *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*. ACM, 2015, pp. 1327–1334.

[128] I. Manotas, L. Pollock, and J. Clause, "SEEDS : A Software Engineer's Energy-optimization Decision Support Framework," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA : ACM, 2014, pp. 503–514. [Online]. Available : http://doi.acm.org/10.1145/2568225.2568297

[129] "Inner class example," https://docs.oracle.com/javase/tutorial/java/javaOO/innerclasses.html, accessed : 2015-06-03.

[130] I. H. Moghadam and M. O. Cinneide, "Automated refactoring using design differencing," in *Software Maintenance and Reengineering (CSMR), 16th European Conference on*, ser. Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR. IEEE Computer Society, 2012, pp. 43 – 52.

[131] "Mylyn wiki," http://wiki.eclipse.org/Mylyn, accessed : 2016-02-23.

[132] L. M. Layman, L. A. Williams, and R. St Amant, "Mimec : intelligent user notification of faults in the eclipse ide," in *Proceedings of the 2008 international workshop on Cooperative and human aspects of software engineering*. ACM, 2008, pp. 73–76.

[133] "Mylyn task-focused interface," http://help.eclipse.org/juno/index.jsp?topic=%2Forg.eclipse.mylyn.help.ui%2FMylyn%2FUser_Guide%2FTask-Focused-Interface.html, accessed : 2016-02-23.

[134] Z. Soh, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol, "Towards understanding how developers spend their effort during maintenance activities," in *Reverse Engineering (WCRE), 2013 20th Working Conference on*, Oct 2013, pp. 152–161.

[135] E. Van Emden and L. Moonen, "Java quality assurance by detecting code smells," in *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*. IEEE, 2002, pp. 97–106.

[136] M. Lanza and R. Marinescu, *Object-oriented metrics in practice : using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media, 2007.

[137] "Inner class example," https://docs.oracle.com/javase/tutorial/java/javaOO/innerclasses.html, accessed : 2015-06-03.

[138] C. Blum and A. Roli, "Metaheuristics in combinatorial optimization : Overview and conceptual comparison," *ACM Comput. Surv.*, vol. 35, no. 3, pp. 268–308, Sep. 2003. [Online]. Available : http://doi.acm.org/10.1145/937503.937505

[139] D. Romano, S. Raemaekers, and M. Pinzger, "Refactoring fat interfaces using a genetic algorithm," Delft University of Technology, Software Engineering Research Group, Tech. Rep., 2014.

[140] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[141] J. Cohen, *Statistical power analysis for the behavioral sciences (rev.* Lawrence Erlbaum Associates, Inc, 1977.

[142] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, and M. S. Hamdi, "Improving multi-objective code-smells correction using development history," *Journal of Systems and Software*, vol. 105, no. 0, pp. 18 – 39, 2015.

[143] R. K. Yin, *Case Study Research : Design and Methods - Third Edition*, 3rd ed. SAGE Publications, 2002.

[144] R. Morales, A. Sabane, P. Musavi, F. Khomh, F. Chicano, and G. Antoniol, "Finding the best compromise between design quality and testing effort during refactoring," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, 2016, Conference Proceedings, pp. 24–35.

[145] R. Morales, Z. Soh, F. Khomh, G. Antoniol, and F. Chicano, "On the use of developers'context for automatic refactoring of software anti-patterns," *Journal of Systems and Software*, vol. 128, pp. 236 – 251, 2017.

[146] A. Lluch-Lafuente, S. Edelkamp, and S. Leue, "Partial order reduction in directed model checking," in *International SPIN Workshop on Model Checking of Software*. Springer, 2002, pp. 112–127.

[147] J. H. Holland, *Adaptation in natural and artificial systems : an introductory analysis with applications to biology, control, and artificial intelligence.* U Michigan Press, 1975.

[148] M. Dorigo, V. Maniezzo, and A. Colorni, "Ant system : optimization by a colony of cooperating agents," *Systems, Man, and Cybernetics, Part B : Cybernetics, IEEE Transactions on*, vol. 26, no. 1, pp. 29–41, 2006.

[149] Y.-G. Gueheneuc and H. Albin-Amiot, "Recovering binary class relationships : Putting icing on the uml cake," *ACM SIGPLAN Notices*, vol. 39, no. 10, pp. 301–314, 2004.

[150] Y.-G. Guéhéneuc and G. Antoniol, "Demima : A multi-layered framework for design pattern identification," *Software Engineering, IEEE Transactions on*, vol. 34, no. 35, pp. 667–684, Sep 2008.

[151] D. Knuth, *The Art of Computer Programming, Volume 4A : Combinatorial Algorithms*, ser. algorithms. Pearson Education, 2014, no. pt. 1.

[152] F. Khomh, M. D. Penta, Y.-G. Gueheneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change- and fault-proneness," *Empirical Softw. Engg.*, vol. 17, no. 3, pp. 243–275, Jun. 2012.

[153] M. Hollander, D. A. Wolfe, and E. Chicken, *Nonparametric statistical methods*. John Wiley & Sons, 2013.

[154] N. Cliff, *Ordinal methods for behavioral data analysis*. Psychology Press, 2014.

[155] J. Romano, J. D. Kromrey, J. Coraggio, J. Skowronek, and L. Devine, "Exploring methods for evaluating group differences on the nsse and other surveys : Are the t-test and cohens'd indices the most appropriate choices," in *annual meeting of the Southern Association for Institutional Research*, 2006.

[156] Y.-G. Guéhéneuc, "Ptidej : Promoting patterns with patterns," in *Proceedings of the 1st ECOOP workshop on Building a System using Patterns. Springer-Verlag*, 2005.

[157] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Addison-Wesley, 2011.

[158] R. Morales, F. Chicano, F. Khomh, and G. Antoniol. (2017) RePOR replication package. [Online]. Available : http://www.swat.polymtl.ca/rmorales/jss_repor/

[159] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *Software Engineering, IEEE Transactions on*, vol. 33, no. 1, pp. 2–13, 2007.

[160] P. Bourque, R. E. Fairley *et al.*, *Guide to the Software Engineering Body of Knowledge (SWEBOK (R)) : Version 3.0.* IEEE Computer Society Press, 2014.

[161] Y. Le Traon, T. Jéron, J.-M. Jézéquel, and P. Morel, "Efficient object-oriented integration and regression testing," *Reliability, IEEE Transactions on*, vol. 49, no. 1, pp. 12–25, 2000.

[162] B. Baudry and Y. Le Traon, "Measuring design testability of a uml class diagram," *Information and software technology*, vol. 47, no. 13, pp. 859–879, 2005.

[163] R. Binder, *Testing object-oriented systems : models, patterns, and tools.* Addison-Wesley Professional, 2000.

[164] J. J. Durillo and A. J. Nebro, "jmetal : A java framework for multi-objective optimization," *Advances in Engineering Software*, vol. 42, pp. 760–771, 2011.

[165] A. Jaszkiewicz, "A comparative study of multiple-objective metaheuristics on the bi-objective set covering problem and the pareto memetic algorithm," *Annals of Operations Research*, vol. 131, no. 1-4, pp. 135–158, 2004.

[166] E. Zitzler and L. Thiele, "Multiobjective evolutionary algorithms : a comparative case study and the strength pareto approach," *evolutionary computation, IEEE transactions on*, vol. 3, no. 4, pp. 257–271, 1999.

[167] G. Anthes, "Invasion of the mobile apps," *Commun. ACM*, vol. 54, no. 9, pp. 16–18, Sep. 2011. [Online]. Available : http://doi.acm.org/10.1145/1995376.1995383

[168] J. Voas, J. B. Michael, and M. van Genuchten, "The mobile software app takeover," *Software, IEEE*, vol. 29, no. 4, pp. 25–27, July 2012.

[169] D. L. Parnas, "Software aging," in *ICSE '94 : Proc. of the 16th Int'l conference on Software engineering.* IEEE Computer Society Press, 1994, pp. 279–287.

[170] M. Gottschalk, J. Jelschen, and A. Winter, "Energy-efficient code by refactoring," in *Softwaretechnik Trends*, vol. 33, no. 2. Bonn : Gesellschaft für Informatik, 05 2013, pp. 23–24.

[171] S. E. S. Taba, F. Khomh, Y. Zou, A. E. Hassan, and M. Nagappan, "Predicting bugs using antipatterns," in *Proc. of the 29th Int'l Conference on Software Maintenance*, 2013, pp. 270–279.

[172] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells : A case study of two open source systems," in *3rd Int'l Symposium on Empirical Software Engineering and Measurement, ESEM 2009,*, 2009, pp. 390–400.

[173] J. J. Park, J. Hong, and S. Lee, "Investigation for software power consumption of code refactoring techniques," in *The 26th International Conference on Software Engineering and Knowledge Engineering, Hyatt Regency, Vancouver, BC, Canada, July 1-3, 2013.*, 2014, pp. 717–722.

[174] A. Ouni, M. Kessentini, H. Sahraoui, and M. S. Hamdi, "The use of development history in software refactoring using a multi-objective evolutionary algorithm," pp. 1461–1468, 2013.

[175] D. Singh and W. J. Kaiser, "The atom leap platform for energy-efficient embedded computing," *Center for Embedded Network Sensing*, 2010.

[176] D. Li, S. Hao, J. Gui, and W. G. J. Halfond, "An Empirical Study of the Energy Consumption of Android Applications," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2014.

[177] R. Saborido, V. Arnaoudova, G. Beltrame, F. Khomh, and G. Antoniol, "On the impact of sampling frequency on software energy measurements," *PeerJ PrePrints*, vol. 3, p. e1219, 2015. [Online]. Available : http://dx.doi.org/10.7287/peerj.preprints.1219v2

[178] C. Sahin, L. Pollock, and J. Clause, "From Benchmarks to Real Apps : Exploring the Energy Impacts of Performance-directed Changes," *Journal of Systems and Software*, pp. –, 2016. [Online]. Available : http://www.sciencedirect.com/science/article/pii/S0164121216000893

[179] A. R. Tonini, L. M. Fischer, J. C. B. de Mattos, and L. B. de Brisolara, "Analysis and evaluation of the android best practices impact on the efficiency of mobile applications," in *Computing Systems Engineering (SBESC), 2013 III Brazilian Symposium on*. IEEE, 2013, pp. 157–158.

[180] "Monkey runner concepts," https://developer.android.com/studio/test/monkeyrunner/index.html, [Online ; accessed 18th-May-2017].

[181] "Debugging Android apps," https://developer.android.com/reference/android/os/Debug.html, [Online ; accessed 18th-May-2017].

[182] "Android API guides : Location strategies," https://developer.android.com/guide/topics/location/strategies.html, [Online ; accessed 18th-May-2017].

[183] B. L. Miller and D. E. Goldberg, "Genetic algorithms, tournament selection, and the effects of noise," *Complex systems*, vol. 9, no. 3, pp. 193–212, 1995.

[184] K. Miettinen, *Nonlinear Multiobjective Optimization*. Springer US, 2012. [Online]. Available : https://books.google.ca/books?id=bnzjBwAAQBAJ

[185] C. Sahin, M. Wan, P. Tornquist, R. McKenna, Z. Pearson, W. G. Halfond, and J. Clause, "How does code obfuscation impact energy usage ?" *Journal of Software : Evolution and Process*, 2016.

[186] A. Banerjee and A. Roychoudhury, "Automated re-factoring of android apps to enhance energy-efficiency," in *Proceedings of the International Workshop on Mobile Software Engineering and Systems*, ser. MOBILESoft '16. New York, NY, USA : ACM, 2016, pp. 139–150. [Online]. Available : http://doi.acm.org/10.1145/2897073.2897086

[187] "Android API guides : Broadcasts," https://developer.android.com/guide/components/broadcasts.html, [Online ; accessed 18th-May-2017].

[188] K. Mao, M. Harman, and Y. Jia, "Robotic testing of mobile apps for truly black-box automation," *IEEE Software*, vol. 34, no. 2, pp. 11–16, 2017.

[189] W. Martin, M. Harman, Y. Jia, F. Sarro, and Y. Zhang, "The app sampling problem for app store mining," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR '15. Piscataway, NJ, USA : IEEE Press, 2015, pp. 123–133. [Online]. Available : http://dl.acm.org/citation.cfm?id=2820518.2820535

[190] L. Moonen, G. Kok *et al.*, "Refactoring test code," *Software Engineering [SEN]*, no. R 0119, pp. 1–6, 2001.

[191] M. Tufano, F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, A. D. Lucia, and D. Poshyvanyk, "An empirical investigation into the nature of test smells," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Sept 2016, pp. 4–15.