A Systematic Literature Review on Software Refactoring

by

Jallal Elhazzat

A thesis submitted in partial fulfillment of the
requirements for the degree of
Master of Science
(Software Engineering)
in The University of Michigan-Dearborn
2020

Master's Thesis Committee:

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


DEARBORN


ISELab
Intelligent Software Engineering

Table of Contents

List of Tables

Table

List of Figures

Figure

List of Abbreviations

IDE Integrated Development Environment

SLR Systematic Literature Review

IGA Interactive Genetic Algorithm

GA Classic Genetic Algorithm

Abstract

Due to the growing complexity of software systems, there has been a dramatic increase in research and industry demand on refactoring. Refactoring research nowadays addresses challenges beyond code transformation to include, but not limited to, scheduling the opportune time to carry refactoring, recommending specific refactoring activities, detecting refactoring opportunities and testing the correctness of applied refactoring.

Very few studies focused on the challenges that practitioners face when refactoring software systems and what should be the current refactoring research focus from the developers' perspective and based on the current literature. Without such knowledge, tool builders invest in the wrong direction, and researchers miss many opportunities for improving the practice of refactoring.

In this thesis, we collected papers from several publication sources and analyzed them to identify what do developers ask about refactoring and the relevant topics in the field We found that developers and researchers are asking about design patterns, design and user interface refactoring, web services, parallel programming, and mobile apps. We also identified what popular refactoring challenges are the most difficult and the current important topics and questions related to refactoring. Moreover, we discovered gaps between existing research on refactoring and the challenges developers face.

Chapter 1: Introduction

A recent study [1] by the US Air Force Software Technology Support Centre (STSC) shows that the code restructuring of several software systems reduced developers' time by over 60% when introducing new features into a restructured architecture. General Motors (GM) is recalling nearly 4.3 million vehicles in 2017 after discovering a software quality defect of poor modularity in an evolved program in a car controller. It caused performance issues that prevented air bags from deploying in time during a crash [2]. That flow has already been linked to one death and three injuries.

Clearly, urgently, software engineers need better ways to reduce and manage the growing complexity of software systems and improve their productivity. Refactoring [3, 4, 5] is a technique that improves the design structure while preserving the overall functionality and behavior. Refactoring is a key practice in agile development processes and is well supported by refactoring tools that are standard with all major IDEs. Refactoring is an extremely important solution to address the challenge of managing software complexity [6, 7, 8], and has experienced tremendous adoption in Object-oriented systems [9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19].

Evolution is a characteristic of software which means modifying the software to adapt new requirements and to incorporate new features. These modification over time can degrade the software quality and increase the complexity of code leading to higher costs of development and maintenance. Therefore, there is a need of techniques to improve the quality and reduce the complexity of the software.

The research area for this purpose is called restructuring or in case of an object-oriented

environment, Refactoring. Martin Fowler defines Refactoring as ”a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior” [3]. This implies that refactoring is a method which reconstruct the code’s structure without altering its behavior in order to improve the software quality in terms of maintainability, extensibility, and reusability. Refactoring typically consists of small steps after each the functionality of the code will be unchanged. Refactoring can be done in various areas of the software: Code, Database, or User interface. However, we aim to focus on code refactoring. It might be difficult for a developer to be justified to spend time on improvement of a piece of code in order to have the same exact functionality. However, it can be seen as an investment for future developments. Specifically, refactoring is an imperative task on software’s with longer lifespans with multiple developers need to read and understand the codes. Refactoring can improve both the quality of software and the productivity of its developers. Increasing the quality of software is due to decreasing the complexity of it at design and source code level caused by refactoring which is proved by many studies [20, 21]. The long-term effect of refactoring is improving the productivity of developers by increasing two crucial factors, understandability and maintainability of the codes, especially when a new developer join to an existing project. It is shown that refactoring can help to detect, fix and reduce software bugs and leading to software projects which are less likely to expose bug in development process [22]. Another study claims that there are some specific kinds of refactoring methods that are very probable to induce bug fixes [23].

Refactoring is a way of removing or reducing the presence of technical debt. Technical debt is a concept analogous to financial credit and it consists of code, design, test, and documentation debts. In software engineering world, it implies extra efforts and costs caused by an improper design or code structure. This can be seen more dramatically in large and long-lived software systems.

Technical debt can be managed by increasing awareness, detecting and repaying, and preventing accumulation of it. Refactoring is the best strategy to cope with technical debt before it gets out of control. Refactoring is beneficial to keep technical debt low and can be more efficient when it is automated [24].Critical systems are those in which failure results in significant physical damages, economic disasters, or threats to human life. There are three types of critical systems: safety, mission, and business critical systems. Examples of these systems are automotive industry, spacecraft navigation systems, and banking. Regular changes are inevitable in software-critical systems, therefore refactoring plays a crucial role. It is shown that refactoring can improve the overall security of safety-critical system [25].

Software design is a human activity that cannot be fully automated because designers understand the problem domain intuitively and they have targeted design goals in mind. Thus, several studies show that fully automated refactoring does not always lead to the desired architecture [26]. On the other hand, manual refactoring is error-prone, time consuming and not practical for radical changes. Based on interviews that we conducted as part of an NSF I-Corps project, programmers spend an average of 45% of their overall development time manually applying refactoring. Batory *et al*. [27] presented several case studies where architectural refactoring involved more than 750 refactoring steps and took more than 3 weeks to execute. Thus, it is important to develop intelligent methods to determine when and how to integrate programmer feedback to semi-automate architecture refactoring.

Due to the growing complexity of software systems, there has been a dramatic increase in research and industry demand on refactoring. Refactoring research nowadays addresses challenges beyond code transformation to include, but not limited to, scheduling the opportune time to carry refactoring, recommending specific refactoring activities, detecting refactoring opportunities and

testing the correctness of applied refactoring. Very few studies focused on the challenges that practitioners face when refactoring software systems and what should be the current refactoring research focus from the developers' perspective and based on the current literature. Without such knowledge, tool builders invest in the wrong direction, and researchers miss many opportunities for improving the practice of refactoring. In this thesis, we collected papers from several publication sources and analyzed them to identify what do developers ask about refactoring and the relevant topics in the field. We found that developers and researchers are asking about design patterns, design and user interface refactoring, web services, parallel programming, and mobile apps. We also identified what popular refactoring challenges are the most difficult and the current important topics and questions related to refactoring. Moreover, we discovered gaps between existing research on refactoring and the challenges developers face. This thesis is organized as follows: Chapter II introduces our systematic literature review. A summary and future research directions are presented in chapter III.

Chapter 2: Systematic Literature Review on Software Refactoring

2.1 Related Concepts

2.1.1 Software Refactoring

Refactoring is defined as the process of improving the code after it has been written by changing its internal structure without changing its external behavior. The idea is to reorganize variables, classes and methods to facilitate future adaptations and enhance comprehension. This reorganization is used to improve different aspects of the software quality such as maintainability, extensibility, reusability, etc. Some modern Integrated Development Environments (IDEs), such as Eclipse, Netbeans, provide support for applying the most commonly used refactoring, e.g., move method, rename class, etc.

In order to identify which parts of the source code need to be refactored, most of the existing work relies on the notion of bad smells (e.g., Fowler's textbook [28]), also called design defects or anti-patterns. Typically, code smells refer to design situations that adversely affect the development of the software. When applying refactoring to fix design defects, software metrics can be used as an overall indication of the quality of the new design. For instance, high intra-class cohesion and low inter-class coupling usually indicate a high-quality system.

Refactoring is one of the most used terms in software development and has played a major role in the maintenance of software for decades. While most developers have an intuitive understanding of the refactoring process, many of us lack a true mastery which is an important skill. In this article, we will explore the textbook definition of refactoring, how this definition holds up to the reality of software development, and how we can ensure our codebase is prepared for refactoring. Along the

way, we will walk through an entire set of refactoring, from start to finish to illustrate the simplicity and importance of this ubiquitous process.

Refactoring is one of the most self-evident processes in software development, but it is surprisingly difficult to perform properly. In most cases, we deviate from strict refactoring and execute an approximation of the process; sometimes, things work out and we are left with cleaner code, but other times, we get snared, wondering where we went wrong. In either case, it is important to fully understand the importance and simplicity of barebones refactoring.

In short, the process of refactoring involves taking small, manageable steps that incrementally increase the cleanliness of code while still maintaining the functionality of the code. As we perform more and more of these small changes, we start to transform messy code into simpler, easier to read, and more maintainable code. It is not a single refactoring that makes the change: It's the cumulative effect of many small refactoring performed toward a single goal that makes the difference.

2.1.2 The Refactoring Operations

The refactoring operations considered in the approaches proposed in this thesis cover the most used operations selected from diff t categories: "Moving features", "Data organizers", "Method calls simplifiers", and "Generalization modification These refactoring are listed in Table 2.1. We selected these refactoring operations because they have the most impact on code quality attributes.

Table 2.1: List of refactoring operations included in this thesis.

Refactoring	**Controlling Parameter**
Moving Features Between Objects	
Move Method	Source, Target, Method
Move Field	Source, Target, Attribute
Extract Class	Source, Target, Attributes, Methods
Organizing Data	
Encapsulate Field	Source, Attribute
Simplifying Method Calls	
Decrease Field Security	Source, Attribute
Decrease Method Security	Source, Method
Increase Field Security	Source, Attribute
Increase Method Security	Source, Method
Dealing with Generalization	
Pull Up Field	Source, Target, Attribute
Pull Up Method	Source, Target, Method
Push Down Field	Source, Target, Attribute
Push Down Method	Source, Target, Method
Extract SubClass	Source, Target, Attributes, Methods
Extract SuperClass	Source, Target, Attributes, Methods

2.1.3 Code Quality Metrics

Many studies have utilized structural metrics as a basis for define quality indicators for a good system design [18, 51]. As an illustrative example, [29] proposed a set of quality measures, using the ISO 9126 specification, called QMOOD. This model is developed based on international standard for software product quality measurement. QMOOD is a comprehensive way to assess the software quality and includes four levels.

We employed the first two levels known as" Design Quality Attributes" and" Object-oriented Design Properties" to calculate our fitness functions used in this thesis

Reusability, Flexibility, Understandability, Functionality, Extendibility, Effectiveness, Complexity, Cohesion, Coupling). Each of these quality metrics is defined using a combination of low-level metrics as detailed in Tables 2.2 and 2.3.

Table 2.2: QMOOD design metrics.

Design Metric	**Design**	**Description**
Design Size in Classes (*DSC*)	Design Size	Total number of classes in the design.
Number of Hierarchies (*NOH*)	Hierarchies	Total number of" root" classes in the design (*count (MaxInheritenceTree (class)=0)*)
Average Number of Ancastors (*ANA*)	Abstraction	Average number of classes in the inheritance tree for each class.
Direct Access Metric (*DAM*)	Encapsulation	Ratio of the number of private and protected attributes to the total number of attributes in a class.
Direct Class Coupling (*DCC*)	Coupling	Number of other classes a class relates to, either through a shared attribute or a parameter in a method.
Cohesion Among Methods of class (*CAMC*)	Cohesion	Measure of how related methods are in a class in terms of used parameters. It can also be computed by: *1 − LackOf-CohesionOfMethods()*
Measure of Aggregation (*MOA*)	Composition	Count of number of attributes whose type is user defined class(es).
Measure of Functional Abstraction (*MFA*)	Inheritance	Ratio of the number of inherited methods per the total number of methods within a class.
Number of Polymorphic Methods (*NOP*)	Polymorphism	Any method that can be used by a class and its descendants. Counts of the number of methods in a class excluding private, static and final ones.
Class Interface Size	Messaging	Number of public methods in class.
Number of Methods (*NOM*)	Complexity	Number of methods declared in a class

Table 2.3: QMOOD quality attributes.

Quality attributes	**Definition** / **Computation**
Reusability	A design with low coupling and high cohesion is easily reused by other designs.
	$-0.25 * Coupling + 0.25 * Cohesion + 0.5 * Messaging + 0.5 * DesignSize$
Flexibility	The degree of allowance of changes in the design.
	$0.25 * Encapsulation - 0.25 * Coupling + 0.5 * Composition + 0.5 * Polymorphism$
Understandability	The degree of understanding and the easiness of learning the design implementation details.
	$0.33 * Abstraction + 0.33 * Encapsulation - 0.33 * Coupling + 0.33 * Cohesion - 0.33 * Polymorphism - 0.33 * Complexity - 0.33 * DesignSize$
Functionality	Classes with given functions that are publicly stated in interfaces to be used by others.
	$0.12 * Cohesion + 0.22 * Polymorphism + 0.22 * Messaging + 0.22 * DesignSize + 0.22 * Hierarchies$
Extendibility	Measurement of design's allowance to incorporate new functional requirements.
	$0.5 * Abstraction - 0.5 * Coupling + 0.5 * Inheritance + 0.5 * Polymorphism$
Effectiveness	Design efficiency in fulfilling the required functionality.
	$0.2 * Abstraction + 0.2 * Encapsulation + 0.2 * Composition + 0.2 * Inheritance + 0.2 * Polymorphism$

The QMOOD model has been used previously in the area of search-based software refactoring [30], [31] and so we use it to estimate the effect of the suggested refactoring solutions on software quality. QMOOD has the advantage that it defines six high-level design quality attributes (reusability, flexibility, understandability, functionality, extendibility, and effectiveness) that can be calculated using 11 lower level design metrics.

2.2 Classification of Refactoring Studies

2.2.1 Manual Refactoring

We start, this section, by summarizing existing manual approaches for software refactoring. In Fowler's book [3] a non-exhaustive list of low-level design problems in source code has been difficult For each type of code smell, a list of possible refactoring is suggested that can be applied by the developers. Du Bois *et al*. [32] start from the hypothesis that refactoring opportunities correspond to those that improve cohesion and coupling metrics, and use this to perform an optimal distribution of features over classes. They analyze how refactoring manipulate coupling and cohesion metrics, and how to identify refactoring opportunities that improve these metrics. However, this approach is limited to only certain refactoring types and a small number of quality metrics. Murphy-Hill *et al*. [33, 34] proposed several techniques and empirical studies to support refactoring activities. In [34, 35], the authors proposed new tools to assist software developers in applying refactoring such as selection assistant, box view, and refactoring annotation based on structural information and program analysis techniques.

Recently, Ge and Murphy-Hill [36] have proposed a new refactoring tool called Ghost Factor that allows the developer to transform code manually, but checks the correctness of the transformation automatically. Benefactor [37] and Witchdoctor [38] can detect manual refactoring and then complete them automatically. Tahvildari *et al*. [39] also propose a framework of object-oriented metrics used to suggest to the software developer refactoring opportunities to improve the quality of an object-oriented legacy system. Dig [40] proposes an interactive refactoring technique to improve the parallelism of software systems. However, the proposed approach did not consider learning from the developers' feedback and focused on making programs more parallel. Other contributions are based on rules that can be expressed as assertions (invariants, pre- and post-

conditions). All these techniques are more concerned around the correctness of manually applied refactoring rather than interactive recommendations. The use of invariants has been proposed to detect parts of the program that require refactoring [41]. In addition, Opdyke [4] has proposed the definition and use of pre- and post-conditions with invariants to preserve the behavior of the software when applying refactoring. Hence, behavior preservation is based on the verification/satisfaction of a set of pre- and post-condition. All these conditions are expressed as first-order logic constraints expressed over the elements of the program. To summarize, manual refactoring is a tedious task for developers that involves exploring the software system to find the best refactoring solution that improves the quality of the software and fix design defects.

2.2.2 Automated Refactoring

To automate refactoring activities, new approaches have been proposed. JDeodorant [42] is an automated refactoring tool implemented as an Eclipse plug-in that identifies certain types of design defect using quality metrics and then proposes a list of refactoring strategies to fix them. Search-based techniques [43] are widely studied to automate software refactoring and consider it as an optimization problem, where the goal is to improve the design quality of a system based mainly on a set of software metrics. The majority of existing work combines several metrics in a single fitness function to find the best sequence of refactoring. Seng *et al*. [44] have proposed a single-objective optimization approach using a genetic algorithm to suggest a list of refactoring to improve software quality. The work of O'Keeffe *et al*. [30] uses various local search-based techniques such as hill climbing and simulated annealing to provide an automated refactoring support. They use the QMOOD metrics suite [29] to evaluate the improvement in quality.

Kessentini *et al*. [45] have proposed single-objective combinatorial optimization using a genetic algorithm to find the best sequence of refactoring operations that improve the quality of the code by minimizing as much as possible the number of design defects detected in the source code. Kilic *et al*. [46] explore the use of a variety of population-based approaches to search-based parallel refactoring, finding that local beam search could find the best solutions. Harman et al. [47] have proposed a search-based approach using Pareto optimality that combines two quality metrics, CBO (coupling between objects) and SDMPC (standard deviation of methods per class), in two separate fitness functions. Ouni et al. [48] proposed also a multi-objective refactoring formulation that generates solutions to fix code smells. O´ Cinn´eide et al [49] have proposed a multi-objective search-based refactoring to conduct an empirical investigation to assess some structural metrics and to explore relationships between them. They have used a variety of search techniques (Pareto-optimal search, semi-random search) guided by a set of cohesion metrics.

The majority of existing multi-objective refactoring techniques propose as output a set of non-dominated refactoring solutions (the Pareto front) that fi a good trade-off between the considered maintainability objectives. This leaves it to the software developers to select the best solution from a set of possible refactoring solutions, which can be a challenging task as it is not natural for developers to express their preferences in terms of a fitness functions value. Thus, the exploration of the Pareto front is still performed manually, which limits the use of multi-objective search techniques to address software engineering problems. An intelligent exploration of the Pareto front is required to expand the applicability of multi-objective techniques for search-based software engineering problems.

In summary, developers should accept the entire refactoring solution and existing tools do not provide the flexibility to adapt the suggested solution in existing fully automated refactoring

techniques. Furthermore, existing automated refactoring tools execute the whole algorithm again to suggest new refactoring after a number of code changes are introduced by developers, rather than simply trying to update the proposed solutions based on the new code changes. While automation is important, it is essential to understand the points at which human oversight, intervention, and decision making should impact on automation. Human developers might reject changes made by any automated programming technique. Especially if they feel that they have little control, there will be a natural reluctance to trust and use the automated refactoring tool [50].

2.2.3 Interactive Refactoring

Interactive techniques have been generally introduced in the literature of Search-Based Software Engineering and especially in the area of software modularization. Hall *et al*. [51] treated software modularization as a constraint satisfaction problem. The idea of this work is to provide a baseline distribution of software elements using good design principles (e.g. minimal coupling and maximal cohesion) that will be refined by a set of corrections introduced interactively by the designer.

The approach, called SUMO (Supervised Re-modularization), consists of iteratively feeding domain knowledge into the re-modularization process. The process is performed by the designer in terms of constraints that can be introduced to refine the current modularizations. Initially, the system begins with generating a module dependency graph from an input system. This dependency is based on the correlation between software elements (coupling between methods, shared attributes etc.). Possible modularizations are then generated from the graph using multiple simulated authoritative decompositions. Then, using a clustering technique called Bunch, an initial set of clusters is generated that serves as an input to SUMO.

The SUMO algorithm provides a hypothesized modularization to the user, who will agree with some relations, and disagree with others. The user's corrections are then integrated into the modularization process, to generate a more satisfactory modularization. The SUMO algorithm does not necessarily rely on clustering techniques, but it can benefit from their output as a starting point for its refinement process.

Bavota *et al*. [52] presented the adoption of single objective interactive genetic algorithms in software re-modularization process. The main idea is to incorporate the user in the evaluation of the generated re-modularizations. Interactive Genetic Algorithms (IGAs) extend the Classic Genetic Algorithms (GAs) by partially or entirely involving the user in the determination of the solution's fitness function. The basic idea of the Interactive GA (IGA) is to periodically add a constraint to the GA such that some specific components shall be put in a given cluster among those created so far. Initially, the IGA evolves similarly to the non-interactive GA.

After a user-defined set of iterations, the individual with the highest fitness value is selected from the population set (in the case of single-objective GA) or from the fi front (in the case of multi-objective GA) and presented to the user. After analyzing the current modularization, the user provides feedback in terms of constraints dictating for example, that a specific element needs to be in the same cluster as another one. Although user feedback is important in guaranteeing convergence, it is essential not to overload the user by asking for a decision about all the current relationships between elements, especially for a large system.

Overall, the above existing studies of interactive re-modularization are limited to few types of refactoring such as moving classes between packages and splitting packages. Furthermore, the interaction mechanism is based on the manual evaluation of proposed re-modularization solutions which could be a time-consuming process. The proposed interactive re-modularization techniques are

also based on a mono-objective algorithm and did not consider multiple objectives when evaluating the solutions. A recent study [53] extended our previous work [54] to propose an interactive search based approach for refactoring recommendations. The developers have to specify a desired design at the architecture level then the proposed approach try to fix the relevant refactoring that can generate a similar design to the expected one. In our work, we do not consider the use of a desired design, thus developers are not required to manually modify the current architecture of the system to get refactoring recommendations. Furthermore, developers maybe interested to change the architecture mainly when they want to introduce an extensive number of refactoring that radically change the architecture to support new features.

Several possible levels of interaction are not considered by existing refactoring techniques. It is easy for developers to identify large classes or long methods that should be refactored, but they find it difficult in general, to locate a target class when applying a move method refactoring [55]. In addition, existing refactoring tools do not update their recommended refactoring solutions based on the software developer's feedback such as accepting, modifying or rejecting certain refactoring operations.

Furthermore, none of the above interactive studies considered reducing the interaction effort with developers which is an important step to improve the applicability of refactoring tools as highlighted in the survey with developers.

To address the above-mentioned limitations, we proposed in this proposal, a new way for software developers to refactor their software systems as a sequence of transformations based on diff t levels of interaction, implicit exploration of non-dominated refactoring solutions and dynamic adaptive ranking of the suggested refactoring.

2.2.4 Search Based Software Refactoring

Search-based techniques [43] are widely studied to automate software refactoring where the goal is to improve the design quality of a system based mainly on a set of software metrics. The majority of existing work combines several metrics in a single fitness function to find the best sequence of refactoring. Seng et al. [56] have pro- posed a single-objective optimization approach using a genetic algorithm to suggest a list of refactoring to improve software quality. The work of O'Keeffe et al. [30] uses various local search-based techniques such as hill climbing and simulated annealing to provide an automated refactoring support. They use the QMOOD metrics suite [29] to evaluate the improvement in quality. Kessentini et al. [45] have proposed single-objective combinatorial optimization using a genetic algorithm to fi the best sequence of refactoring operations that improve the quality of the code by minimizing as much as possible the number of design defects detected in the source code. Kilic et al. [46] explore the use of a variety of population-based approaches to search-based parallel refactoring, finding that local beam search could find the best solutions. Harman et al. [47] have proposed a search-based approach using Pareto optimality that combines two quality metrics, CBO (coupling between objects) and SDMPC (standard deviation of methods per class), in two separate fitness functions. Ouni et al. [48] proposed also a multi-objective refactoring formulation that generate solutions to find code smells. O'Cinn'eide et al[49] have proposed a multi-objective search base refactoring to conduct an empirical investigation to assess some structural metrics and to explore relationships between them. They have used a variety of search techniques (Pareto-optimal search, semi-random search) guided by a set of cohesion metrics. The majority of existing multi-objective refactoring techniques propose as output a set of non-dominated refactoring solutions (the Pareto front) that fi a good tradeoff between the considered maintainability objectives. This leaves it to the software developers to select the best solution from a set of possible refactoring solutions, which can

be a challenging task as it is not natural for developers to express their preferences in terms of a fitness functions value. Thus, the exploration of the Pareto front is still performed manually, which limits the use of multi-objective search techniques to address software engineering problems. An intelligent exploration of the Pareto front is required to expand the applicability of multi-objective techniques for search-based software engineering problems as addressed in this proposal.

2.2.5 Refactoring Recommendation

Much effort has been devoted to the definition of approaches supporting refactoring. One representative example is JDeodorant, the tool proposed by Tsantalis and Chatzigeorgiou [57]. Our paper is mostly related to approaches exploiting search-based techniques to identify refactoring opportunities, and our discussion focuses on them since the bot is based on multi-objective refactoring. We point the interested reader to the survey by Bavota *et al*. [58] for an overview of approaches supporting code refactoring.

O'Keeffe and Cinnéide [59] presented the idea of formulating the refactoring task as a search problem in the space of alternative designs, generated by applying a set of refactoring operations. Such a search is guided by a quality evaluation function based on eleven object-oriented design metrics that reflect refactoring goals. Harman and Tratt [60] were the fi to introduce the concept of Pareto optimality to search-based refactoring. They used it to combine two metrics, namely CBO (Coupling Between Objects) and SDMPC (Standard Deviation of Methods Per Class), into a fi function and showed its superior performance as compared to a mono-objective technique [60].

The two aforementioned works [59, 60] paved the way to several search-based approaches aimed at recommending refactoring operations [44, 61, 62, 63, 64, 65]. Several other studies proposed refactoring at the model level as well [66, 67, 68, 69, 70, 71, 72, 73]. A representative

example of these techniques is the recent work by Alizadeh et al. *et al*. [74], who proposed an interactive multi-criteria code refactoring approach to improve the QMOOD quality metrics while minimizing the number of refactoring. In our approach, we decided to rely on a simpler optimization algorithm by only considering the refactoring of recently changed files in other pull requests rather than the root-canal refactoring approach of Alizadeh et al. *et al*. [74].

2.2.6 Empirical Studies on Refactoring

Empirical studies on software refactoring mainly aim at investigating the refactoring habits of software developers and the relationship between refactoring and code quality. We only discuss studies reporting find relevant to our work. Murphy-Hill *et al*. [75] investigated how developers perform refactoring. Examples of the exploited datasets are usage data from 41 developers using the Eclipse environment and information extracted from versioning systems. Among their several fixes they show that developers often perform *floss refactoring;* namely, they interleave refactoring with other programming activities, confirming that refactoring is rarely performed in isolation. Kim *et al*. [76] present a survey of software refactoring with 328 Microsoft engineers. They show that the major obstacle of adopting many existing refactoring tools is their configuration and painful integration within their pipelines without disturbing developers with their current focus in terms of meeting deadlines and making regular code changes. Those finding stress out the need for refactoring bots that can be adopted for continuous integration without considerable configuration effort.

2.2.7 Software Bots

The design and implementation of software bots are still in its infancy with a significant focus on chatbots. For instance, Lebeuf et al. *et al*. [77, 78] discussed the potential of using chat bots in software engineering and how they can be helpful to increase collaborations between programmers. The authors also proposed a possible classification of potential benefits of using software bots in various domains, especially to improve the productivity of developers. An extensive empirical study of over 90 software bots was performed by Wessel et al.*et al*. [79] to provide a classification and taxonomy for them. They found that around 21 bots were actually tried on GitHub repositories and the dominant majority are around testing but without providing any code actions or recommendations to developers. The authors found that none of these bots provides explanations of their analysis which reduced the adoption by developers.

Some examples of regression testing bots include Travis CI and the bot designed by Urli et al. [80] to repair bugs. These tools did not open a new pull-request, but they are executed manually by the developers where they can check the recommended patches. Another bot related to quality assessment but not refactoring is Fix-it *et al*. [81]. It is mainly limited to a few types of code changes, mainly targeting dynamic analysis metrics.

Finally, Wyrich et al. *et al*. [82] proposed a vision paper to emphasize the importance of refactoring bots and motivates their potential use in practice. They proposed a prototype, not a complete bot, by running SonarQube to detect code smells. However, the work is still in its initial stage where refactoring is not recommended yet.

2.3 Summary of Systematic Literature Review on Refactoring

Due to the growing complexity of software systems, there has been a dramatic increase and

industry demand for tools and techniques on software refactoring in the last ten years, definition traditionally as a set of program transformations intended to improve the system design while preserving the behavior. Refactoring studies are expanded beyond code-level restructuring to be applied at diff t levels (architecture, model, requirements, etc.), adopted in many domains beyond the object-oriented paradigm (cloud computing, mobile, web, etc.), used in industrial settings and considered objectives beyond improving design to include other non-functional requirements (e.g., improve performance, security, etc.). Thus, challenges to be addressed by refactoring work are nowadays beyond code transformation to include, but not limited to, scheduling the opportune time to carry refactoring, recommendations of specific refactoring activities, detection of refactoring opportunities and testing the correctness of applied refactoring. Therefore, the refactoring research efforts are fragmented over several research communities, various domains, and diff t objectives. To structure the fitness and existing research results, we provide a systematic literature review and analyzes the results of about 2800 research papers on refactoring covering the last two decades to offer the most scalable and comprehensive literature review of existing refactoring research studies. Based on this survey, we created a taxonomy to classify the existing research, identified research trends and highlighted gaps in the literature and avenues for further research.

Several studies [83, 84] show that programmers are postponing software maintenance activities that improve software quality, even while seeking high-quality source code for themselves. In fact, the time and monetary pressures force programmers to neglect improving the quality of their source code [7]. Due to the growing complexity of software systems, the last ten years have seen a dramatic increase and industry demand for tools and techniques on software refactoring. To get a deep understanding of the current state of the field and existing research results, we first conducted a systematic literature review (SLR) and analyzed over 2800 research papers on refactoring, spanning

the last two decades. This SLR offers the most scalable and comprehensive literature review of refactoring research to date. Based on our SLR, we created a taxonomy to classify the existing research, identified research trends, and highlighted gaps in the literature and avenues for further research. Refactoring is among the fastest growing software engineering research areas, if not the fastest. Figure 2.1 shows the dramatic growth of the refactoring fi during the last decade. During just the last three years (2014-2016), over 850 papers were published in the field with an average of 270 papers each year. Over 4990 authors from all over the world contributed to the field of software refactoring. We highlight the most active authors in Figure 2.2, based on both number of publications and citations in the area. As seen in Figure 2.3, most of the active refactoring researchers are located in the US, thus motivating the proposed infrastructure in US.

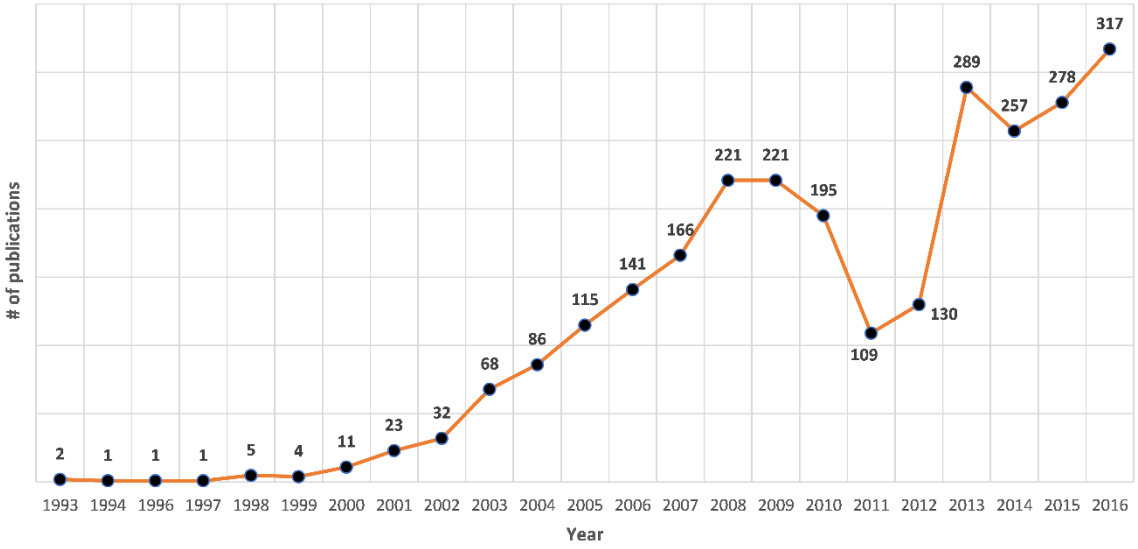

Figure 2.1: Number of refactoring publications over the last two decades.

Figures 2.4 highlight that refactoring research has expanded significantly since its inception in the early 90s. Refactoring now expands beyond code-restructuring and targets different artefacts (architecture, model, requirements, etc.) [28, 9, 10, 11, 12, 13, 14, 15, 16, 85, 18, 86, 87], is pervasive in many domains beyond the object-oriented paradigm (cloud computing, mobile, web, etc.)[88, 89, 90, 91, 68,92, 93, 94, 95, 96], is widely adopted in industrial settings

[69,71], and the objectives expand beyond improving design into other nonfunctional requirements (e.g., improve performance, security, etc.) [87, 40, 97, 98, 99, 100, 85]. The focus of the refactoring community nowadays goes beyond code transformation to include, but not limited to, scheduling the opportune time to carry refactoring [101, 102, 32, 103], recommending specific refactoring activities [87, 54, 40, 32, 104, 105, 106, 107, 108, 65, 109], inferring refactoring from the code [12, 110], and testing the correctness of applied refactoring [111, 106, 103]. Therefore, the refactoring research efforts are fragmented over several research communities, various domains, and diff t objectives, motivating the need for a shared infrastructure to promote reuse and collaboration.

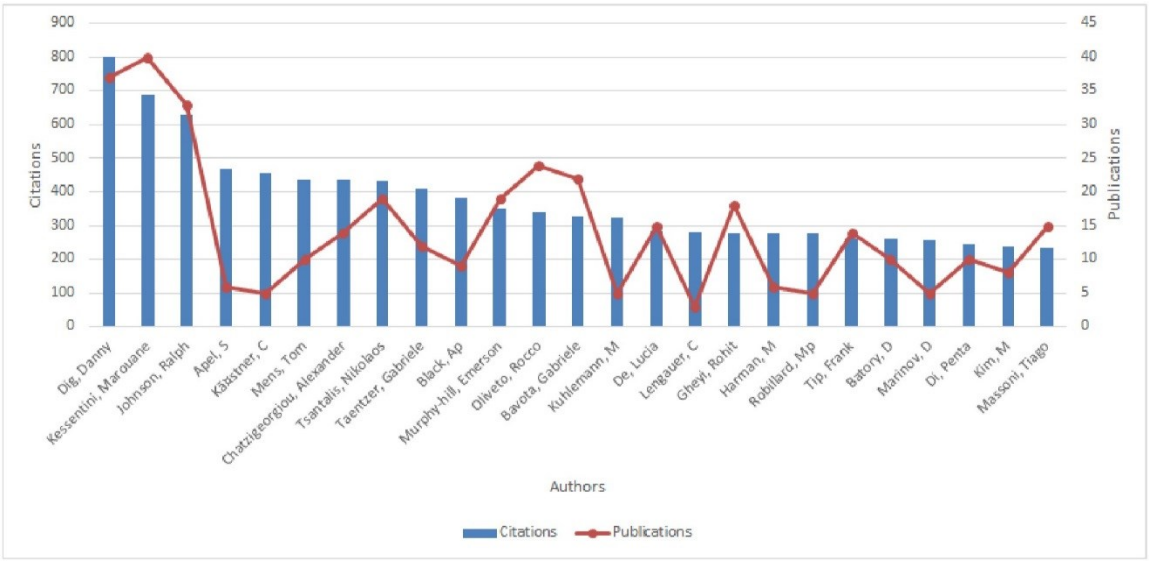

Figure 2.2: Leading refactoring researchers over the last decade based on both publications and citations.

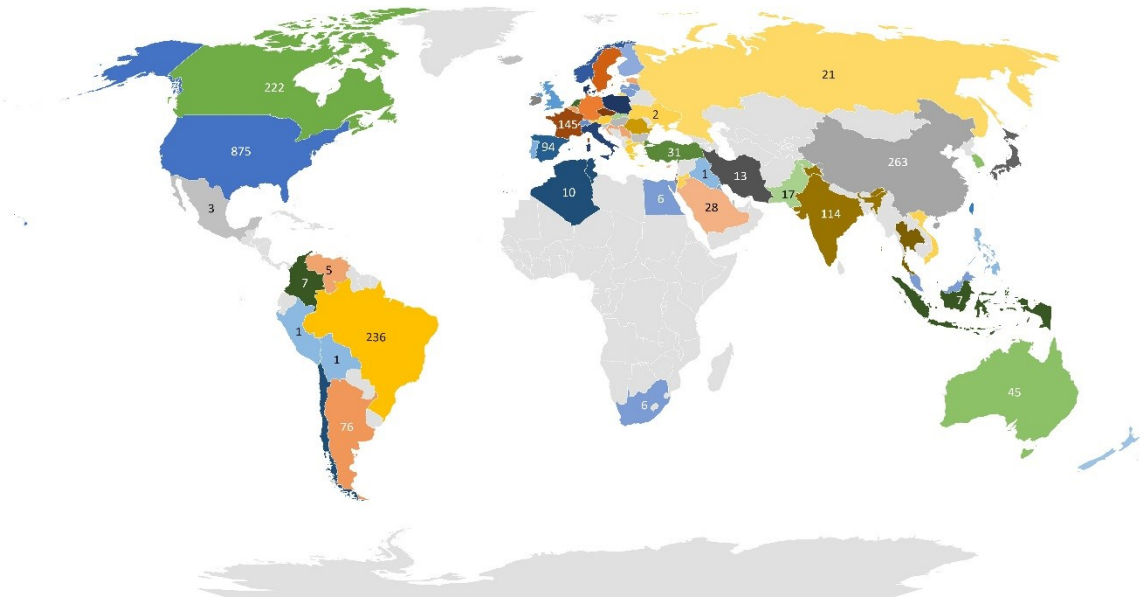

Figure 2.3: Distribution of refactoring researchers around the world.

Manual refactoring can be challenging and error prone. Many Integrated development environment (IDE) and software programming tools have implemented refactoring techniques in their products as a recommendation/guideline or partially/fully automated. Based on a survey [112], 38% of developers answered that the refactoring engine of an IDE was used and 7% of them stated that refactoring was done partially automated. The main reasons for developers to do refactoring manually is that they do not trust automated process for complex refactoring techniques, or the necessary modification is not supported in their choice of IDE. In another study [113], authors pointed out three factors: awareness, trust, and opportunity, and issues with tool work-flow as the limitations affecting usage of tools for refactoring. Therefore, this study can be useful for people from industry and market to be updated from the latest advancements in refactoring.

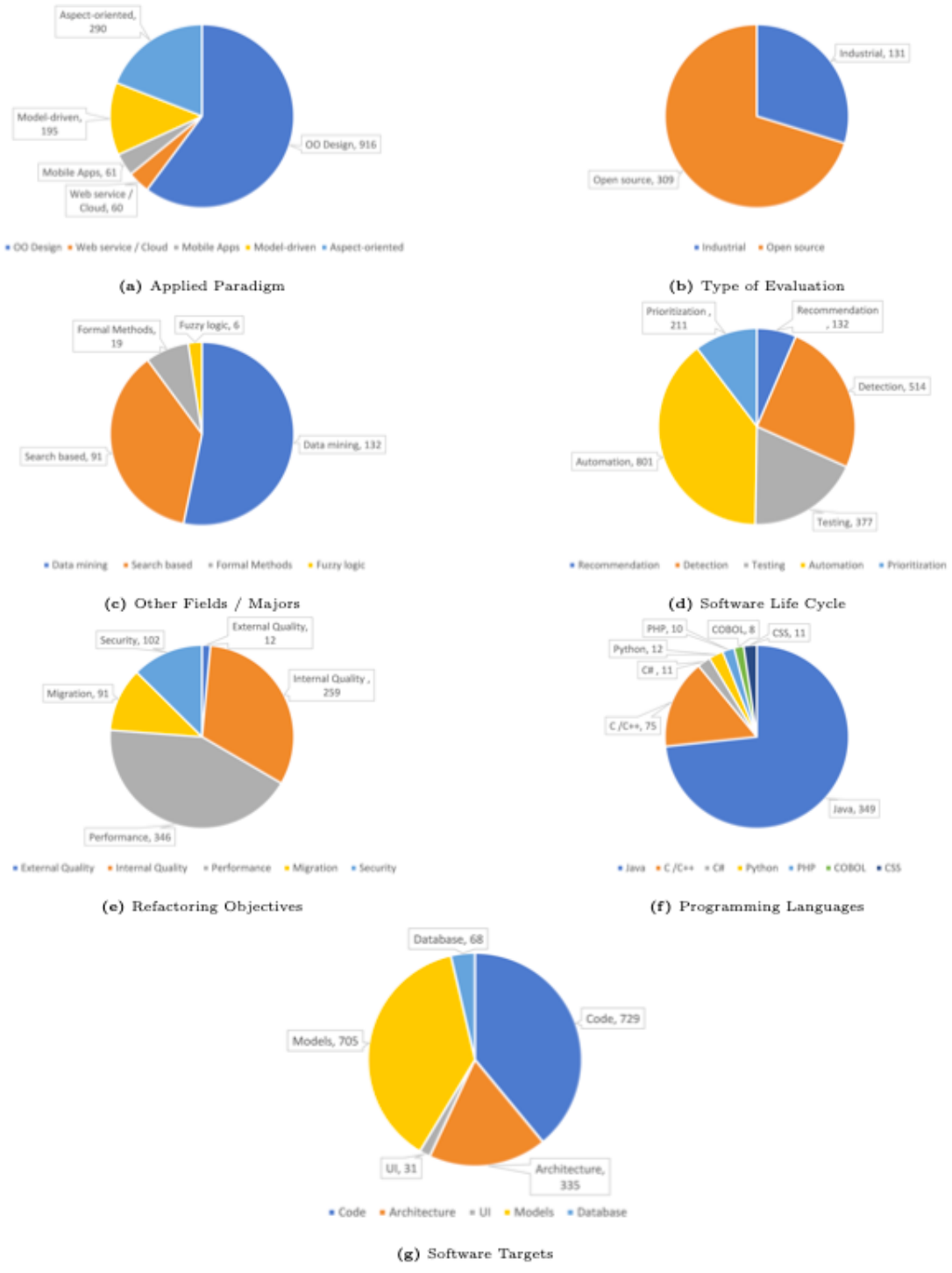

Figure 2.4: Taxonomy of refactoring researches and the number of publications during the past two decade.

Chapter 3: Conclusion

Refactoring is nowadays widely adopted in the industry because bad design decisions can be very costly and extremely risky. On the one hand, automated refactoring does not always lead to the desired design. On the other hand, manual refactoring is error-prone, time-consuming and not practical for radical changes. Thus, recent research trends in the field focused on integrating developers' feedback into automated refactoring recommendations because developers understand the problem domain intuitively and may have a clear target design in mind. However, this interactive process can be repetitive, expensive, and tedious since developers must evaluate recommended refactoring, and adapt them to the targeted design especially in large systems where the number of possible strategies can grow exponentially.

In Chapter I and Chapter II, we defined the problem and the challenges of code refactoring, the contributions of this dissertation, required background (including software refactoring, code quality, *etc*.), and state-of-the-art and related works to this field of refactoring. While code-level refactoring has been widely studied and is well supported by tools, understanding refactoring rationale, or why developers should apply recommended refactoring, is less well understood. Without a rigorous understanding of the rationale for refactoring, existing refactoring recommendation tools will continue to suffer from a high false-positive rate and limited relevance for developers. If, however, refactoring rationale can be identified automatically, this can be used to guide refactoring recommendations to be more purposeful and less ad hoc.

Moreover, once these refactoring have been applied, it is time-consuming for developers to manually document them. However, most existing approaches to automatic generation of

documentation focus on functional changes, which are easier to generate from code changes.

References

[1] “A non-compatible engine control software used at multiple airbus factories.” Businessweek, Retrieved Sept. 2011.

[2] “General motors recalling nearly 4.3 million vehicles on airbag concerns related to software.” Reuters, Sept. 2016.

[3] M. Fowler, M., et al. "Refactoring: improving the design of existing code, Add." Wesley Prof ,1999.

[4] W. F. Opdyke, Refactoring object-oriented frameworks. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[5] Griswold, William G. "Program restructuring as an aid to software maintenance.",1992.

[6] Xiao, Lu, et al. "Identifying and quantifying architectural debt." 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE). IEEE, 2016.

[7] O. Bjuhr, K. Segeljakt, M. Addibpour, F. Heiser, and R. Lagerstro¨m, “Soft- ware architecture decoupling at ericsson,” in Software Architecture Workshops (ICSAW), 2017 IEEE International Conference on, pp. 259–262, IEEE, 2017.

[8] Fontana, Francesca Arcelli, et al. "An experience report on detecting and repairing software architecture erosion." 2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA). IEEE, 2016.

[9] Baxter, Ira D., et al. "Clone detection using abstract syntax trees." Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272). IEEE, 1998.

[10] H. Gall, K. Hajek, and M. Jazayeri, “Detection of logical coupling based on product release history,” in Proc. 14th, pp. 190–197, Nov. 1998.

[11] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy, “An empirical study of code clone genealogies,” in Proc. Joint 10th and 13th, pp. 187–196, Sept. 2005.

[12] S. Kim and M. D. Ernst, “Prioritizing warning categories by analyzing software history,” in Proc. 4th, May 2007.

[13] Kamiya, Toshihiro, Shinji Kusumoto, and Katsuro Inoue. "CCFinder: a multilinguistic token-based code clone detection system for large scale source code." IEEE Transactions on Software Engineering 28.7, 2002.

[14] N. Moha, Y.-G. Gu´eh´eneuc, L. Duchien, and A.-F. Le Meur, "DECOR: A method for the specification and detection of code and design smells," vol. 36, pp. 20–36, Jan. 2010.

[15] M. Kessentini, S. Vaucher, and H. Sahraoui, "Deviance from perfection is a better criterion than closeness to evil when identifying risky code," in 25th International Conference on Automated Software engineering (ASE), pp. 113– 122, 2010.

[16] R. L. Bocchino Jr, V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian, "A type and effect system for deterministic parallel java," in ACM Sigplan Notices, vol. 44, pp. 97–116, ACM, 2009.

[17] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, "Automated detection of refactorings in evolving components," in European Conference on Object- Oriented Programming, pp. 404–428, Springer, 2006.

[18] A. Ouni, M. Kessentini, and H. Sahraoui, "Search-based refactoring using recorded code changes," in 17th European Conference on Software Maintenance and Reengineering (CSMR), pp. 221–230, 2013.

[19] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," IEEE Transaction on Software Engineering, vol. 35, no. 3, pp. 347–367, 2009.

[20] K. Stroggylos and D. Spinellis, "Refactoring–Does It Improve Software Quality?," Fifth International Workshop on Software Quality (WoSQ'07: ICSE Workshops 2007), pp. 3–8, 2007.

[21] A. Kaur and M. Kaur, "Analysis of Code Refactoring Impact on Software Qual- ity," MATEC Web of Conferences, vol. 57, p. 02012, May 2016.

[22] W. Ma, L. Chen, Y. Zhou, and B. Xu, "Do We Have a Chance to Fix Bugs When Refactoring Code Smells?," 2016 International Conference on Software Analysis, Testing and Evolution (SATE), pp. 24–29, 2016.

[23] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo, "When does a refactoring induce bugs? An empirical study," Pro- ceedings - 2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation, SCAM 2012, pp. 104–113, 2012.

[24] G. Suryanarayana, G. Samarthyam, and T. Sharma, Refactoring for software design smells: managing technical debt. Morgan Kaufmann, 2014.

[25] B. Alshammari, C. Fidge, and D. Corney, “Assessing the Impact of Refactoring on Security-Critical Object-Oriented Designs,” in 2010 Asia Pacific Software Engineering Conference, pp. 186–195, IEEE, Nov 2010.

[26] M. Drozdz, D. G. Kourie, B. W. Watson, and A. Boake, “Refactoring tools and complementary techniques.,” AICCSA, vol. 6, pp. 685–688, 2006.

[27] Batory, Don, Jacob Neal Sarvela, and Axel Rauschmayer. "Scaling step-wise refinement." IEEE Transactions on Software Engineering 30.6 (2004): 355-371.

[28] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, Refactoring: Im- proving the Design of Existing Code. Addison-Wesley Professional, 1999.

[29] J. Bansiya and C. G. Davis, “A hierarchical model for object-oriented design quality assessment,” IEEE Transactions on software engineering, vol. 28, no. 1, pp. 4–17, 2002.

[30] M. O’Keeffe and M. O. Cinn´eide, “Search-based refactoring for software main- tenance,” Journal of Systems and Software, vol. 81, no. 4, pp. 502–516, 2008.

[31] A. C. Jensen and B. H. Cheng, “On the use of genetic programming for au- tomated refactoring and the introduction of design patterns,” in 12th Annual Conference on Genetic and Evolutionary Computation (GECCO), pp. 1341– 1348, 2010.

[32] B. Du Bois, S. Demeyer, and J. Verelst, “Refactoring-improving coupling and cohesion of existing code,” in 11th Working Conference on Reverse Engineering (WCRE), pp. 144–151, 2004.

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

[34] E. Murphy-Hill and A. P. Black, “Programmer-friendly refactoring errors,” IEEE Transactions on Software Engineering, vol. 38, no. 6, pp. 1417–1431, 2012.

[35] E. Murphy-Hill and A. P. Black, “Breaking the barriers to successful refactoring: observations and tools for extract method,” in Proceedings of the 30th international conference on Software engineering, pp. 421–430, ACM, 2008.

[36] X. Ge and E. Murphy-Hill, “Manual refactoring changes with automated refactoring validation,” 36th International Conference on Software Engineering (ICSE), vol. 36, pp. 1095–1105, 2014.

[37] X. Ge and E. Murphy-Hill, “Benefactor: a flexible refactoring tool for eclipse,” in Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion, pp. 19–20, ACM, 2011.

[38] S. R. Foster, W. G. Griswold, and S. Lerner, “Witchdoctor: IDE support for real-time auto-completion of refactoring,” in Proceedings of the International Conference on Software Engineering, pp. 222–232, 2012.

[39] L. Tahvildari and K. Kontogiannis, “A metric-based approach to enhance design quality through meta-pattern transformations,” in 7th European Conference on Software Maintenance and Reengineering (CSMR), pp. 183–192, 2003.

[40] D. Dig, “A refactoring approach to parallelism,” IEEE software, vol. 28, no. 1, pp. 17–22, 2011.

[41] Y. Kataoka, D. Notkin, M. D. Ernst, and W. G. Griswold, “Automated sup- port for program refactoring using invariants,” in International Conference on Software Maintenance (ICSM), p. 736, IEEE Computer Society, 2001.

[42] 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, pp. 1037–1039, ACM, 2011.

[43] M. Harman, S. A. Mansouri, and Y. Zhang, “Search-based software engineering: Trends, techniques and applications,” ACM Computing Surveys (CSUR), vol. 45, no. 1, p. 11, 2012.

[44] O. Seng, J. Stammel, and D. Burkhart, “Search-based determination of refactor- ings for improving the class structure of object-oriented systems,” in 8th annual Conference on Genetic and Evolutionary Computation (GECCO), pp. 1909– 1916, ACM, 2006.

[45] M. Kessentini, W. Kessentini, H. Sahraoui, M. Boukadoum, and A. Ouni, “De- sign defects detection and correction by example,” in IEEE International Con- ference on Program Comprehension, (DIRO, Universit´e de Montreal, Canada), pp. 81–90, 2011.

[46] H. Kilic, E. Koc, and I. Cereci, “Search-based parallel refactoring using population-based direct approaches,” in International Symposium on Search Based Software Engineering, pp. 271–272, Springer, 2011.

[47] M. Harman and L. Tratt, “Pareto Optimal Search Based Refactoring at the Design Level,” in Proceddings GECCO 2007, (Department of Computer Science, King’s College London, Strand, London, WC2R 2LS), pp. 1106–1113, 2007.

[48] A. Ouni, M. Kessentini, H. Sahraoui, and M. S. Hamdi, “Search-based refac- toring: Towards semantics preservation,” in IEEE International Conference on Software Maintenance, ICSM, (DIRO, Universit´e de Montreal, Canada), pp. 347–356, 2012.

[49] M. O Cinneide, L. Tratt, M. Harman, S. Counsell, and I. Hemati, Moghadam, “Experimental assessment of software metrics using automated refactoring,” in Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement - ESEM ’12, (School of Computer Science and Informatics, University College Dublin, Ireland), p. 49, 2012.

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

[51] M. Hall, N. Walkinshaw, and P. McMinn, “Supervised software modularisation,” in Software Maintenance (ICSM), 2012 28th IEEE International Con-ference on, pp. 472–481, IEEE, 2012.

[52] G. Bavota, F. Carnevale, A. De Lucia, M. Di Penta, and R. Oliveto, “Putting the developer in the-loop: an interactive GA for software re-modularization,” in International Symposium on Search Based Software Engineering, pp. 75–89, Springer, 2012.

[53] Y. Lin, X. Peng, Y. Cai, D. Dig, D. Zheng, and W. Zhao, “Interactive and guided architectural refactoring with search-based recommendation,” in Pro- ceedings of the 2016 24th ACM SIGSOFT International Symposium on Foun- dations of Software Engineering, pp. 535–546, ACM, 2016.

[54] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, and M. Cinneide “Recommendation system for software refactoring using innovization and interactive dynamic optimization,” Proceedings of the 29th ACM/IEEE international conference on Automated software engineering - ASE ’14, pp. 331–336, 2014.

[55] 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 com- putation, pp. 1461–1468, ACM, 2013.

[56] O. Seng, J. Stammel, and D. Burkhart, “Search-based determination of refac- torings for improving the class structure of object-oriented systems,” in Pro- ceedings of the 8th annual conference on Genetic and evolutionary computation, pp. 1909–1916, ACM, 2006.

[57] N. Tsantalis and A. Chatzigeorgiou, “Identification of move method refactoring opportunities,” IEEE Transactions on Software Engineering, vol. 35, no. 3, pp. 347–367, 2009.

[58] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, “Recommending refactoring operations in large software systems,” in Recommendation Systems in Software Engineering (M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann, eds.), pp. 387–419, Springer Berlin Heidelberg, 2014.

[59] M. O’Keeffe and M. O Cinneide, “A stochastic approach to automated design improvement,” in International Conference on Principles and practice of programming in Java, pp. 59–62, Computer Science Press, Inc., 2003.

[60] M. Harman and L. Tratt, “Pareto optimal search-based refactoring at the de- sign level,” in 9th annual conference on Genetic and evolutionary computation (GECCO), pp. 1106–1113, 2007.

[61] M. Kessentini, W. Kessentini, H. Sahraoui, M. Boukadoum, and A. Ouni, “De- sign defects detection and correction by example,” in International Conference on Program Comprehension (ICPC), pp. 81–90, IEEE, 2011.

[62] A. Ouni, M. Kessentini, and H. Sahraoui, “Search-based refactoring using recorded code changes,” in Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR 2013), pp. 221–230.

[63] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, and M. O Cinneide, Recommendation system for software refactoring using innovization and interactive dynamic optimization,” in Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE 2014), pp. 331–336, 2014.

[64] W. Mkaouer, M. Kessentini, A. Shaout, P. Koligheu, S. Bechikh, K. Deb, and A. Ouni, “Many-objective software remodularization using nsga-iii,” ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 24, no. 3, pp. 17:1–17:45, 2015.

[65] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, and K. Deb, “Multi- criteria code refactoring using search-based software engineering: An industrial case study,” ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 25, no. 3, p. 23, 2016.

[66] M. Fleck, J. Troya, M. Kessentini, M. Wimmer, and B. Alkhazi, “Model trans- formation modularization as a many-objective optimization problem,” IEEE Transactions on Software Engineering, 2017.

[67] A. Ouni, R. G. Kula, M. Kessentini, T. Ishio, D. M. German, and K. Inoue, “Search-based software library recommendation using multi-objective optimization,” Information and Software Technology, vol. 83, pp. 55–75, 2017.

[68] A. Ouni, R. Gaikovina Kula, M. Kessentini, and K. Inoue, “Web service an- tipatterns detection using genetic programming,” in Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation, pp. 1351–1358, ACM, 2015.

[69] A. Ouni, M. Kessentini, S. Bechikh, and H. Sahraoui, “Prioritizing code-smells correction tasks using chemical reaction optimization,” Software Quality Journal, vol. 23, no. 2, pp. 323–361, 2015.

[70] M. Kessentini, M. Wimmer, H. Sahraoui, and M. Boukadoum, “Generating transformation rules from examples for behavioral models,” in Proceedings of the Second International Workshop on Behaviour Modelling: Foundation and Applications, p. 2, ACM, 2010.

[71] A. ben Fadhel, M. Kessentini, P. Langer, and M. Wimmer, “Search-based de- tection of high-level model changes,” in Software Maintenance (ICSM), 2012 28th IEEE International Conference on, pp. 212–221, IEEE, 2012.

[72] M. Kessentini, H. Sahraoui, M. Boukadoum, and M. Wimmer, “Search-based design defects detection by example,” in International Conference on Funda- mental Approaches to Software Engineering, pp. 401–415, Springer, Berlin, Heidelberg, 2011.

[73] M. Kessentini, A. Bouchoucha, H. Sahraoui, and M. Boukadoum, “Example- based sequence diagrams to colored petri nets transformation using heuristic search,” in European Conference on Modelling Foundations and Applications, pp. 156–172, Springer, Berlin, Heidelberg, 2010.

[74] V. Alizadeh, M. Kessentini, W. Mkaouer, M. Ocinneide, A. Ouni, and Y. Cai, “An interactive and dynamic search-based approach to software refactoring rec- ommendations,” IEEE Transactions on Software Engineering, 2018.

[75] E. Murphy-Hill, C. Parnin, and A. P. Black, “How we refactor, and how we know it,” IEEE Transactions on Software Engineering (TSE), vol. 38, no. 1, pp. 5–18, 2011.

[76] M. Kim, T. Zimmermann, and N. Nagappan, “An empirical study of refactor- ingchallenges and benefits at microsoft,” Software Engineering, IEEE Transactions on, vol. 40, pp. 63 July 2014.

[77] C. Lebeuf, M.-A. Storey, and A. Zagalsky, “Software bots,” IEEE Software, vol. 35, no. 1, pp.18–23, 2018.

[78] C. Lebeuf, M.-A. Storey, and A. Zagalsky, “How software developers mitigate collaboration friction with chatbots,” arXiv preprint arXiv:1702.07011, 2017.

[79] M. WESSEL, B. M. DE SOUZA, I. STEINMACHER, I. S. WIESE, I. POLATO, A. P. CHAVES, and M. A. GEROSA, “The power of bots: Understanding bots in oss projects,” Proceedings of the ACM on Human-Computer Interaction, vol. 2, pp. 1–19, 2018.

[80] S. Urli, Z. Yu, L. Seinturier, and M. Monperrus, “How to design a program repair bot?: insights from the repairnator project,” in Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, pp. 95–104, ACM, 2018.

[81] V. Balachandran, “Fix-it: An extensible code auto-fix component in review bot,” in 2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 167–172, IEEE, 2013.

[82] M. Wyrich and J. Bogner, “Towards an autonomous bot for automatic source code refactoring,”

[83] M. Feathers, Working Effectively with Legacy Code. Prentice Hall PTR, 2004. [84] J. Kerievsky, Refactoring to Patterns. 2004.

[85] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, “Automated detection of refactorings in evolving components,” in ECOOP, vol. 4067, pp. 404–428, 2006.

[86] N. Tsantalis and A. Chatzigeorgiou, “Identification of move method refactoring opportunities,” IEEE Transactions on Software Engineering, vol. 35, no. 3, pp. 347–367, 2009.

[87] W. Kessentini, M. Kessentini, H. Sahraoui, S. Bechikh, and A. Ouni, "A cooper- ative parallel search-based software engineering approach for code-smells detection," IEEE Transactions on Software Engineering, vol. 40, no. 9, pp. 841–861, 2014.

[88] A. Ouni, M. Kessentini, K. Inoue, and M. O. Cinn´eide, "Search-based web ser- vice antipatterns detection," IEEE Transactions on Services Computing, 2015.

[89] H. Wang, M. Kessentini, and A. Ouni, "Bi-level identification of web service defects," in International Conference on Service-Oriented Computing, pp. 352– 368, Springer, 2016.

[90] H. Wang, A. Ouni, M. Kessentini, B. Maxim, and W. I. Grosky, "Identification of web service refactoring opportunities as a multi-objective problem," in Web Services (ICWS), 2016 IEEE International Conference on, pp. 586–593, IEEE, 2016.

[91] H. Wang, M. Kessentini, and A. Ouni, "Prediction of web services evolution," in International Conference on Service-Oriented Computing, pp. 282–297, Springer, 2016.

[92] M. K. M. M. G. Marwa Daagi, Ali Ouni and S. Bouktif, "Web service interface decomposition using formal concept analysis," in International Conference on Web Services ICWS2017, pp. 171–180, IEEE, 2017.

[93] H. Wang, M. Kessentini, T. Hassouna, and A. Ouni, "On the value of quality of service attributes for detecting bad design practices," in Web Services (ICWS), 2017 IEEE International Conference on, pp. 341–348, IEEE, 2017.

[94] M. Kessentini and A. Ouni, "Detecting android smells using multi-objective genetic programming," in Proceedings of the 4th International Conference on Mobile Software Engineering and Systems, pp. 122–132, IEEE Press, 2017.

[95] A. Ouni, M. Daagi, M. Kessentini, S. Bouktif, and M. M. Gammoudi, "A machine learning-based approach to detect web service design defects," in Web Services (ICWS), 2017 IEEE International Conference on, pp. 532–539, IEEE, 2017.

[96] M. Kessentini, H. Wang, J. T. Dea, and A. Ouni, "Improving web services design quality using heuristic search and machine learning," in Web Services (ICWS), 2017 IEEE International Conference on, pp. 540–547, IEEE, 2017.

[97] M. Kim and D. Notkin, "Discovering and representing systematic code changes," in Proceedings of the 31st International Conference on Software Engineering, pp. 309–319, IEEE Computer Society, 2009.

[98] Y. Cai, R. Kazman, C. Jaspan, and J. Aldrich, "Introducing tool-supported architecture review into software design education," in 2013 26th International Conference on Software Engineering Education and Training (CSEE&T), pp. 70–79, IEEE, 2013.

[99] A. Yamashita and L. Moonen, “Do developers care about code smells? an exploratory survey,” in 20th Working Conference on Reverse Engineering (WCRE), pp. 242–251, IEEE, 2013.

[100] A. Telea and L. Voinea, “Visual software analytics for the build optimization of large-scale software systems,” Computational Statistics, vol. 26, no. 4, p. 635, 2011.

[101] D. Sahin, M. Kessentini, S. Bechikh, and K. Deb, “Code-smell detection as a bilevel problem,” ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 24, no. 1, p. 6, 2014.

[102] L. Xiao, Y. Cai, and R. Kazman, “Titan: A toolset that connects software architecture with quality analysis,” in 22nd, 2014.

[103] Y. Lin and D. Dig, “A study and toolkit of check-then-act idioms of java concur- rent collections,” Software Testing, Verification and Reliability, vol. 25, no. 4, pp. 397–425, 2015.

[104] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit, “Ref-finder: a refactoring re- construction tool based on logic query templates,” in Proceedings of the International Symposium on Foundations of Software Engineering, FSE, pp. 371–372, 2009.

[105] R. Marinescu, “Detection strategies: metrics-based rules for detecting design fl ws,” in 20th International Conference on Software Maintenance (ICSM), pp. 350–359, Sept 2004.

[106] E. Murphy-Hill, C. Parnin, and A. P. Black, “How we refactor, and how we know it,” in Proceedings of the International Conference on Software Engineering, pp. 287–297, 2009.

[107] J. Kim, D. Batory, D. Dig, and M. Azanza, “Improving refactoring speed by 10x,” in Proceedings of the 38th International Conference on Software Engineering, pp. 1145–1156, ACM, 2016.

[108] A. Ouni, M. Kessentini, H. Sahraoui, and M. Boukadoum, “Maintainability defects detection and correction: a multi-objective approach,” Automated Soft- ware Engineering, vol. 20, no. 1, pp. 47–79, 2012.

[109] M. D’Ambros, A. Bacchelli, and M. Lanza, “On the impact of design flows on software defects,” in Quality Software (QSIC), 2010 10th International Conference on, pp. 23–31, IEEE, 2010.

[110] M. Kim, T. Zimmermann, and N. Nagappan, “An empirical study of refacoring challenges and benefits at microsoft,” IEEE Transactions on Software Engineering, vol. 40, no. 7, pp. 633–649, 2014.

[111] M. O. Cinneide, D. Boyle, and I. H. Moghadam “Automated refactoring for testability,” in Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on, pp. 437–443, IEEE, 2011.

[112] D. Silva, N. Tsantalis, and M. T. Valente, “Why we refactor? confessions of GitHub contributors,” in Proceedings of the 2016 24th ACM SIGSOFT Inter- national Symposium on Foundations of Software Engineering - FSE 2016, (New York, New York, USA), pp. 858–870, ACM Press, 2016.

[113] E. Murphy-Hill, C. Parnin, and A. P. Black, “How We Refactor, and How We Know It,” IEEE Transactions on Software Engineering, vol. 38, pp. 5–18, Jan 2012.