

Expanding Search-Based Software Modularization to Enterprise-Level Projects: A Case Study at Adyen

Master's Thesis

Casper Schröder

Expanding Search-Based Software Modularization to Enterprise-Level Projects: A Case Study at Adyen

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Casper Schröder
born in 's-Hertogenbosch, the Netherlands



Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Tech-
nology
Delft, the Netherlands
www.ewi.tudelft.nl



Adyen
Simon Carmiggelstraat 2-50
Amsterdam, the Netherlands
www.adyen.com

Expanding Search-Based Software Modularization to Enterprise-Level Projects: A Case Study at Adyen

Author: Casper Schröder
Student id: 4485629
Email: c.j.schroder@student.tudelft.nl

Abstract

Code quality of software products often degrades while they grow. Counteracting the degradation of code quality or improving it requires immense effort. Tools that reduce this effort are a hot topic in software engineering research. Software Modularization in particular aims to aid in the process of improving the quality of code structure, by finding flaws in code structure and suggesting improvements. Much research has been done in this field, however, most of it is applied on small to medium scale codebases. In addition, the quality of solutions implied by this research is often not properly validated. This thesis aims to apply an existing approach to an enterprise-level codebase, namely that of Adyen, and validating the results with developers experienced with the code. We achieve this by taking a graph-based approach, applying the NSGA-II algorithm, and introducing a new metric called the Estimated Build Cost of module Cache Breaks. We evaluate the approach in two ways. First, we performed a controlled experiment exploring the feasibility of the approach on larger scale codebases. For this experiment, we apply the approach to the Adyen codebase. The results show that the approach is scalable and shows a significant improvement of code quality in terms of the metrics used. We then performed a user study where we explore the feasibility of these results in practice. For this study, solutions generated during the experiment are split up and filtered to form groups with less than 10 changes, which are reviewed by developers that have changed that code recently or are a senior with experience in that specific area of the code. The results show that the algorithm is successful in identifying flaws in the codebase. However, the improvements it suggests are less precise and require future work.

Thesis Committee:

Chair: Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor: Dr. M. Aniche, Faculty EEMCS, TU Delft
Company supervisor: MSc. A. van der Feltz, Adyen
Committee Member: Dr. N. Yorke-Smith , Faculty EEMCS, TU Delft

Preface

I would like to start off this thesis by thanking the people that made my study and thesis possible. I would like to thank the Delft University of Technology and all my teachers for enabling me to learn so much in the past five years. I would like to thank the other Computer Science students, some of whom became good friends, for the team spirit we had and how we supported each other. I would like to give special thanks to my professor and supervisor Maurício, for the great guidance and feedback during the thesis. I would also like to thank Annibale Panichella, for sharing ideas and giving feedback at several moments during the thesis. I would also like to thank Arie van Deursen and Neil Yorke-Smith for being members of my thesis committee and their time.

I am honored by the opportunity Adyen gave me to do this research with them, and for supporting it to continue by enabling working from home during the Covid-19 lockdown. I am grateful to my company supervisor Adriaan and the rest of the DATT team at Adyen for helping me apply this research on the Adyen codebase, sharing ideas, and most of all having fun while doing it.

Last but not least I would like to thank my friends, family, and girlfriend for all the support over the years.

Casper Schröder
Delft, the Netherlands
September 2, 2020

Contents

Preface	iii
Contents	v
List of Figures	vii
1 Introduction	1
1.1 Contributions	3
2 Code Quality & Modularization	5
2.1 Code Quality	5
2.2 Modularization	7
3 Related Work	11
3.1 Code Quality Metrics	11
3.2 Algorithms/Approaches	14
3.3 Comparison of Existing Modularization Approaches	22
4 Adapting to Scale	25
4.1 Adyen	25
4.2 Challenges of the Scale	25
4.3 Algorithm	26
4.4 Modeling & Representation	27
4.5 Estimated Build Cost of module Cache Breaks	28
5 Constructing & Fine-Tuning the Algorithm	31
5.1 Optimization Variables	31
5.2 Selection	33
5.3 Crossover	33
5.4 Mutation	36

5.5	Constraints	37
5.6	Population Initialization	37
5.7	Optimization Variable & Parameter Tuning	38
6	Empirical Study: Approach in Theory	41
6.1	Research Questions	41
6.2	Methodology	41
6.3	Results	42
7	Case Study: Approach in Practice	51
7.1	Research questions	51
7.2	Methodology	51
7.3	Results	55
8	Discussion	57
8.1	Implications	57
8.2	Practical Use	58
8.3	Threats to Validity	58
9	Conclusions and Future Work	61
	Bibliography	65
A	Glossary	75
A.1	Terminology	75
B	Interview	77
B.1	Interview Questions	77
B.2	Interview answers	78
C	Figures & Tables	85
C.1	LOC correlation with build costs	85
C.2	Optimization variable tweaking results	87
C.3	Parameter tuning results	91
D	Pseudocode Estimated Build Cost of module Cache Breaks Difference	99

List of Figures

2.1	Module dependencies inferred from class dependencies	8
5.1	Crossover operator: effects of trying to preserve building blocks and prevent duplicates. Two modules (M1 & M2) are copied to child solutions in two steps respectively. Classes copied from parent 1 are colored blue and ones copied from parent 2 are colored green. The classes that are copied from parent 1 due to preventing duplicates are colored red.	35
6.1	Pareto front of the first run ¹	42
6.2	Solutions that improve on all metrics from the pareto front of the first run ²	43
6.3	Pareto front of the second run ³	43
6.4	Solutions that improve on all metrics from the pareto front of the second run ⁴	44
6.5	IntraMD convergence of the 72 hour-long run, best and average values of the pareto front	47
6.6	InterMD convergence of the 72 hour-long run, best and average values of the pareto front	48
6.7	EBCCB convergence of the 72 hour-long run, best and average values of the pareto front	49
C.1	The runtime of a compile Java task relative to the lines of code in the module	86
C.2	The runtime of a jar task relative to the lines of code in the module	87

Chapter 1

Introduction

Over the past decades, software has become more intertwined with our daily lives. With the ever-growing demand for functionality provided software, and tech companies wanting to out-compete each other, software services need more and more features and are ever-growing. With the value of the software being defined by its functionality and not its code quality, the latter often degrades when the former is expanded. Especially in enterprise-level software, the emphasis on the delivery speed of features is high. Often the amount of developers working on the code is also relatively high, of which most do not fully understand the entire codebase. The missing understanding of the codebase combined with the number of changes being done and the time pressure for new functionality results in bad code quality if not prevented by putting in lots of effort. The effort cost is not attractive from a business perspective, but the effects of ignoring code quality can be worse.

Bad code quality negatively impacts the software development process in many ways [4] and can also impact the quality of the product [62]. One of the main aspects that is negatively impacted is the understandability of the code. In other words, how easy developers can work with the code. Another is the costs associated with building, running, and testing the code. Different parts of the codebase can get more coupled, duplicate code can arise and code can get less efficient. While these effects might not be apparent in the short term, the code quality keeps degrading over time, slowly increasing the negative effects and decreasing the understandability [66]. This concept is known as technical debt [20].

As understandability and usability are abstract and subjective concepts, they cannot be easily defined or measured. Instead, researchers often measure code quality by coupling and cohesion [54]. Coupling represents how intertwined the code is, while cohesion represents how well the code is organized so that it belongs together. These are still abstract concepts but measurable metrics that represent them are easier to make. These metrics can be used to create tools

that can help improve code quality.

These tools that assist in this process are very useful, as improving code quality can be very costly and time-consuming. In this thesis, we propose and explore a modularization recommendation approach, based on a multi-objective evolutionary algorithm, that is adjusted for enterprise-level software. The approach is applied to an enterprise-level codebase which is object-oriented. The goal of this approach is to reduce the effort needed to modularize a codebase by removing the need to manually search through dependencies. This thesis contributes to the knowledge in the field of search algorithms for modularization by applying an existing approach on larger scale software than done before and adjusting it as needed, adding a new metric as an optimization variable, and investigating the quality of the suggestions in practice with the help of developers working on the enterprise software it is applied on.

There exist a significant body of related work in modularization and code refactor suggestions. Some use Machine Learning [60, 70], some Cutting or Clustering [33, 37, 43, 17], and most use Evolutionary Algorithm approaches [36, 35, 46, 56, 50, 1, 53, 30, 38, 27, 6, 2, 31, 61]. The most used algorithm is the NSGA-II algorithm [22], which we also use in this thesis. Many different code quality measures are used, yet close to all works use some measurements representing coupling and cohesion.

In order to apply the approach to enterprise-level software, we make decisions based on existing work. These decisions try to find the most efficient way to tackle the problem. We test several optimization variable configurations consisting of metrics used in existing work. We also add a new metric to represent a coupling issue that is more prevalent in bigger codebases, namely transitive coupling. The metric is called the Estimated Build Cost of module Cache Breaks (EBCCB), and it represents weighted transitive coupling. The configuration we decide to use contains the following metrics: Intra-Module Dependencies, Inter-Module Dependencies, # Class Move Refactorings, and EBCCB. The results show that the approach was successful, and the added metric significantly reduces weighted transitive coupling.

To validate the results work in a real scenario, parts of the generated solutions were reviewed by developers that are experienced in that part of the codebase. They were interviewed to understand the quality of the suggestions. From these interviews, we found that less than half of the inspected code changes, which were all good according to the metrics, were good in reality, which questions the applicability of this type of approach in reality. However, all suggested changes led a developer to find a flaw in the code and an idea on how to fix it.

These results show that the approach and likely similar ones can be applied to enterprise-level codebases and achieve a theoretical significant improvement in terms of the metrics used. It is also shown that the metric values do not necessarily translate to practical quality, which implies that modularization research

results should be validated. This validation is often overlooked [52].

The thesis starts with explaining the relation between code quality and modularization and stating the modularization problem. After this, it goes into related work. The chapter after describes the challenges the bigger scale codebase introduces, and which decisions were made to tackle these challenges. This is followed up by the chapter that describes the configuration of the algorithm. The theoretical and practical value of the approach is examined in the empirical study and case study chapters. The empirical study chapter describes how the approach performs from a metric point of view, and the case study chapter describes how the solutions delivered by the approach were reviewed by developers. Finally, the approach is discussed, conclusions are made, and the potential future work is proposed.

1.1 Contributions

This project makes three main contributions.

1. A multi-objective evolutionary optimization approach to the remodularization problem can be applied on large-scale enterprise-level codebases and reach significant improvements from a metric point of view, verified by a controlled experiment showing its effectiveness.
2. A user study on the practical value of solutions that are generated by the approach. This study shows the importance of this type of validation and that more work is required to properly translate the improvement shown by the metrics to a real scenario.
3. A new metric is introduced to measure an aspect of a codebase which becomes more important the bigger the codebase becomes, namely weighted transitive dependencies. The metric is called the Estimated Build Cost of module Cache Breaks.

Chapter 2

Code Quality & Modularization

In this chapter, code quality and the relation it holds with the modularization problem are explained. The chapter starts with going into code quality, specifically structural quality, and the two most used aspects of it in research, coupling, and cohesion. After this, it explains the dependency structure and its implications and uses for modularization. Finally, it states the modularization problem.

2.1 Code Quality

Code quality represents lots of factors. How well the code performs in terms of runtime, how readable it is, how well it is documented, etc. For this thesis, we will look at the quality of the code structure specifically, as it is the goal of modularization to improve this. Code structure quality represents how well classes are put together, how well classes are divided among modules, and how proper these modules are connected among other things. Two aspects that are often used when investigating code structure quality are coupling and cohesion.

2.1.1 Coupling

Bavota et al. state that coupling is a fundamental property of software systems and that coupling measures are often used to determine code structure quality [8]. Coupling is an abstract concept, representing how interconnected the code is. As stated by Stevens, Myers, and Constatine, coupling is the measure of the strength of association between modules [63]. On a class level, this represents how connected to other classes a class is, be it through using other classes or being used by other classes. If a piece of code is using many other pieces of code, or it is used by many other pieces of code, chances are that it's using or being used wrongly.

2. Code Quality & Modularization

Higher coupling is a negative aspect, among other reasons due to the following [8, 11, 13, 15]:

- It makes the code harder to change. The more code is connected, the more code needs to be checked and possibly changed if a part of the code is changed.
- It propagates errors further. If a part of the code is faulty and performs unwanted behavior, returns faulty output, or even crashes, this impacts a bigger part of the codebase.
- It makes the code harder to understand. To fully grasp a part of the code, it and the code it depends on need to be looked at. More coupled code means more code to inspect.

Coupling plays an important role in modularization research, as it is relatively easy to measure and an easy to grasp concept. It has also been shown to have a strong relation with fault detection [13]. Decoupling code is the process of decreasing coupling and as such improving code quality [42]. These improvements in terms of coupling can be achieved by removing code connections in any way.

Coupling has been empirically shown to be perceived by developers similarly as it is measured by some coupling measures. Revelle, Gethers, and Poshyvanyk introduce a set of textual and structural feature coupling metrics and show that the metrics align with how developers perceive coupling [57]. Bavota et al. show that coupling is best measured in structural and semantic measures [8]. In their study, developers' perception aligns most with the semantic coupling measures.

2.1.2 Cohesion

Another code quality concept often used in code quality assessment is cohesion [19, 26, 14]. Cohesion is the abstract idea representing the degree to which elements inside a module belong together [69]. From a class perspective, this means it should fulfill one purpose or hold one responsibility, and every variable and method defined should contribute to that purpose or responsibility. From a module perspective, this means that classes in a module should use each other and/or be used together and have a clear shared purpose.

Higher cohesion is a positive aspect. It makes the code easier to understand and work with. It causes the code to be more robust. If a piece of code is highly cohesive, it is most likely easier to reuse, refactor, or replace. In other words, it has opposite effects of high coupling.

Cohesion plays an important role in modularization research as well, as it is a powerful tool in the design of modular structures [69]. It can show how

to divide the code into modules [42]. Code can be restructured optimizing for many different aspects, however, if the code has low cohesion afterwards, it will be hard to use, understand and maintain, and more error-prone. Cohesion is generally harder to measure than coupling, as its definition is more abstract, making it difficult to represent it in metrics.

Cohesion being harder to grasp becomes clear from a developer perspective in empirical research such as Counsell et al. [19]. They show a difference in judging cohesion between novice and experienced developers. Their results imply that a class is cohesive if it has low coupling, higher numbers of comment lines, and methods that contribute to a common goal of the class. The difficulty of forming proper cohesion metrics is clear from the work of Etzkorn et al. [26]. They show that many cohesion metrics correlate to the developers' perspective, however not to a degree where the metrics capture the majority of the value. In the case of improving code quality through refactoring, developers do not necessarily pursue an optimal balance between cohesion and coupling, as shown by Candela et al. [14]. Additionally, the majority of developers surveyed confirmed that they were guided by other properties than cohesion and coupling.

2.2 Modularization

A type of problem definition that specifically tries to improve on code structure quality is modularization. Modularization is the effort of dividing code into groups in such a way that it results in a good code structure. Often this is done on module level, meaning that it tries to optimize package or class organization over modules. Modularization is key in large codebases and its quality is important, as bad modularization can be worse than no modularization at all [9]. In order to properly define the problem of modularization, we need to explain the concept of a dependency structure.

2.2.1 Dependency Structure

A dependency structure represents the individual pieces of code of a codebase, and the connections between them. In object-oriented programming, classes are the pieces of code and the dependencies are the connections between them. A class depends on another class if the former uses any functionality from the latter directly, by for example using a method from the latter. From now we will refer to this as class dependencies.

These dependencies form a dependency structure, which can be represented by a directed graph where the nodes are classes and the edges are the dependencies from the using classes to the used classes. This graph will be referred to as the class dependency graph. By grouping these classes into modules, the module dependency structure can be inferred. If modules that classes are in

2. Code Quality & Modularization

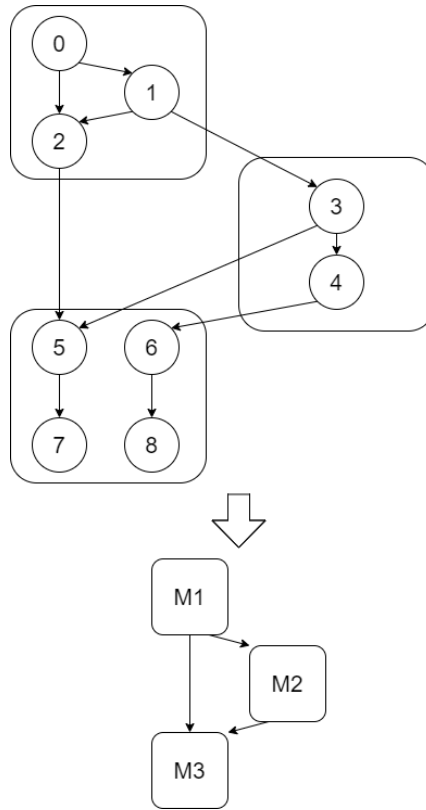


Figure 2.1: Module dependencies inferred from class dependencies

are represented as groups of nodes, the edges that go from group to group, show the dependencies between modules. This can be seen in Figure 2.1. This figure shows classes as nodes, labeled with numbers, and the dependencies as directed edges. The squares represent a possible modularization, which results in the module to module dependency structure below the arrow.

To run a piece of code, it needs to be compiled. Before this can happen, all its dependencies should be compiled first. These two graph representations can be used to determine which modules need to be compiled and built first when compiling and building a module. These graphs can also be used to derive information about the codebase. Specifically, they can be used as the input of metrics that measure code quality, for example, metrics representing coupling and cohesion.

2.2.2 Problem

The problem we are faced with is as follows: *What is the best division of code into modules?* We want to create these modules in such a way that the result is a dependency structure of high quality, and in turn a code structure of high

quality. Managing dependencies to solve this problem is not trivial. One of Martin's most well-known books describes design flaws that all relate to dependency management, and how they should be prevented or fixed through proper development, management, and planning[42]. As many other factors also go into what makes a specific modularization "good", finding the optimal solution is infeasible. Instead, modularization is optimized according to one or multiple metrics, often ones that represent coupling and cohesion and are measured using the dependency structure among others. By creating a modularization that improves on these metrics, we assume that the code structure quality and as such the overall code quality is also improved upon.

Chapter 3

Related Work

Modularization of code is not a new topic in software engineering research. The field has been researched for more than 20 years [60, 33]. The goal of modularization is to improve the codebase it is applied to from a structural point of view. As a "good" codebase is an abstract concept that is hard to quantify, instead metrics are used to measure code quality. The metrics that are used to measure code quality are explored in the first section. To apply these metrics on code projects, various approaches have been implemented over the years using various algorithms, which will be described in the second section of this chapter. Finally, papers that compare approaches are inspected and potential shortcomings of existing research and additions to the field are identified.

3.1 Code Quality Metrics

There are many ways to measure the quality of code. In this section, we describe code quality metrics that have been proposed by related work, structural metrics specifically.

Allen, Khoshgoftaar, and Chen propose coupling and cohesion as quality measures, and that they can be easily measured in a modular system [3]. They measure coupling by the amount of inter-module connections, and cohesion by the intra-module coupling, normalized to between zero and one. In describing these measures, the authors refer to work done by Briand, who has expanded these measures further to also apply on class level. In his paper with Devanbu and Melo, the coupling measure is further expanded to inter-class coupling [10]. This coupling measure is based on how every class in an object-oriented system is connected. It covers more than direct class to class connections. This measurement has been empirically shown to be a reasonable measure of fault-proneness. In another paper by Briand, Daly, and Wüst, many established ways of measuring cohesion between classes are compared [12]. A critique that is applied to most measurements is that they do not have empirical studies behind

3. Related Work

them and the ones that do may not improve the quality of the code in practice when they are used. In a later paper, Briand et al. show that when a proper set of coupling and cohesion measures are used together, accurate models can be built to find fault-prone classes [13].

Chidamber and Kemerer propose a set of six metrics to measure code quality, which has been used in modularization related research [16]. The metrics are the following:

1. Weighted Methods per Class (WMC): the complexity of all methods in a class.
2. Depth of Inheritance Tree (DIT): the maximum length from the class to the root of the inheritance tree.
3. Number Of Children (NOC): the number of children a class has.
4. Coupling Between Object classes (CBO): the count of the number of other classes to which it is coupled.
5. Response For a Class (RFC): the size of the response set of a class. The response set of a class consists of all methods that this class can call.
6. Lack of Cohesion in Methods (LCOM): measured by for every pair of methods in a class, summing the number of class variables that are used by only one of the methods.

Hitz and Montazeri show that the measures proposed by Chidamber and Kemerer were not correctly empirically tested, because the representation condition was ignored [32]. The representation condition states that for every empirical relation, the same relation should hold after applying the measure. They show that this does not hold for the measures.

Another set of metrics called Modularization Quality (MQ) is proposed by Mancoridis et al. [40]. This set consists of intra-connectivity represented by a fraction of the maximum number of intra-dependencies, and inter-connectivity represented by a fraction of the maximum number of inter-dependencies. The resulting modularization is a weighted sum of the two, such that the resulting value is bounded between -1 (no cohesion) and 1 (no coupling).

Martin also proposes a set of metrics in his book [42]. Three metrics that are often used in modularization research are the following. Acyclic-Dependencies Principle (ADP), which states that no cycles are allowed in the package-dependency graph. The Common-Closure Principle, which states that classes in a package should be closed together against the same kinds of changes. A change that affects a package affects all the classes in that package and no other packages. And the Common-Reuse Principle (CRP) states that

classes in a package are reused together. If you reuse one class in a package, you should reuse them all.

Different types of metrics have been proposed and investigated. Meyers and Binkley study sliced base cohesion and coupling metrics [44]. It shows that these metrics quantify the deterioration of a program as it ages. They show that the metrics are not proxies for size-based metrics and that they should have baselines. The cohesion metrics they use are: the number of statements in every slice, the ratio of the smallest slice compared to a module's length, length of slices to length of module, largest slice to the length of a module, amount of statements common to all slices, and the number of slices with no more than n statements in common.

Besides structural cohesion metrics, semantic cohesion metrics are also used. Marcus, Poshyvanyk, and Ferenc have used the conceptual cohesion of classes to predict faults [41]. This cohesion measure is based on textual information in the source code, both in identifiers as in comments. They use Latent Semantic Indexing to get this textual information.

Some studies combine structural and semantic metrics. Bavota et al. measure package cohesion for modularization by using the following metrics [7]. Information-Flow-based Coupling (ICP) and Conceptual Coupling Between Classes (CCBC). ICP is measured by the number of methods called between to classes weighed by the number of parameters shared in those methods. CCBC is measured by code semantics in comments and identifiers.

Because of the different quality types to measure, different types of metrics, and the number of metrics introduced, some proposed models to measure code quality are significantly more complex. One of these examples is the model that Bansiya and Davis propose, which assesses the quality of high-level design attributes [5]. It combines metrics such as encapsulation, modularity, coupling, and cohesion to represent quality attributes such as reusability, flexibility, understandability, etc. They show that models like these can be used to monitor the quality of a software product. Another example of a model with high complexity is the one Sarkar, Rama, and Kak present. They present a new set of metrics to measure software modularization quality [59]. It is a combination of coupling-based structural metrics, size-based metrics, architectural metrics, and metrics based on the similarity of purpose, which amalgamates to twelve metrics in total.

Models to measure coupling and cohesion also differ in complexity, Dhama presents a model to measure coupling and cohesion, where both concepts are divided into several aspects [23]. Cohesion is split into functional, data flow, action-bundling, and logical bundling cohesion. Coupling is split into data and control flow, global, and environmental coupling. This model is not applied in this paper and no conclusions are made.

Acknowledging the importance of practicality for code quality metrics,

Bavota et al. performed an empirical study on how developers perceive software coupling in several open-source systems [8]. They focussed on class coupling specifically, and show that the concept of coupling goes beyond the structural measures which are often used to represent it. Coupling can also be captured by semantic measures, which seem to complement the structural measures.

History of development can be an important factor, as Ying et al. show. They show that source code change recommendations can be formed by using the change history a project [68]. They imply that sets of files that have been frequently changed together will likely need to be changed together in the future. They achieve this through data mining and coupling/cohesion metrics.

Multiple papers have also shown a correlation between code quality metrics and historical development data, using one to determine the other. Romano and Pinzger use code metrics to predict change-prone java interfaces [58]. They use the following metrics: Coupling Between Objects, Lack of Cohesion Of Methods, Weighted Methods per Class, Number of Children, Depth of Inheritance Tree, Response For Classes, and Interface Usage Cohesion. Elish and Al-Khiaty present a suite of metrics for quantifying historical changes to predict change prone classes [25]. This suite consists of 16 metrics meant to translate the change history to future changes.

Not all of these attempts are successful, however. Kabaili, Keller, and Lustman propose to use cohesion as an indicator of changeability [34]. They state that coupling has already been shown to be such an indicator. Two class level cohesion metrics, LCC and LCOM, are used but are shown to not cover cohesion properly and should not be used as changeability indicators.

3.2 Algorithms/Approaches

The modularization problem has been approached using several types of algorithms. This section will describe the algorithms used. A summary of these approaches can be seen in Tables 3.1, 3.2, and 3.3.

Table 3.1: Summary of problems optimized for, algorithms used, and variables optimized of related work (1/3)

Authors (year)	Problem	Approach	Optimized for (+: maximize / -: minimize)
Schwanke, Hanson (1994)	Modularization	Machine Learning (neural network)	More-than-similar judgements (+)
Jermaine (1999)	Modularization	K-cut	Cohesion (intra-edges) (+) Coupling (inter-edges) (-) Cohesion (core functionality similarity) (+)
Kook Lee et al. (2002)	Split components from codebase	Novel clustering algorithm	Coupling (message passing between classes) (-) Coupling (connection in some form) (-)
Harman, Hierons, Proctor (2002)	Modularization	GA	Cohesion (intra-edges) (+) Coupling (inter-edges) (-) # of modules (-)
Zhang, Jacobsen (2003)	Refactor suggestions	Aspect Mining	Discover non-localized units of modularity (type-based, lexical-based patterns) (+)
Mitchell (2003, 2006, 2007)	Modularization	GA, clustering by hill climbing	Cohesion (intra-edges) (+) Coupling (inter-edges) (-)
Kwong et al. (2010)	Modularization	GA	Cohesion (intra-edges) (+) Coupling (inter-edges, class interactions) (-) Functional performance (+)
Kessentini et al. (2010)	Defect detection	GA	# detected defects compared to example defects (+)
Marx, Beck, Diehl (2010)	Split components from codebase	Exhaustive, hill climbing, and min-cut algorithms	Amount of class to class dependencies broken (-)

3. Related Work

Table 3.2: Summary of problems optimized for, algorithms used, and variables optimized of related work (2/3)

Authors (year)	Problem	Approach	Optimized for (+: maximize / -: minimize)
Praditwong, Harman, Yao (2011)	Module clustering	Two archive pareto optimal GA	Cohesion (intra-edges) (+) Coupling (inter-edges) (-) # of clusters (+) MQ (+) # of isolated clusters (-) Difference between min and max # of modules in a cluster (-)
Ouni et al. (2012)	Modularization	NSGA-II	Local semantic similarity (vocabulary based) (+) Program structure quality (shared method calls / field access) (+)
Fadhel et al. (2012)	Model refactoring	GA	Similarity between model and expected model (+)
Bavota et al. (2012)	Remodularization	Interactive NSGA-II	MQ Effort required (MoJoFM) Developer can designate changes to be useless
Mahouachi, Kessentini, Cinneide (2013)	Find refactorings that took place	GA	Similarity between historical model and current model (+)
Ghannem, Boussaidi, Kessentini (2013)	Model refactoring	Interactive IGA	Similarity between model and expected model (+), Designer can rate solutions 1-5
Mkaouer et al. (2013)	Model refactoring	Preference based MOEA	Model quality (+) Rule complexity (-)
Abdeen et al. (2013)	Remodularization	NSGA-II	Cohesion (CCP, CRP) (+) Coupling (ADP) (-) Package size (-) # of module changes (-)
Chong, Lee, Ling (2013)	Cluster software	Dendrogram cutting, Least squares regression	Cohesion (avg. distance between each entity in cluster) (+) Coupling (min. distance between centroids of the sides of the cut) (-)

Table 3.3: Summary of problems optimized for, algorithms used, and variables optimized of related work (3/3)

Authors (year)	Problem	Approach	Optimized for (+: maximize / -: minimize)
Mkaouer et al. (2014, 2016)	Refactor suggestions	NSGA-II	Code smell severity improvement (+) Improvement importance (+) Quality of improvement (+) # classes per package (-) # packages (-)
Mkaouer et al. (2015)	Remodularization	NSGA-III	Package cohesion (intra-edges) (+) Package coupling (inter-edges) (-) Semantic grouping (vocabulary similarity) (+) # code changes needed (-) Consistency with development history (+)
Ghannem et al. (2017)	Refactor suggestions	NSGA-II	Structural similarity compared to refactored models (+) Structural similarity compared to well designed models (-) Structural quality (intra, inter-package dependencies) (+)
Mahouachi (2018)	Remodularization	NSGA-II	Semantic Cohesiveness (vocabulary similarity) (+) Refactor effort (rate of achieved improvement) (+) # code smells (-)
Alizadeh (2018)	Refactor suggestions	Interactive NSGA-II	Semantic coherence (+) Quality Model for Object-Oriented Design (+) # of refactorings needed (-) Feature modularization (+)
Da Silva et al. (2020)	Product Line Architecture	NSGA-II	SPL extensibility (+) Variability (-) Coupling (-) Cohesion (+)

3.2.1 Evolutionary Algorithms

Evolutionary Algorithms (EA) work through simulating some sort of population of solutions. Every solution is measured by some optimization variable, often called fitness, to define how good it is. In addition to these fitness measure-

3. Related Work

ments, EAs consist of an exploration technique and a selection technique, effectively simulating evolution in order to optimize a problem. A very common EA is the Genetic Algorithm (GA), which creates new solutions by combining their aspects, and mutation, in which solutions gain or lose aspects in some way. After new solutions are formed, often either the parent solutions or the worst n solutions in terms of fitness are discarded. This process is repeated until a time limit or generation limit is found, or until the algorithm has converged, meaning it found a local optimum from which it cannot find better optima.

Kwong et al. used a GA to optimize the modularization problem [36]. They optimize for cohesion, which they measure by the number of class interactions within modules, and coupling, which they measure by the number of class interactions between modules. Besides cohesion and coupling, they use another objective, namely maximizing the functional performance of the component-based software system based on a formula representing how well the components fit in their respective modules.

Kessentini et al. use a GA in which the population is modeled as combinations of refactorings to find a solution that optimizes by maximizing the number of detected defects in comparison to expected ones in the base of examples [35].

Some studies also show that GAs outperform hill-climbing algorithms in modularization problems. One example is the works related to the Bunch tool. Mitchell has published multiple works that describe the workings of this tool [45, 46, 47]. The purpose of this tool is to modularize a codebase. Initially, it used clustering algorithms, of which some were hill climbing. Later, genetic algorithms were added to increase the search potential of the tool, as hill climbing had an issue in creating new modules. These algorithms optimize the codebase for coupling and cohesion measures. More specifically, the amount of inter-module dependencies and intra-module dependencies. This tool has been used successfully on real projects. Praditwong, Harman, and Yao propose to use a novel two-archive pareto optimal genetic algorithm to optimize module clustering [56]. They optimize for six different values: Cohesion by summing intra-cluster interactions, Coupling by summing inter-cluster interactions, maximizing the number of clusters, maximizing MQ, minimizing the number of isolated clusters, and minimizing the difference between the maximum and the minimum number of modules in a cluster. They advocate that this approach reaches better results than single-objective approaches and that the richer solution space gives more choices to the software engineer that will use it. They use weighted and unweighted module dependency graphs. The GA outperforms the hill-climbing algorithm significantly on the weighted graph, but the computation cost is approximately double. It is argued that this cost is worth the increase in the quality of the results.

Two algorithms that are often used in modularization problems are the Non-dominated Sort Genetic Algorithm-II (NSGA-II) and NSGA-III. Deb et al. intro-

duced the NSGA-II algorithm [22]. This algorithm is a multi-objective evolutionary algorithm (MOEA) which is based on nondominated sorting. Instead of one solution which is the fittest, this algorithm measures the solutions using multiple optimization variables and results in a pareto front. A solution is part of the pareto front if no other solution is found that is equal or better on all optimization variables and at least better on one. This algorithm was found to perform well, but lose proper solution diversity when faced with many optimization variables. Deb and Jain, in their paper, show that for MOEA approaches with higher amounts of objectives optimized for, the NSGA-III algorithm they propose outperforms NSGA-II [21]. This algorithm uses solution reference points to force solution diversity for all dimensions. Ciro et al. even show that for some problems in large-sized instances, NSGA-III outperforms NSGA-II when optimizing more than 2 variables [18].

Mkaouer et al. propose a way to suggest possible refactorings that can deal with code smells [49]. They focus on adding two aspects to existing research, namely code smell severity and code smell class importance. They also propose using the NSGA-II algorithm to implement this. As a continuation of this work, Mkaouer et al. use a modified NSGA-II to refactor in a way that tries to find the best trade-off between three objectives: quality, severity, and importance of improvements [50].

Abdeen et al. state that before their work, modularization was considered from the ground up. They introduce the remodularization problem, which takes an existing codebase and improves its quality while minimizing changes [1]. They use the NSGA-II algorithm. They optimize for coupling, cohesion, modification degree, and class distribution, which are constructed from CCP, CRP, ADP, package sizes, and the number of moved classes. They allow the users of the algorithm to set certain constraints, namely constraining the refactoring space, guiding the process by marking code that is likely to move and limiting the maximum amount of changes.

Some studies optimize not only on program structure but also on semantics or structural similarity to verifiably good models. Ouni et al. create refactoring suggestions using the NSGA-II algorithm which optimizes program structure and semantics. The semantics are measured by using vocabulary-based similarity, and program structure by shared method calls and shared field access [53]. From the solutions generated, they decide that the best solution is the one that is ideal in terms of Euclidian distance in the pareto front. Ghannem et al. use the NSGA-II to suggest refactorings by maximizing the structural and textual similarity between the model to be refactored and models that have undergone refactorings, and minimizing the structural and textual similarity between the model and well-designed models that do not need refactoring [30]. Mahouachi also uses the NSGA-II algorithm to optimize remodularization for 3 objectives, structural quality, semantic cohesiveness, and minimizing the refactoring effort.

3. Related Work

Structural quality is measured by intra- and inter-package dependencies, semantic cohesiveness by vocabulary similarity, and refactoring effort by the rate of achieved improvement [38]. Mkaouer et al. approach the modularization problem using the NSGA-III algorithm and 7 optimization objectives, including size metrics, intra- and inter-edges representing cohesion and coupling, number of code changes, and similarity to historical changes [51]. They argue that modularization should be optimized for more than structure quality in the form of cohesion and coupling to achieve solutions of higher quality. Fadhel et al. use a GA to create an adjusted high-level code model through refactorings that should be as similar to a given expected model as possible [27]. It optimizes for the similarity between the given and expected model as fitness. Mahouachi, Kessentini, and Cinneide use a genetic algorithm to take a set of refactoring possibilities and several versions of the software to find which refactorings have taken place between those versions [39]. The algorithm is near identical to the one used in the work by Fadhel et al. [27].

Some works have made the algorithms interactive or preference-based, to in a way optimize for a developer perspective as well. Ghannem, Boussaidi, and Kessentini provide an interactive genetic algorithm to perform model refactoring [29]. The GA is also near-identical to that of Fadhel et al. [27], with the interactive part added. The designer can rate changes in 5 ways, reaching from critical to inappropriate. The value of these refactorings is weighted by the designer's feedback. Bavota et al. use an interactive genetic algorithm (based on NSGA-II) to tackle the modularization problem [6]. They evaluate the solutions with MQ as the quality metric and MoJoFM [67] as the effort required. Alizadeh et al. used an interactive version of the NSGA-II algorithm to create software refactoring recommendations [2]. The goal is to improve software quality while minimizing the deviation from the initial design. The algorithm runs for a set amount of generations, optimizing for the number of code smells, semantic coherence, Quality Model for Object-Oriented Design, and the number of refactorings needed. After that, a developer reviews the results, and feedback is fed into the algorithm. Solutions are modeled as a list of field/method level refactorings. 6 optimization variables are used that are constructed from summing and weighing 10 different measures. Mkaouer et al. tackle multiple software modeling challenges with a different algorithm. They use a preference-based multi-objective evolutionary optimization technique [48]. The preferences are modeled by weights assigned by the user and setting a desirability threshold to optimization variables. The algorithm optimizes for model quality and rule complexity.

Some studies emphasize the importance of preserving building blocks while performing crossover. This means that a different crossover operator is used which is made specifically to prevent aspects in a solution that amplify each other's positive effect from being split up. Harman, Hierons, and Proctor pro-

pose an approach to modularization which uses a GA [31]. They use a fitness function that includes 3 metrics: coupling in the form of inter-module coupling, cohesion in the form of intra-module coupling, and granularity in the form of the number of modules. They introduce a new crossover operator, which instead of choosing a single point in the list of changes and forming children with the pieces it divides the modules of the parents uniformly over the children. This should preserve building blocks and is shown to work better than the single point crossover in most architectural structures. Da Silva et al. propose two new crossover operators for multi-objective search-based approaches [61]. They apply them using NSGA-II to the Product Line Architecture problem, optimizing for feature modularization, SPL extensibility, variability, coupling, and cohesion. The first crossover operator first adds all the parent's features to their respective children except for one, which is put into the other parent's child. After this, duplicate functionality is removed based on the one that has the worst feature modularization. The second crossover operator is similar to single-point crossover, but it only chooses a crossover point only in one parent. The child receives the first part from parent one. For the second part, the child receives elements from the other parent which are not yet in its first part. The remaining elements are received from parent 1 as well.

3.2.2 Machine Learning

Another type of approach used to tackle modularization is machine learning. Machine learning approaches for optimization problems often combine some sort of exploration algorithm with a neural network that evaluates the quality of solutions or parts of solutions.

Schwanke and Hanson demonstrate, back in 1994, a tool that uses machine learning in the form of neural networks to modularize software [60]. It uses more-than-similar judgments as features by constructing triples where $\langle S, G, B \rangle$ represents that S is more similar to G than S is to B. This tool is meant to be a software architecture assistant, which gives advice in the form of recommended changes. The tool is compared to nearest neighbor clustering and classification tools and is shown to have similar effectiveness. Zhang and Jacobsen introduce the Prism project, which discovers non-localized units of modularity by applying aspect mining based on type-based and lexical-based patterns [70]. These units are used for refactoring possibility recommendations, as they most likely fit better elsewhere.

Machine learning has been used successfully for code structure quality prediction in general as well, which means it could be used as a metric in an optimization algorithm. Thwin and Quah used a General Regression Neural Network to predict software quality [65]. They use the following metrics: Depth of Inheritance Tree, Number of Children, Coupling Between Objects, Response For

a Class, Inheritance Coupling, Coupling Between Methods, Weighted Methods per Class, and Number of Object Allocations.

3.2.3 Cutting & Clustering

As modularization is often represented as a type of graph, cutting and clustering approaches are another way of optimizing it.

Chong, Lee, and Ling propose a dendrogram cutting approach to cluster software [17]. The approach uses least-squares regression to find the best cut level. The value of the cut is determined by its effect on cohesion and coupling. Cohesion is represented by the average distance between each entity in a cluster, and coupling by the minimum distance between the two centroids of the sides of the cut. The distance between parts of the code is defined by 4 different similarity measures. It's stated that to properly use this approach, utility classes need to be removed from the simulations first, as those negatively affect the accuracy of the software clustering. Jermaine implements the k-cut method in order to modularize software [33]. The cuts are meant to create modules with high cohesion and low coupling. The edges are weighted by their sociability metric that represents the amount of inter- and intra-module function invocations. Marx, Beck, and Diehl propose a way to optimize a cut between a component and the remaining system, in order to reuse it in other software or outsourcing development [43]. They use a combination of exhaustive, hill climbing, and min-cut algorithms to extract components. The cut is optimized for the (possibly weighted) amount of class to class dependencies broken.

Kook Lee et al. introduce a component identification method that considers coupling, cohesion, dependency level, connection through interface, and granularity [37]. This approach makes use of a novel component clustering algorithm to cluster nodes. Nodes are made up of one or more classes. The clustering algorithm weighs the connection between nodes using cohesion based on core functionality similarity, and coupling modeled as static connections and the amount of message passing connections.

3.3 Comparison of Existing Modularization Approaches

Due to the vastness of existing works on modularization specifically, some studies have compared the different approaches. They identify which ones seem promising, and what seems to be missing from existing work.

Ebad and Ahmed, in their work, compare different approaches to the modularization problem [24]. They create a framework that measures approaches based on the given packaging goal, underlying principle, input artifact, inter-

3.3. Comparison of Existing Modularization Approaches

nal quality attribute, search algorithm, fitness function, scalability, soundness, practicality, and supportability. They observe the following aspects:

- Most see packaging as an optimization problem
- Most maximize intra-package cohesion and minimize inter-package coupling
- Most use Genetic Algorithms as their search methods
- All approaches are graph-based
- Most use source code as the input artifact
- Selection of parameters of the heuristic search method is very crucial (due to local optima)
- Most approaches are scalable
- The soundness of results is questionable
- Only one is supported by tools to easily apply it to a codebase

Mohan and Greer also performed a survey of search-based refactoring for software maintenance [52]. Several search techniques are used in the surveyed papers: Hill climbing, Simulated annealing, Genetic algorithms, and multi-objective evolutionary algorithms. The papers apply their approaches to projects that are generally adequate as they are large enough to justify them representing a real project. These projects generally consist of tens of thousands of lines of code. There was one study that used a large system with more than one million lines of code [55]. They conclude that software developer feedback is needed to support the approaches.

The two factors this thesis focuses on that are made clear in these review/comparison papers are the following:

- The assumption that these approaches are scalable, even though none seems to have been applied on a large scale codebase.
- The questionable quality of the results, caused by the evaluation of the results. This is measured by the metric improvement only, not by verifying the results with software developers that are experienced with the code.

As Mohan and Greer state in their paper: "It needs to be evaluated whether the search-based refactoring techniques that have been developed can carry over to the business environment or whether real-world application scenarios will bring to light further issues." [52].

Chapter 4

Adapting to Scale

To achieve one of the goals of this thesis, we need to find a way to successfully apply an approach similar to existing research on a significantly bigger scale. This chapter describes what Adyen is as a business and the effects on the codebase. The chapter also contains the identified challenges of the bigger scale and the reasoning behind choices that were made to tackle or minimize these challenges.

4.1 Adyen

Adyen is a payment service provider, gateway, acquirer, and point of sale terminal service¹. The company was founded in 2006 and has grown rapidly, totaling over 1400 employees as of August 2020. Most of the code is part of a monolithic repository, consisting of 5.5M+ lines of code. This is the codebase we will apply the approach to. Due to the emphasis on uptime, performance, and growth potential for this type of business, good code quality and with it code structure quality is essential. A codebase of this size and speed of growth brings additional challenges when compared to smaller codebases.

4.2 Challenges of the Scale

The most obvious challenge coined by the bigger scale is the complexity of the codebase. Instead of a piece of software with very focused functionality and consisting of tens of thousands or hundreds of thousands lines of code, we will be applying the approach to a piece of software with many different functionalities and responsibilities and which exists out of more than 5 million lines of code. This increases the size of the input the chosen algorithm will take, and through the input size, and differences between and amount of functionalities, increase

¹<https://www.adyen.com/>

the complexity of the software. This effect will be amplified by the complexity of the algorithms used in the approaches, which scale worse than linearly with the size of the problem.

The many different functionalities and responsibilities the codebase holds emphasize an aspect more than smaller codebases, namely transitive coupling. Transitive coupling can be described as follows: if module M1 depends on module M2, and module M2 depends on module M3, M1 is transitively dependent on M3. The reason this is further emphasized is because of the depth of these dependencies caused by the size of the codebase, the chance of different functionalities being dependent on code they do not need, and the loss of overview of developers. For a developer, understanding what every module is responsible for and which code it contains in a codebase with twenty small modules is doable, while the same task is infeasible to do for hundreds of modules supporting different functionalities. When this is combined with different teams of developers being responsible for different functionalities in a fast-growing codebase, unnecessary coupling is bound to arise. This unnecessary coupling causes more unnecessary transitive coupling. The negative effects of coupling mentioned in section 2.1.1 are amplified further through transitive coupling, especially if the modules that are being transitively depended on are sizable.

The speed of growth of the codebase as mentioned before, mostly caused by being enterprise software and the push for features created by business goals, poses a risk in code quality. The pressure can cause flaws to slip through, and the developers to take on a "fix it later" mentality. This implies that over time code quality will degrade if gone unchecked, potentially increasing the size of the problem.

Another aspect of the size and goal of the codebase which influences modularization as a problem is that in order to completely refactor and reorganize the codebase an immense amount of time and effort is needed. Because of this, a modularization solution that is very different from the current state of the modularization will be mostly useless in practice.

4.3 Algorithm

Given the challenges posed by the bigger scale, we need to choose an algorithm that will be able to find good solutions in an acceptable amount of time. As modularization is not a problem which needs to be fixed overnight, we will aim for an algorithm that can run for more than a day, and preferably less than a week. We also want to have a high diversity of solutions. In addition, the algorithm needs to be able to handle the complexity of the problem, and optimize for multiple variables.

The Non-dominated Sort Genetic Algorithm II (NSGA-II) [22] was chosen to

optimize this problem. This choice is supported by the fact that the amount of optimization variables we are aiming for is low (< 5), it one of the most used algorithms for modularization, and it has shown to be one of the most effective in modularization and similar problems [1, 2, 6, 61, 30, 50, 38, 53]. Due to the possible runtime, this algorithm can be applied on the entire codebase in one run, and make improvements in different parts. The fact that its result is a pareto front of solutions means it can and most likely will contain multiple suggestions for improvement each run.

4.4 Modeling & Representation

Modularization can be optimized on different levels. Often it is performed on a module level [36, 31, 46, 6], meaning the smallest changes are classes moved from modules to other modules. It has also been performed on a package or subfolder level [1, 38], making the problem harder than the module level. Recent research has shown the potential of doing it on class level, meaning the smallest changes are methods and variables moved between classes [51, 2].

To tackle the bigger scale and show whether modularization approaches can be extended to it, we need to respect the challenges it brings. Therefore, we choose to optimize the modularization on a module level.

Given that modularization solutions which restructure the entire codebase are useless in practice on this scale, we need to find solutions similar to the current structure of the codebase. More specifically than the modularization problem, we tackle the remodularization problem. A solution to the remodularization problem is a modularization that is better than the current modularization, while not requiring too much effort to achieve.

We model the software remodularization problem as a search problem where the search space consists of all possible modularizations. In other words, all possible groupings of classes in modules. All classes are part of exactly one module, which can be seen as a group of classes. Code contained in a class, and the class' dependencies are seen as unchangeable information. The problem is modeled as two graphs, one representing the dependencies on class level and one on module level. The class dependencies are unchangeable, while the module level dependencies are inferred from those class dependencies. If there exists at least one class dependency from module M1 to a class in module M2, there exists a module dependency from M1 to M2. This results in another graph that shows the module dependencies (see chapter 2).

A solution to this problem consists of a new and optimized dependency structure. In our algorithm, classes are moved to different modules to achieve better dependency structures. A solution in the algorithm is modeled as a list of tuples containing the classes that were moved and the modules they were moved to,

for example, see Table 4.1.

Table 4.1: An example of a solution to the remodularization problem

Move	To
example.package.class1	example-module
example.package.class2	example-module
other.package.class4	another-module

The choice to model the solutions as changes instead of a complete dependency structure was made to reduce the cost of evaluating solutions. Re-evaluating entire modularizations takes significantly more processing power than evaluating the difference that a list of changes makes. Also, when a solution is modeled as an entire modularization, other steps of the algorithm run significantly slower and the algorithm needs more memory. Modeling the solutions this way has been argued by Abdeen et al. [1] and implemented in other remodularization research [38, 50].

4.5 Estimated Build Cost of module Cache Breaks

The added emphasis on transitive coupling shows the need for a metric that measures it. To optimize for this properly, we want a metric that represents weighted transitive coupling. The metric we propose is the Estimated Build Cost of module Cache Breaks (EBCCB). The value is formed by summing for each module an estimation of its build cost multiplied by the number of times its cache was broken over a set period of time. This section will further explain the relation between caching and transitive coupling, and why this coupling is weighted by using module rate of change and build cost.

4.5.1 Caching

Caching is often used to remedy the fact that building code after making changes in enterprise software can take a long time. Module caches reduce the build cost of the codebase after a change. Instead of all modules, only the modules that changed and the modules that depend on the ones that are recompiled and rebuild need to be recompiled and rebuild. This causes any change to have a cascading effect throughout the codebase, in the reverse direction of the module dependencies. Removing a single dependency or splitting a module could have a significant effect on the number of caches broken. As this cache break effect follows the transitive dependencies, it represents the transitive coupling aspect of EBCCB.

4.5.2 Rate of Change

How volatile a module is (i.e. how often it changes) is represented by its Rate of Change (ROC). The ROC of a module is based on the number of commits that change one or more classes in the module. The modules are weighted by ROC because the more a module changes, the more likely it is to introduce errors, introduce new dependencies, or grow in size. An unwanted transitive dependency on a completely static module is less problematic than on a volatile one.

To calculate how many times a module's cache breaks, the ROC of itself and all its transitive dependencies are needed. All ROCs of the transitive dependencies need to be considered, for which the module dependency structure formed by a solution is used. If multiple dependencies of a module have their cache broken by the same commit, the module only has to be rebuilt once.

4.5.3 Build Cost

The build cost of a module gives an indication of how big and complex it is. Unnecessarily breaking the cache of a low build cost module has a smaller negative impact than that of a high build cost module. Similarly, bigger and more complex modules are impacted more by the negative effects caused by (transitive) coupling.

Building every solution is impossible, so instead an approximation is made. To approximate the build cost of a module, we build a function through regression that uses the lines of code (LOC) of a module to predict its build time. In other words, the LOC of a module combined with this function is taken as a proxy for the build cost. This value can be determined quickly and it can be easily re-evaluated after changes are made by subtracting the LOC of the moved classes from their original modules and adding it to the ones they were moved to.

To find the LOC of a module, the LOC of every class in it is summed up. A function is derived through regression of Y on X where Y is the module's build time and X is the module's LOC. The similarity between these values, specifically in the codebase this algorithm was applied on for the case study, can be seen in Figures C.1 and C.2 in Appendix C.

4.5.4 Metric value

Multiplying the estimated build cost with the number of cache breaks of a module results in an estimation of the build cost of cache breaks of said module. The summation of this for all modules is the total Estimated Build Cost of Cache

4. Adapting to Scale

Breaks. The resulting formula for the metric is the following:

$$EBCCB(S) = \sum_m LOCtoCost(LOC(m)) * Cb(m) \quad (4.1)$$

Where S is a solution, m is a module, LOCtoCost is the formula that translates LOC to the build cost, and LOC(m) is the lines of code of a module. Cb(m) is the number of cache breaks of a module, in other words, the number of commits that change a class in the module or in any of its transitive dependencies.

The pseudocode of this optimization variable can be found in Appendix D.

Chapter 5

Constructing & Fine-Tuning the Algorithm

In a nutshell, our algorithm works as follows. The problem is modeled as two graphs, one representing the dependencies on class level and one on module level. A solution is represented as a set of classes that are moved from their original module to a new one. The used optimization variables consist of a combination of some or all of the following values: a coupling metric, multiple cohesion metrics, amount of classes moved to a different module, and expected build cost of cache breaks. Selection is used in the same way as in the introduction of NSGA-II. Crossover is done by two operators: one simple single-point crossover, and one that tries to preserve good aspects of a solution. Mutation is performed by a min-cut based operator and by a random neighborhood operator, which both determine a group of classes to move from one module to another. The problem has one constraint, namely that there should be no circular dependencies. The details of these aspects are stated in the next sections of this chapter.

This chapter describes how the remodularization problem was approached. It starts with how the problem was modeled and which algorithm was used. After that, it goes into the specific aspects of the algorithm, which variables were optimized for, and the constraints the problem has. Finally, it describes the tuning process applied to achieve proper performance.

5.1 Optimization Variables

We want to optimize how well the architecture is modularized, so we experiment with different metrics:

- Inter-Module Coupling (InterMD), also referred to as Coupling, is measured by the total amount of module dependencies.

5. Constructing & Fine-Tuning the Algorithm

- Cohesion of modules. We experiment with 3 different metrics used as separate optimization variables:
 - The Common Closure Principle (CCP), which is measured by the sum of the number of classes in a module changed in the same commit, for each module and every commit over a set period of time[42].
 - The Common Reuse Principle (CRP), which is measured by the number of classes used together from a module for each class. [42].
 - Intra-Module Coupling (IntraMD), namely the number of intra-dependencies (class dependencies in a module) divided by the maximum possible amount of intra-dependencies [31].
- The effort required to achieve the solution, modeled by the amount of "move class" refactorings, i.e. the number of classes that have been moved to a different module in the solution.
- The Estimated Build Cost of module Cache Breaks (EBCCB) over a set time in the future.

The optimization variables representing coupling and cohesion are often proposed as code quality measures [44, 41, 33, 32, 12, 8, 5]. They are also often used in combination with search algorithms to improve the code quality of a codebase [1, 53, 38, 56, 51, 46, 39, 31, 17, 6, 3, 2].

For coupling, the average amount of inter-dependencies of modules was used. This metric is used very often in search-based (re)modularization and refactoring research [1, 33, 17, 38, 6, 46, 51, 56, 53]. For cohesion, several values are tested. One is the value used originally proposed as a part of Modularization Quality by Mancoridis et al. [40], and used for a similar problem by others [31, 46, 47, 36, 56], two others are, as stated by Robert Martin, the Common Closure Principle (CCP), which aims to minimize the number of packages that would be changed in any given release cycle, and the Common Reuse Principle (CRP), which states that classes that are reused together should be packaged together [42]. The Acyclic Dependencies Principle (ADP) is ignored because it is inherent to the EBCCB, as calculating this optimization variable will reveal any circular dependencies. Usage of CCP, CRP, and ADP for re-modularization is also proposed by Abdeen et al. [1].

The amount of changes needed to achieve the solution covers another aspect of modularization, namely the effort required to achieve the solutions. As stated before, many changes can be made to the codebase, however, the effort needed might not be worth the gain in quality. As the architecture is from enterprise software, and the goal is to improve it by moving classes and iteratively running the algorithm, it does not make sense to generate solutions that require moving thousands of classes. Finding solutions that achieve similar

quality gains with fewer changes is preferred. Therefore the amount of changes needed to reach the solution is one of the optimization variables.

Finally, *Estimated Build Cost of Cache Breaks* (EBCCB, see section 4.5) combines a form of transitive coupling with the size and rate of change of modules and represents a tangible value that shows the expected build cost of cache breaks based on time. It represents a form of weighted transitive coupling.

5.2 Selection

For selection, the partial order and crowding distance of solutions were used in combination with simple tournament selection with a tournament size of 5, as has been proposed in the original paper introducing NSGA-II [22]. The partial order defines which solutions are taken to the next generation and which fall off. Solution i is chosen over j if i has a lower rank than j , meaning it is part of a better front. If i and j have the same rank, crowding distance decides which order the solutions take. Crowding distance is calculated by first sorting the population according to each optimization variable. The solutions with the maximal and minimal value for one of the metrics are assigned an infinite distance value. The others are assigned a distance value equal to the absolute normalized difference in an optimization variable of two adjacent solutions. This process is repeated for the other optimization variables. Each optimization variable value is normalized beforehand. Solutions with a higher crowding distance value are chosen over solutions with lower values.

5.3 Crossover

Crossover is the process of creating new solutions (children) by combining parts of solutions that are in the current population (parents). For crossover, two operators were tested:

- A single-point crossover. This crossover operator makes a sorted list of the classes that were moved in both parents. It picks a random point in this list. All changes before the point that belong to parent 1 are duplicated in child 1, and the ones that belong to parent 2 are duplicated in child 2. For all changes after this point, the ones that belong to parent 1 are duplicated to child 2, and from parent 2 to child 1. The list of classes is sorted consistently during a run of the algorithm, based on the order they were in when given as input.
- A crossover operator that tries to preserve building blocks, which in this case are multiple changed classes that only have a positive effect when moved together. This operator is based on the crossover operator used in

5. Constructing & Fine-Tuning the Algorithm

Harman et al. [31]. It aims to preserve building blocks by doing a uniform crossover based on modules instead of classes. This means that all classes that are in a module in one parent are set to be the same in one child, and the classes in the same module in the other parent in the other child. This is repeated for each module, pairing the two children with the two parents randomly for each.

Performing crossover in this way sometimes causes conflicts, as a class might be in module A in parent 1 and module B in parent 2. If both of these modules are added to the same child, the class should not be in both. So as soon as a class from one parent is added to one child, the same class from the other parent is added to the other child. This process can be seen in Figure 5.1.

In this figure, classes are the circles labeled with numbers, and the rounded squares are the modules they are in. The figure shows the two first steps of the crossover process. In the first step, module 1 (M1) is randomly chosen. This module is copied from parent 1 to child 1, marked in blue, and from parent 2 to child 2, marked in green. To prevent duplicates, class 2 is copied from parent 1 to child 1, marked in red. In the second step, module 2 (M2) is copied from parent 1 to child 2 and from parent 2 to child 1, marked similarly in terms of colors. Note that class 2 is not copied from parent 1 to child 2, as it is already present. We end up with 2 child solutions which have the placements of class 2 and 4 mixed with respect to the parent solutions.

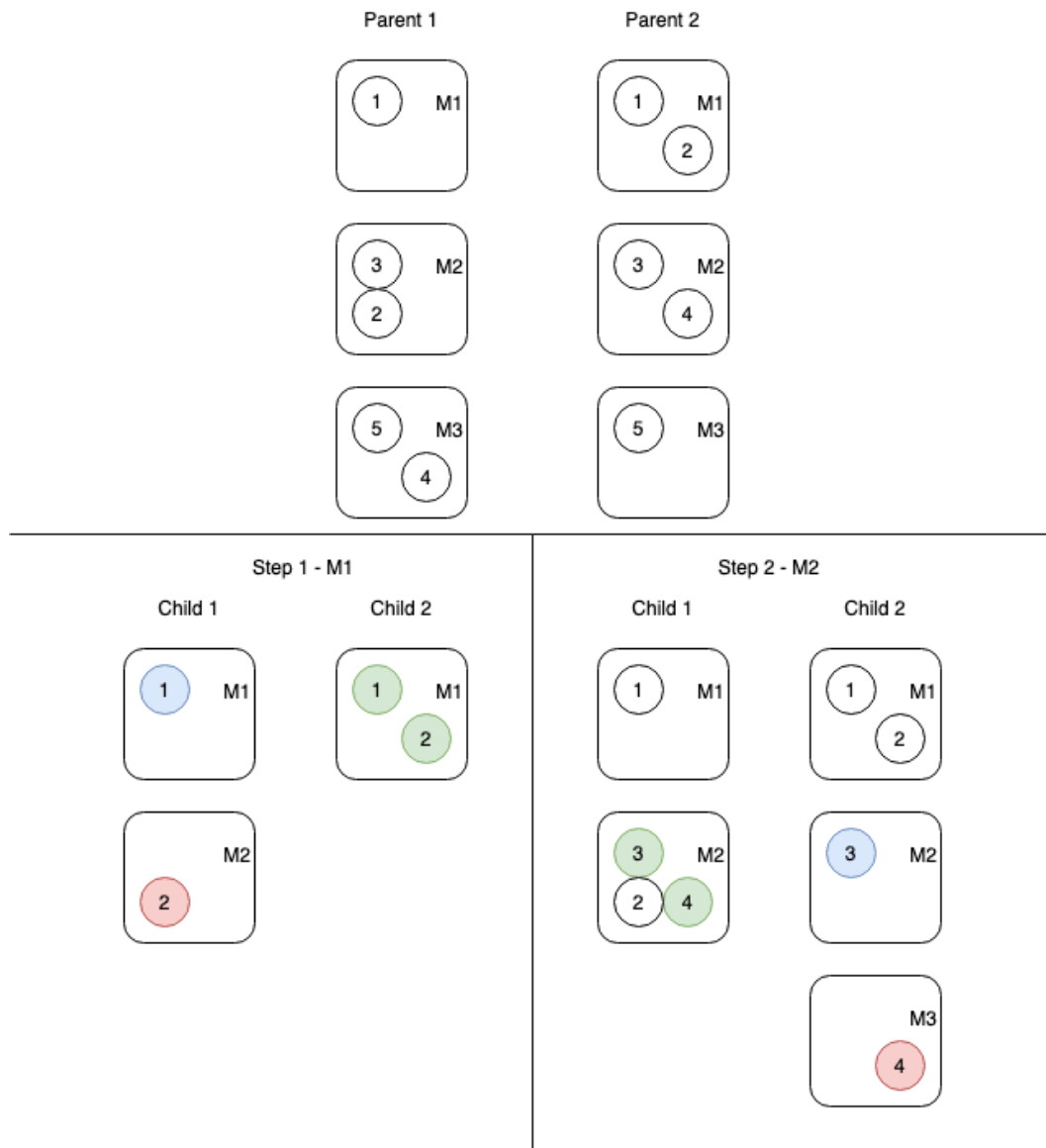


Figure 5.1: Crossover operator: effects of trying to preserve building blocks and prevent duplicates. Two modules (M1 & M2) are copied to child solutions in two steps respectively. Classes copied from parent 1 are colored blue and ones copied from parent 2 are colored green. The classes that are copied from parent 1 due to preventing duplicates are colored red.

5.4 Mutation

The most important factor of mutation in this problem is the ability to move both a small amount and a large number of classes from one module to another. Two were chosen that can do the following:

- A min-cut-based operator, which picks a random class and adds it to a graph, then it traverses all dependencies from and to that class and adds all the found classes to the graph. It keeps repeating this process until a certain size is met, which is limited due to the computation cost associated with min-cut. The operator then performs the Stoer-Wagner algorithm to determine the min-cut of this graph [64]. The initially chosen class is in one of the resulting subgraphs. This subgraph is then moved to a different module. This operator is inspired by existing research that used graph cutting algorithms for modularization [17, 33, 43].
- A mutation operator that moves a neighborhood of classes in a module of a random size to another module. This operator was introduced by Fraser and Arcuri [28]. It can perform anywhere between 1 and many mutations, with increasingly less chance for higher amounts. The mutation operator was adopted to this problem by changing the mutation to moving classes in the following way. It picks a random class, adds it to a set, then keeps adding random classes in the neighborhood until the following condition is met: $r > (c)^n$ where r is a random number between 0 and 1, c is a constant between 0 and 1 which impacts the average size of the resulting set and n is the size of the set. The neighborhood is defined as all classes which are in the same module, and depend on a class in the graph or are depended on by a class in the set, but are not in the set. This set is then moved to a different module.

The module the classes are moved to is chosen randomly from the set of modules that the neighborhood of classes is connected to with one added module which is a new, empty one. Limiting the moving of classes to modules that the neighborhood of classes is connected to originated from trying out several configurations of the algorithm. Moving the classes to a module that is not connected to the neighborhood of classes almost always results in the classes being in a module that they do not fit in. In terms of Cohesion, classes are moved to a module comprised of classes it is not connected to in terms of dependencies. In terms of Coupling, new dependencies are introduced to or from the new module.

5.5 Constraints

There is one significant constraint in this problem: solutions should not contain a circular dependency. There exists a circular dependency if any module can traverse its dependencies (and its dependencies' dependencies etc.) and find itself. If a solution has a circular dependency the code is impossible to build. While this constraint should not be broken by the solutions in the output of the algorithm, it is allowed to break it during, so the algorithm can explore more. To make sure the pareto front does not exist of only solutions containing a circular dependency, these solutions should be dealt with in some way.

5.5.1 Punishment

Punishment was performed by lowering the value of solutions that break constraints, based on how many constraint violations they have. Constraint violation results in a circle in the module dependency graph. The size of this circle is multiplied by a set value, which is then added to the estimated build cost. Also, solutions with fewer constraint violations are chosen over solutions with more violations during selection, as proposed by Deb et al. [22].

5.5.2 Repair

Solutions that break constraints can also undergo an attempt to repair. If a solution contains a circular dependency somewhere, it is caused by one of the newly added dependencies when compared to the original module dependency structure. Changes made that cause this dependency are reverted to make it disappear. Given the modules that are in a circular dependency, it is trivial to find these dependencies, namely, they should be between any of these modules. These dependencies can include ones that do not cause the circular dependency. Differentiating between them is not trivial. In order to remove the circular dependency and repair the solution, all of the classes moved to the modules which are part of the circular dependencies are moved back to their original module. This can affect the quality of solutions negatively by reverting good changes, but it is an efficient and guaranteed way to remove circular dependencies.

5.6 Population Initialization

Because the modularization problem requires the resulting modularization to be similar to the one given as input, we can not initialize the population by creating random modularizations. Also, due to how quickly the circular dependency constraint is broken, solutions initialized by making a set of random class changes rarely adhere to the constraints. Instead, we initialized the population

by mutating each solution for a random number of times between 0 and 100, using the mutation operators described earlier in this chapter.

5.7 Optimization Variable & Parameter Tuning

To get an adequate performance out of the algorithm, we need to choose a proper combination of optimization variables and we need to tune its parameters. Tuning requires a significant amount of algorithm runs, and the possible amount of combinations across optimization variables and algorithm parameters is too high. To remedy this, the algorithm was run on a reduced version of the codebase. This version consists of 17 modules, with a total of 10447 classes. These modules were chosen to represent multiple functionalities of the codebase and to retain the depth of the codebase in terms of transitive module to module dependencies.

5.7.1 Optimization variables

Optimizing for all of the optimization variables stated in section 5.1 using the NSGA-II algorithm is not preferred. For every optimization variable added the pareto front gains a dimension, which increases the optimization difficulty exponentially. In order to keep the pareto front properly spread for all dimensions also requires the population size to be bigger, adding to the runtime. Instead, we want to find combinations of optimization variables that achieve relatively good results in terms of the metrics.

To start we took default values for the parameters based on existing research and intuition formed by running the algorithm with differing configurations (see Appendix section C.3). Using these parameters, 11 different configurations of optimization variables were tested. These combinations of optimization variables were tested by running the algorithm, using the default parameters, 5 times for 3 hours per combination. The configurations and results of these runs can be seen in Appendix C.2. The best configuration found given the improvement on all fronts and an adequate amount of generations is using *IntraMD*, *InterMD*, *EBCCB*, and *# Changes* as optimization variables. These runs also form a baseline which shows that optimizing for EBCCB is required to achieve significant improvement in terms of EBCCB. In other words, it cannot be optimized by optimizing any of the other parameters tested.

5.7.2 Parameters

Given the better combinations of optimization variables from the previous subsection, we can tweak the algorithm parameters. For each possible parameter, several possible values were determined from related research and intuition.

These possible values can be seen in Appendix section C.3. Given the large number of possible combinations and the computational cost required to run them all, we opt for tuning a smaller set of parameters, specifically tuning one parameter at a time while using a default value for the others. 7 parameters were tuned, resulting in 17 different configurations. The algorithm was run 5 times for 3 hours for each configuration. The results can be seen in Appendix section C.3. The resulting parameter values are as follows:

- Population size = 500
- Mutation chance = 0.5
- The optimal ratio of mutation operator usage = 0.5
- Constraint breaks are ignored
- Duplicates are deleted
- The crossover operator that tries to preserve building blocks is used.
- An elite archive is used.

Chapter 6

Empirical Study: Approach in Theory

To first show the usefulness of the approach in a controlled way, we applied it on the Adyen codebase. After running the algorithm for a limited time, solutions should be found that improve the code structure in terms of the metrics. After that, we ran the algorithm for a longer period, to see if and when it converges to see the feasibility of the approach on this scale.

6.1 Research Questions

With the configuration of the algorithm described in the previous chapter, we want to answer the following research questions.

RQ1: What is the improvement of the code structure in the solutions that the proposed approach finds in terms of the metrics used as optimization variables? Are the algorithm and its configuration sufficient to get results that show a significant improvement of the codebase? If so, how big is this improvement?

RQ2: How fast does the proposed algorithm converge? How many generations does it take for the algorithm to converge to a local optimum? From the improvement of the pareto front and the individual optimization variables throughout the generations, we can infer when the algorithm has converged to a local optimum. This shows how long the algorithm will realistically take to do a proper run.

6.2 Methodology

We run the algorithm on the entire codebase for 24 hours on 8 cores at 2.1GHz with 8GB of RAM, 2 times. For RQ1, we look at the values of the chosen op-

timization variables in the resulting solutions. For RQ2, we save the values of the optimization variables of the pareto front every 50 generations and run the algorithm for 72 hours on the same machine to see when the improvement stops.

6.3 Results

This section contains the results that show the improvement that the algorithm shows and its convergence speed, which hold the answers to the first 2 research questions.

6.3.1 RQ1: What is the improvement of the code structure in the solutions that the proposed approach finds in terms of the metrics used as optimization variables?

The results of the runs done to answer RQ1 can be seen in Figures 6.1,6.2,6.3 & 6.4, and Tables 6.1 & 6.2. The figures show the pareto fronts of the two runs. These tables contain the relative improvement of the optimization variable values of the solutions that have the best value for one of the optimization variables. Table 6.1 only contains solutions that improve on all optimization variables, and Table 6.2 contains the solutions with the best improvements for each optimization variable regardless. These tables and figures and tables show relative improvement instead of the absolute values as per Adyen’s request.

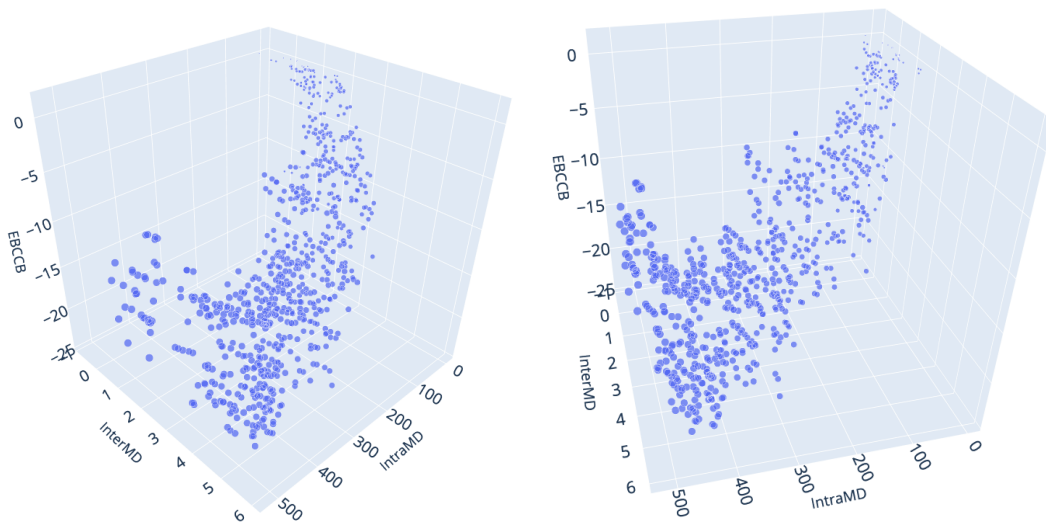


Figure 6.1: Pareto front of the first run¹

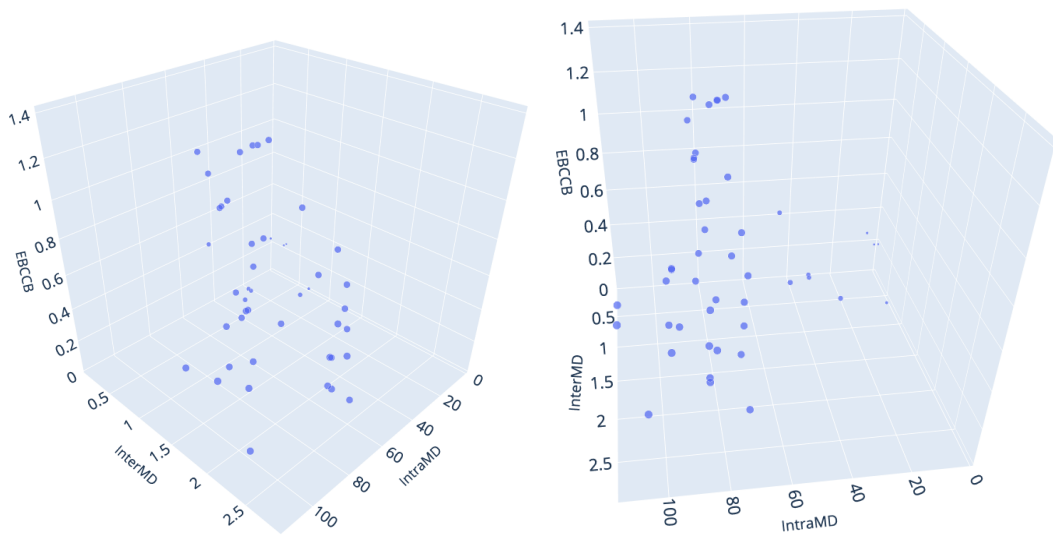


Figure 6.2: Solutions that improve on all metrics from the Pareto front of the first run²

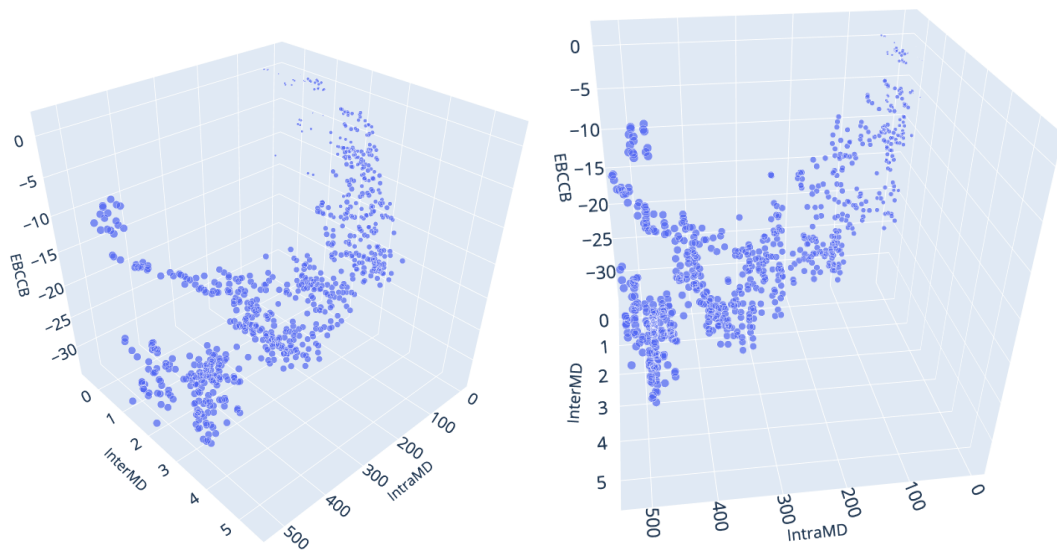


Figure 6.3: Pareto front of the second run³

¹An interactive 3d visualization of these figures can be seen in the links in the footnotes of each figure. https://github.com/SERG-Delft/ga-remodularization-appendix/blob/master/run1_3dview.html

These figures can also be downloaded as a zip file from <https://doi.org/10.5281/zenodo.4011987>

²https://github.com/SERG-Delft/ga-remodularization-appendix/blob/master/run1_3dview_filtered.html

6. Empirical Study: Approach in Theory

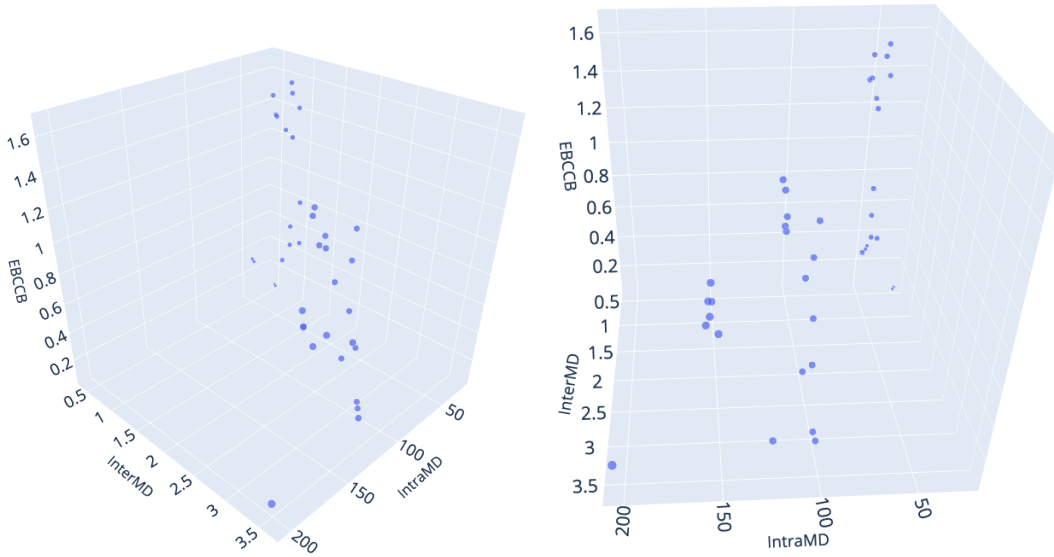


Figure 6.4: Solutions that improve on all metrics from the Pareto front of the second run⁴

Table 6.1: Relative improvement of best solutions for each optimization variable, that improve on all optimization variables.

	Best IntraMD	Best InterMD	Best EBCCB
IntraMD (cohesion)	205.173%	151.335%	55.011%
InterMD (coupling)	3.567%	3.656%	1.226%
EBCCB	0.168%	0.916%	1.667%
#Class change refactorings	170	138	67

Table 6.2: Relative improvement of best solutions for each optimization variable

	Best IntraMD	Best InterMD	Best EBCCB
IntraMD (cohesion)	530.404%	322.315%	55.010%
InterMD (coupling)	1.204%	6.041%	1.226%
EBCCB	N/A	-8.102%	1.667%
#Class change refactorings	418	244	67

³https://github.com/SERG-Delft/ga-remodularization-appendix/blob/master/run2_3dview.html

⁴https://github.com/SERG-Delft/ga-remodularization-appendix/blob/master/run2_3dview_filtered.html

Observation 1: Solutions show a significant improvement in code structure quality in terms of metric values. From the solutions that improve on all optimization variables and improve the most on one of them, improvements of 205.173% for IntraMD, 3.656% for InterMD, and 1.667% for EBCCB can be seen. When disregarding the quality of other optimization variables, potential improvements of 530.404% for IntraMD, 6.041% for InterMD, and 1.667% for EBCCB can be seen. These results show a significant improvement in all metrics.

Observation 2: The algorithm is prone to create solutions with many new modules consisting out of 1 class, due to the IntraMD metric. The percentage improvement of the IntraMD metric implies that the best solution in terms of IntraMD that improves on all optimization variables supposedly triples the value of the codebase in terms of cohesion. This is mostly caused by the fact that creating a new module with one or two classes adds significantly more to the value of the metric than improving the structure of bigger existing modules. This results in a value that does not scale linearly with the actual cohesion quality of the codebase.

Observation 3: The best solution in terms of EBCCB improves upon the other used metrics as well. Disregarding the other two metrics, the best solution in terms of EBCCB still improves upon those metrics. For InterMD, this can be explained by the total transitive build-cache breaks reducing by removing module to module dependencies, which is needed to improve on the coupling metric. For IntraMD, splitting modules into smaller ones and having class dependencies inside of modules instead of to other modules can be beneficial for both IntraMD and EBCCB.

Observation 4: The optimal solutions per metric need different amounts of changes, with EBCCB needing the least and IntraMD needing the most. The optimal solutions for EBCCB seem to need the least amount of changes, while the ones for IntraMD seem to need the most. A possible explanation for the IntraMD is that the more classes are moved to small or new modules, the more the value increases. To improve the EBCCB value, specific changes have lots more impact than others, so fewer changes are needed for a relatively big impact.

RQ1: Our approach shows a potential 205.173% improvement on the IntraMD metric, 3.656% improvement on the InterMD metric, and a 1.667% improvement on the EBCCB metric depending on the implemented solution. These solutions improve the codebase in terms of all used metrics.

6.3.2 RQ2: How fast does the proposed algorithm converge?

In Figures 6.5, 6.6, and 6.7, the improvement of each optimization variable over generations is seen.

Observation 5: For InterMD and EBCCB, the algorithm seems to converge. In contrast to the first observation, for the coupling metric, the algorithm seems to be close to convergence at generation 19000, and for the EBCCB, it seems to be mostly converged, only finding minor improvements after generation 14000. This shows that the algorithm converges after approximately 20000 generations.

Observation 6: For IntraMD, the algorithm has not yet converged. The reason why it has not converged yet is likely because the optimal solution for this metric is a structure constructed of modules with one class per module. The mutation operators can easily move a single class to a new module, which keeps improving solutions in terms of this metric over generations. This process can continue until the one class per module structure is achieved.

Observation 7: The achieved number of generations is close to 20000 after 72 hours.

RQ2: Our approach converges after approximately 20000 generations in terms of the InterMD and EBCCB metrics. It most likely does not converge within a reasonable time in terms of the IntraMD metric.

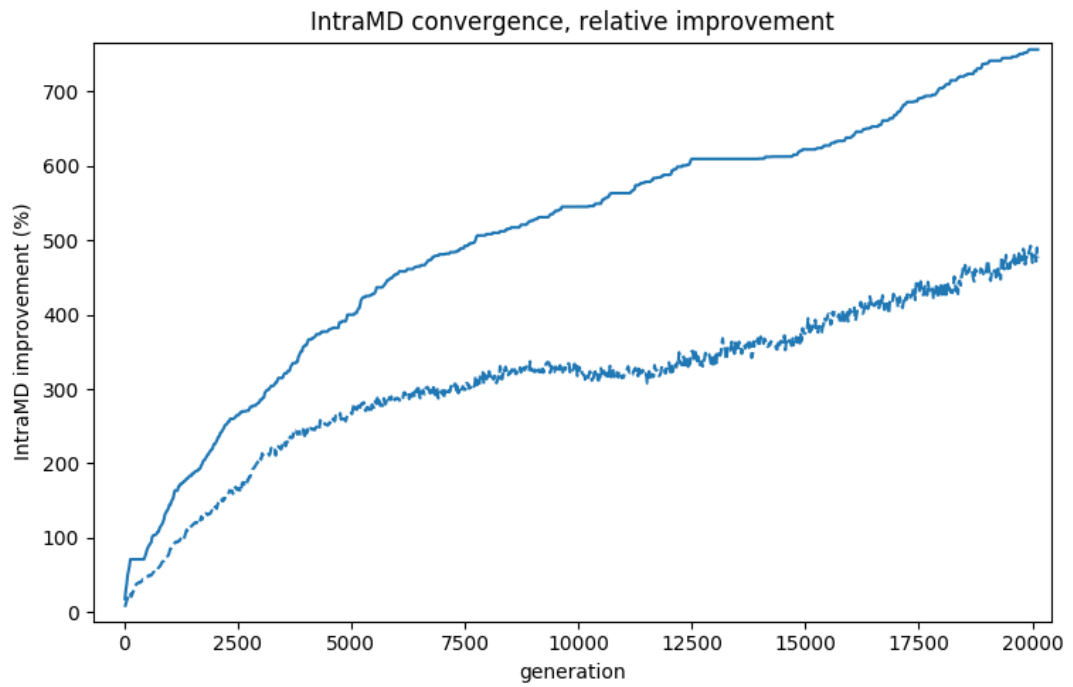


Figure 6.5: IntraMD convergence of the 72 hour-long run, best and average values of the pareto front

6. Empirical Study: Approach in Theory

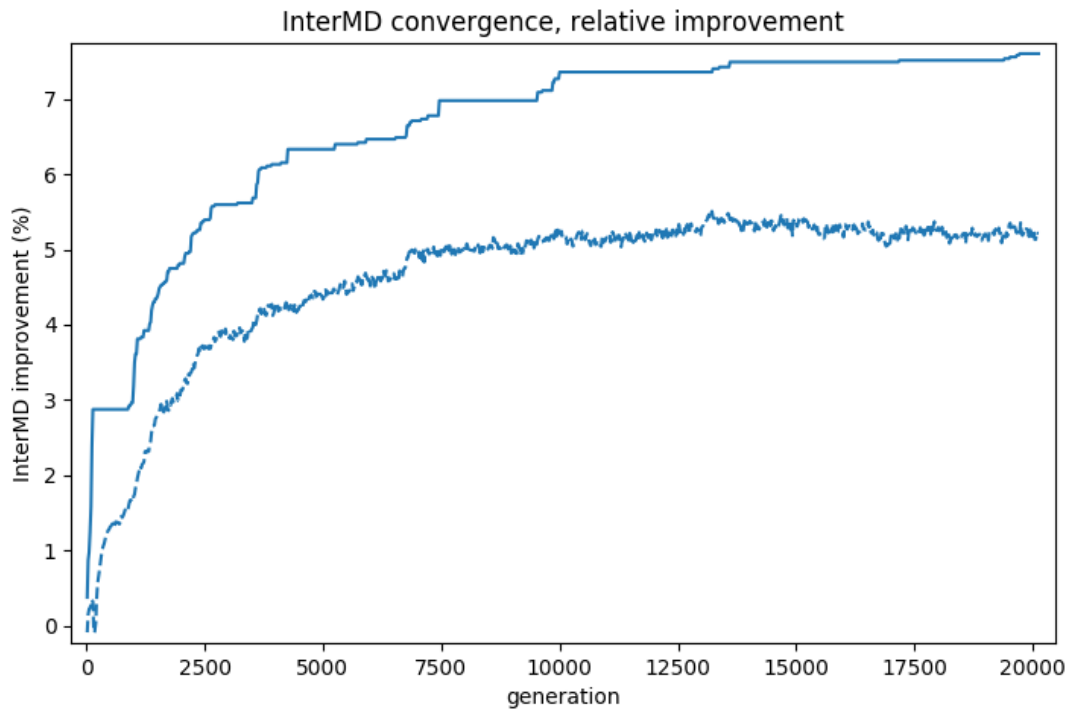


Figure 6.6: InterMD convergence of the 72 hour-long run, best and average values of the Pareto front

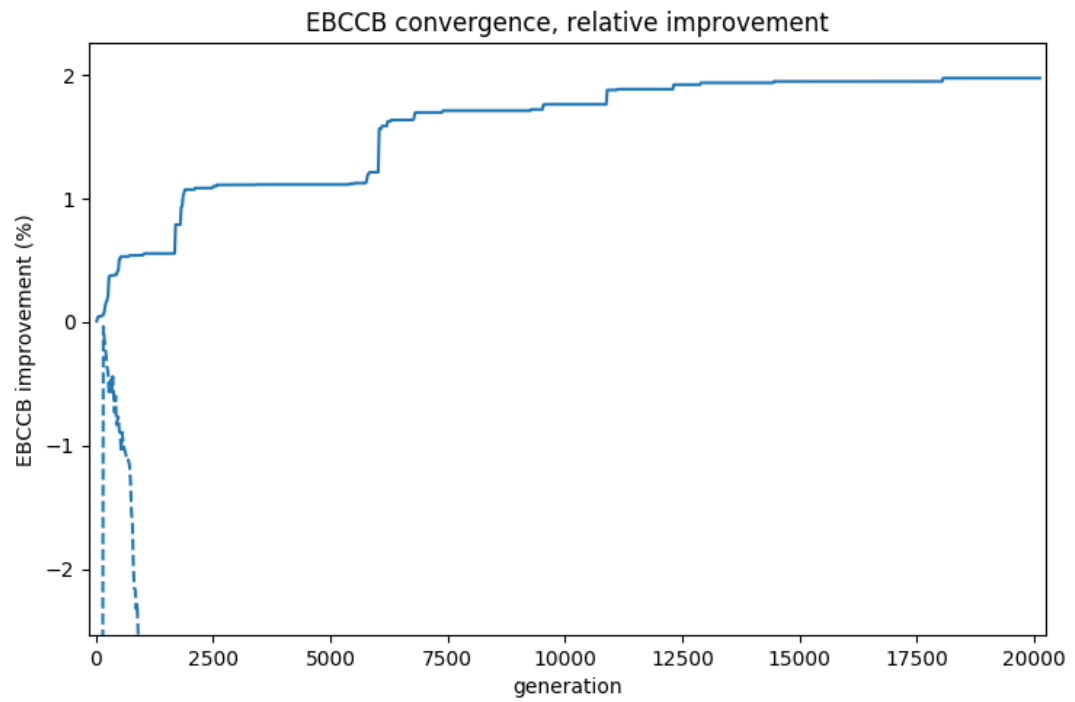


Figure 6.7: EBCCB convergence of the 72 hour-long run, best and average values of the pareto front

Chapter 7

Case Study: Approach in Practice

To find whether the solutions found by the proposed approach are useful from a realistic and business perspective, we need a case study where the subjects are developers that are experienced with the platform. They should be presented with changes suggested by the algorithm. From inspecting which changes they deem to be appropriate and which not, we can infer whether the solutions are useful in practice.

7.1 Research questions

RQ3: How do developers perceive the modularization suggestions of the approach? The fact that the change suggestions generated by the algorithm improve the code structure quality from a metric based point of view does not prove that they are useful in a realistic scenario. They might go against design patterns or make no sense from a developer's point of view.

7.2 Methodology

To answer RQ3, we interview several developers experienced with the code in question about a set of suggested refactorings. We take groups of refactoring suggestions from the best solutions resulting from the algorithm runs done to answer RQ1. The groups consist of up to 10 refactor suggestions. We choose a developer that has made multiple changes to classes in this group over the past 6 months to review the suggestions. If no such developer exists or can be interviewed, a senior developer related to that part of the code is chosen. The developer is first asked some contextual questions, like their experience and understanding of the module dependency structure. They are asked to review a set of class refactor suggestions, and reason about its quality. The answers

from these interviews are analyzed to determine the potential quality of the modularization approach in practice.

7.2.1 Suggestions

In order to make the solutions generated by the approach reviewable and verifiable by developers, we need to split them into smaller groups. The solutions generated by the algorithm consist of tens or hundreds of class refactorings, often in different parts of the codebase. If a developer is expected to review the suggestion in 10 minutes, it should not consist of more than 10 classes, preferably related classes. The reviewed changes should also not be taken from one solution. To achieve this, we separated all suggested class refactorings based on the module the classes are now in, per solution, and filtered the duplicates. From these smaller groups, we took only the ones that improved on at least 2 of the 3 metrics (excluding the number of class moves) and did not worsen the other. An exception was made for the coupling metric, which was allowed to be worsened if the EBCCB was improved because the overall coupling might improve and otherwise suggestions containing the creation of new modules would be very unlikely. This process resulted in 13 suggested change groups, which can be seen in Table 7.1.

Table 7.1: The value of the suggested class move refactoring groups in terms of the optimization variables. Note that for InterMD and EBCCB, lower values are better

	#class moves	IntraMD	InterMD	EBCCB
C1	1	1.002	7	-3339.428
C2	4	2.000	0	-2844.694
C3	1	0.083	-2	-536.365
C4	2	0.985	-4	-2864.221
C5	2	0.006	-7	-8259.967
C6	1	1	3	-9302.729
C7	1	0.007	-2	-146.522
C8	1	0.015	-1	0
C9	2	2	9	-18901.665
C10	1	0	-1	-6681.924
C11	1	0.083	-5	-3305.340
C12	1	0.004	-2	-2899.092
C13	7	1.000	-1	-3376.606

7.2.2 Interview Design

The goal of this interview is to see whether the supposed value of the solutions generated is representative of the value in a real scenario, and what practical value the solutions hold. The questions of the interview are formed in such a way to first verify the experience and knowledge of the interviewee from which the judgments will be made. After this, they are told a summary of the research, specifically for which metrics the code structure was optimized. Then they were asked to verify the code suggestions.

Afterwards, if the code change suggestions are judged to be fitting, the interviewee will be asked to rate the importance of the changes. If the code change suggestions are judged to be unfitting, it could still be an improvement to move the classes to a different module, or it could indicate an issue in the code that has to be fixed in a different way than moving classes. To verify this the interviewee is asked whether looking into this change has given them an idea of how to improve the code in this area. The list of interview questions can be seen in Appendix B.

7.2.3 Interview Procedure

The interviewees were asked to participate in the survey over the internal chat service in direct messaging. The message also contained a summary of what the survey was about, what the goal was, and the reason why they were chosen to participate. The specific module that the classes from the suggestion were in was also included.

The interviews were conducted in person and followed the structure stated in the previous subsection. During the interview, answers to the open questions were reduced to key sentences and written down. These key sentences were repeated back to the interviewee to verify whether it represents their reasoning. The data was analyzed by combining these answers on their key reasoning points.

7.2.4 Developers

The interviewees were chosen by finding developers that made a code change to at least one of the classes in the suggested change, in the past 6 months. If no such person could be found, was not available, or did not feel comfortable judging the suggested changes, a senior developer with experience with the module the classes are in was chosen. This resulted in 11 developers judging 13 different suggestions. Their experience can be seen in Table 7.2. 4 of the developers were seniors instead of developers that made changes to the classes. D1, D6, and D8 looked at 2 suggestions per person, and D4 and D7 looked at the same suggestion.

7. Case Study: Approach in Practice

Table 7.2: The interviewed developers and their experience. Developers that were chosen as seniors to review a suggestion are marked with (s).

Developer	Software Development Experience (Years)	Adyen Experience (Years)	Reviewed Suggestions
D1	9	4	C1 (s), C2 (s)
D2	3.5	3.5	C3
D3	10	2.5	C4
D4	10	0.5	C5
D5	20	7	C6 (s)
D6	6	3	C7, C8
D7	3	3	C5
D8	10	2	C9, C10 (s)
D9	34	1.5	C11 (s)
D10	2.5	2.5	C12
D11	6	2.5	C13

Table 7.3: Whether suggested class move refactoring groups correctly identify a flaw in the code, and properly fix it.

	Correctly identifies a flaw	Accurate refactoring suggestion
C1	✓	x
C2	✓	~
C3	✓	✓
C4	✓	~
C5	✓/ x	✓/ x
C6	✓	x
C7	✓	x
C8	✓	✓
C9	✓	x
C10	✓	x
C11	✓	✓
C12	✓	x
C13	✓	~

7.3 Results

All interview answers can be seen in Appendix B. A summary of the quality of the reviewed class move refactoring groups can be seen in Table 7.3

Observation 8: All suggestions revealed possible improvements of the code according to the developer. There is one exception where the developer saw no possible improvement, however, that same suggestion was deemed to be good by another developer.

Observation 9: About half of the suggestions were deemed to be partially good by developers. 7/13 suggestions were deemed to be at least partially good by developers, meaning that a part of it could be implemented directly, while the rest needs different fixes or needs to be ignored. 4 of those suggestions could be directly implemented from the suggestion as a good change to the current state of the code. The other changes were bad for one or more the following reasons: the classes fit better in their current module or in a different module which was not suggested, the classes are dead code or superseded by other functionality so they should be deleted instead, future plans concerning the classes hinder the suggested change, and/or the module the classes are moved to are in the process of being split up.

Observation 10: All suggestions of moving classes to new modules were reviewed negatively. This is likely due to the size of the suggestions examined because the new modules were too granular. Fitting modules existed already for these classes. This could be caused by the splitting process. Most suggestions consist of moving less than 5 classes, which would cause the new module to be too granular. This does not necessarily imply that these types of changes are bad, as in one case the new module was better, while it was not suggested by the algorithm. This case required more classes from different modules to be moved to this new module than were in the suggested to be moved.

Observation 11: Different developers make different judgments and conclusions. In the case of C5, 2 developers gave completely contradictory judgments on the suggestion. D4 determined C5 to be bad and saw no possible improvements, while D7 judged it to be good.

Also, D2 and D6 had similar reasons for judgment on C3 and C7 but made differing conclusions. In both cases, the suggestion would be good in the current state of the code, but given future plans, the code is where it should be. This caused D2 to conclude that the move should be implemented and reverted when needed, and D6 to conclude that it should not be implemented.

Observation 12: Developers rate the severity of the suggestions higher than their priority on average. Often the severity of the suggestions that are identified to be at least partially good is higher or equal to its priority.

RQ3: All suggestions helped the developer identify a flaw in the code, and form an idea on how to fix it. 4/13 suggestions could be implemented directly, and 3/13 could be partially implemented directly.

Chapter 8

Discussion

This chapter describes the implications given by the results, the potential practical uses of the approach, and the threats to validity.

8.1 Implications

From the results shown in the previous two chapters, two important implications become clear. The first is that the approach can be scaled to enterprise-level codebases. It shows that this type of approach can converge and show a significant relative improvement as when applied on a smaller codebase. The second is that the results must be validated in a real setting. The metric improvement does not necessarily translate to real quality improvement. The combination of these implications shows that successfully applying this type of approach in a business setting can be done if the results of an approach are shown to be sound through validation, and is scalable similar to the approach in this thesis.

Our results imply that the results support the validity of the EBCCB metric. The algorithm converges in terms of the metric, not many changes are required to achieve significant improvements, and code change suggestions that mostly improved on the metric were validated by developers. The best solution in terms of the EBCCB metric improves on both the cohesion and coupling metric. For the coupling metric, this can be expected, but for the cohesion metric, it implies that some of its value is captured in the EBCCB metric.

It is also implied that the improvement of the solutions shown by the metrics cannot be taken at face value. It is shown that around half of the subgroups of suggested class refactorings contained improvements according to the interviewed developers. Also, the importance of suggestions being correct is shown, due to the low priority the developers deem the suggestions to be.

8.2 Practical Use

From a business perspective, it is interesting to explore potential practical uses of the approach explored in this thesis. Due to how the suggestions helped the developers to identify flaws, a potential practical use can be constructed in a similar way to the experiments performed. For example, the algorithm could be run as a weekly job. The solutions generated from this run are then reduced to smaller class move refactor groups in the same way as was done for the interviews. The developers that recently changed the included code are notified with the suggestion and asked to take a short time to find the flaw in the code and fix it.

Another option would be to use the algorithm as an aiding tool in refactoring, specifically splitting modules that are too large. The changes made by mutation can be limited to the specific module or group of classes that need to be refactored. With this limited scope, the algorithm can be run for a relatively short time on the developer's machine. The solutions created might contain the refactoring the developer is looking for, it or could reveal information. Also, if the developer has found a way to refactor the code, it can be verified by running the metrics that the algorithm optimizes for on the adjusted codebase.

8.3 Threats to Validity

We acknowledge the threats to validity described in this section. For internal validity we identify the following threats:

- In the parameter tuning step, we did not explore every combination of parameters. Also, not every possible parameter value was tried. Exploring every combination or testing every possible parameter value would result in a vast amount required runs, making the optimization infeasible. The found combination is valid, but it may not be optimal. More extensive parameter tuning may result in better performance and potentially better solutions from the algorithm.
- The relative improvement in terms of the cohesion metric does not hold a very specific value. The algorithm can "cheat the system" by moving a single class to a new module multiple times, increasing the metric value significantly without actually improving the codebase in terms of cohesion. The metric also will not converge until a solution of one class per module has been constructed. For these reasons, it is not a correctly representative value for the cohesion of the codebase. Using a better cohesion metric might result in better solutions in terms of cohesion.

This does not mean that the solutions do not improve the codebase in terms of cohesion, however. Suggestions that developers concluded to be good also improved on cohesion in their opinion.

- The suggested groups of class move refactorings that were shown to developers in the interview process are not entirely representative of the solutions they originate from, due to the filtering and grouping process beforehand. Due to this, the conclusion that all suggested groups indicate flaws in the code might not hold for all class move refactorings in the solutions. The conclusion that the solutions cannot be taken at face value is still sound, because a majority of the reviewed change groups were flawed, which on their own represent a significant part of the improvement the solutions imply.
- Interviews are prone to bias. Two major inherent biases can be identified in this interview. The first is that developers might be prone to judge suggestions positively after they have been told its positive effects in terms of the metrics. The other is that developers who have been involved in making decisions on in which module the classes which are suggested to be moved should be, or involved in its creation, might be prone to judge the suggestions negatively, as they are already convinced that the classes are where they should be.

In terms of construct validity, we identify the following threats:

- The results imply that different developers judge differently. All but one suggested changes have been reviewed by one developer. For proper validation, it would be preferred to have multiple developers review each suggested change. This would lead to a more rigorous validation.
- We measured the effort of code changes by the number of classes moved. This may not properly represent the effort required for the changes, as some classes might be significantly harder to move than others. To more accurately measure the effectiveness of solutions, the MoJoFM metric [67] could have been used for example. This metric shows the efficiency of a solution by evaluating the achieved improvement relative to the effort required to achieve it. This way of evaluation has been used in remodularization research [6].

In regards to external validity, the approach in this thesis was applied to one codebase which is used and developed by one company. More research is required to determine whether the results are generalizable to systems using different programming languages, developed by different people, fulfilling different purposes, or other systems in general.

Chapter 9

Conclusions and Future Work

This chapter first restates the problem tackled. Afterward, we draw conclusions from the results. Finally, potential future work will be discussed.

The problem tackled in this thesis is the remodularization problem. To optimize for this problem, a modularization into modules has to be found that improves on the given modularization. This modularization can be done on several levels. In this thesis, we looked at a grouping of classes into modules. A solution consists of a set of class moves that when implemented achieve improved modularization.

Specifically, the contribution of this thesis is scaling the approach up to a larger scale codebase and validating the results by interviewing developers that have worked on the code that is suggested to be changed. Both of these factors are often missing in related research [52].

We approached this problem by modeling the codebase in a similar way to existing work, namely graph-based. We view the problem as an optimization problem and apply the NSGA-II algorithm to it. After testing combinations of optimization variables, we set the algorithm to optimize for 4 metrics. The first is cohesion, which is modeled by the amount of intra-module class dependencies divided by the maximum possible per module. The second is coupling, which is modeled by the amount of module to module dependencies. The third is a novel metric representing weighted transitive coupling, called Estimated Build Cost of module Cache Breaks. The final is the number of class move refactorings required to achieve the solution. The algorithm is run for 72 hours once to show how long it takes to converge. It is also run for 24 hours twice, to prove the improvement potential in theory. The same results are split into small groups and reviewed by developers that have worked on the code that is suggested to be changed. We use the results to answer the following research questions.

RQ1: What is the improvement of the code structure in the solutions that the proposed approach finds in terms of the metrics used as optimization variables? From observation 1 we can conclude that the results of this approach show significant improvement to the code in terms of the metrics used, namely a 205.173% improvement on the IntraMD metric, 3.656% improvement on the InterMD metric, and a 1.667% improvement on the EBCCB metric depending on the implemented solution.

RQ2: How fast does the proposed algorithm converge? We can conclude from observations 5 and 6 that the algorithm can converge in terms of two metrics in an acceptable amount of time, namely around 20000 generations or 72 hours on the used hardware. In terms of the other metric, the algorithm will not converge within an acceptable amount of time.

RQ3: How do developers perceive the remodularization suggestions of the approach? The value of the solutions cannot be taken at face value however, we can conclude from observation 9, as it is shown that more than half of the subgroups of suggested class refactorings are not improvements according to the interviewed developers. This does not mean that the suggestions hold no value, as from observation 8 it can be concluded that the suggested refactorings reviewed all indicated flaws in the code according to the developers.

The results and conclusions imply some potential future work. In the modularization field of research, lots of work exists that prove their approach solely on a metric point of view. Due to the results in this thesis showing that the metrics do not always translate to reality well, we propose for validation efforts being done to see whether this research achieves different results. The validation can be performed by applying the approaches to a codebase with full-time developers and then interviewing the developers to review the suggested refactorings similarly as was done for this thesis. An improvement in the interview would be to have suggestions be validated by multiple developers, as observation 11 has shown that different developers can have completely contradictory judgments.

From the results, it's clear that the reviewed suggestions show flaws in the codebase, but some are arguably unfitting. A potential way to find better refactor suggestions, or offer a developer more options, is to introduce a form of local search into the process. This could be in a post-processing step or integrated into the MOEA. More work is needed to see if using such an approach can result in a higher ratio of suggestions that are good in practice.

The number of change group suggestions acquired in the filtering process imply that the generated solutions are not very diverse. This might be caused by improvements being propagated through the front where possible. Other approaches may achieve a more diverse set of code changes. Some research

has indicated that in bigger problems, MOEAs that take front diversity into account, for example, NSGA-III, which can outperform NSGA-II when using more than two optimization variables [18]. NSGA-III has been used successfully for modularization in the past when using more optimization variables [51]. It may be more successful in producing a set of solutions that varies more in terms of code changes, or in achieving higher performance.

Another possible way of improving the solutions in practice is taking a developer's perspective into the algorithm. This way, factors of code that can be more easily identified by developers than metrics are taken into account. This can be achieved by making the algorithm interactive or preference-based. Using interactive evolutionary algorithms or preference-based has shown to be successful in the past for modularization and refactor recommendation research [6, 29, 2, 48]. These approaches may result in better results in practice, due to the developer's influence on the solutions produced. This should be verified by for example using a similar way of interviewing as performed for this thesis.

Assuming that the previous proposed research cannot acquire solutions with 100% good code changes in practice, more work is needed to determine how to divide the solutions into smaller groups that can be reviewed by a developer. If this is done properly, less value of the solution will be lost in that process, and reviewing these solutions becomes easier. Given this work, existing modularization research can be measured to see how well the supposed solution quality translates to a business environment or other real-world scenarios.

We would also like to propose research into modularization quality metrics that take transitive coupling into account. Most metrics work on a local module measurement, and none examined seem to take into account the depth of module dependencies or weighed transitive coupling. The EBCCB metric proposed in this thesis has shown to be able to break transitive dependencies successfully. This metric has not yet been validated thoroughly, like some others [32]. We acknowledge that this metric may not be optimal and propose research that either improves and validates this metric or introduces a better one.

Bibliography

- [1] H. Abdeen, H. Sahraoui, O. Shata, N. Anquetil, and S. Ducasse. Towards automatically improving package structure while respecting original design decisions. In *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, October 2013. doi: 10.1109/wcre.2013.6671296. URL <https://doi.org/10.1109/wcre.2013.6671296>.
- [2] V. Alizadeh, M. Kessentini, W. Mkaouer, M. Ocinneide, A. Ouni, and Y. Cai. An interactive and dynamic search-based approach to software refactoring recommendations. *IEEE Transactions on Software Engineering*, pages 1–1, 2018. ISSN 2326-3881. doi: 10.1109/TSE.2018.2872711. URL <https://doi.org/10.1109/TSE.2018.2872711>.
- [3] E. B. Allen, T. M. Khoshgoftaar, and Y. Chen. Measuring coupling and cohesion of software modules: an information-theory approach. In *Proceedings Seventh International Software Metrics Symposium*. IEEE Comput. Soc, 2001. doi: 10.1109/metric.2001.915521. URL <https://doi.org/10.1109/metric.2001.915521>.
- [4] R. Baggen, J. P. Correia, K. Schill, and J. Visser. Standardized code quality benchmarking for improving software maintainability. *Software Quality Journal*, 20(2):287–307, May 2011. doi: 10.1007/s11219-011-9144-9. URL <https://doi.org/10.1007/s11219-011-9144-9>.
- [5] J. Bansiya and C. G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28(1):4–17, 2002. doi: 10.1109/32.979986. URL <https://doi.org/10.1109/32.979986>.
- [6] 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 *Search Based Software Engineering*, pages 75–89.

- Springer Berlin Heidelberg, 2012. doi: 10.1007/978-3-642-33119-0_7. URL https://doi.org/10.1007/978-3-642-33119-0_7.
- [7] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto. Using structural and semantic measures to improve software modularization. *Empirical Software Engineering*, 18(5):901–932, September 2012. doi: 10.1007/s10664-012-9226-8. URL <https://doi.org/10.1007/s10664-012-9226-8>.
- [8] G. Bavota, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia. An empirical study on the developers perception of software coupling. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, May 2013. doi: 10.1109/icse.2013.6606615. URL <https://doi.org/10.1109/icse.2013.6606615>.
- [9] G. Booch, R. A. Maksimchuk, M. W. Engle, B. J. Young, J. Connallen, and K. A. Houston. Object-oriented analysis and design with applications, third edition. *ACM SIGSOFT Software Engineering Notes*, 33(5):29–29, August 2008. doi: 10.1145/1402521.1413138. URL <https://doi.org/10.1145/1402521.1413138>.
- [10] L. Briand, P. Devanbu, and W. Melo. An investigation into coupling measures for c++. In *Proceedings of the 19th international conference on Software engineering - ICSE '97*. ACM Press, May 1997. doi: 10.1145/253228.253367. URL <https://doi.org/10.1145/253228.253367>.
- [11] L. Briand, J. Daly, and J. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Trans. Software Eng.*, 25:91–121, 1999. doi: 10.1109/32.748920. URL <https://doi.org/10.1109/32.748920>.
- [12] L. C. Briand, J. W. Daly, and J. Wüst. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering*, 3(1):65–117, 1998. doi: 10.1023/a:1009783721306. URL <https://doi.org/10.1023/a:1009783721306>.
- [13] L. C. Briand, J. Wüst, J. W. Daly, and D. V. Porter. Exploring the relationships between design measures and software quality in object-oriented systems. *Journal of Systems and Software*, 51(3):245–273, May 2000. doi: 10.1016/s0164-1212(99)00102-8. URL [https://doi.org/10.1016/s0164-1212\(99\)00102-8](https://doi.org/10.1016/s0164-1212(99)00102-8).
- [14] I. Candela, G. Bavota, B. Russo, and R. Oliveto. Using cohesion and coupling for software remodularization: Is it enough? *ACM Transactions on*

-
- Software Engineering and Methodology*, 25(3):1–28, August 2016. doi: 10.1145/2928268. URL <https://doi.org/10.1145/2928268>.
- [15] M. Cataldo, A. Mockus, J. A. Roberts, and J. D. Herbsleb. Software dependencies, work dependencies, and their impact on failures. *IEEE Transactions on Software Engineering*, 35(6):864–878, November 2009. doi: 10.1109/tse.2009.42. URL <https://doi.org/10.1109/tse.2009.42>.
- [16] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994. doi: 10.1109/32.295895. URL <https://doi.org/10.1109/32.295895>.
- [17] C. Y. Chong, S. P. Lee, and T. C. Ling. Efficient software clustering technique using an adaptive and preventive dendrogram cutting approach. *Information and Software Technology*, 55(11):1994–2012, November 2013. doi: 10.1016/j.infsof.2013.07.002. URL <https://doi.org/10.1016/j.infsof.2013.07.002>.
- [18] G. C. Ciro, F. Dugardin, F. Yalaoui, and R. Kelly. A NSGA-II and NSGA-III comparison for solving an open shop scheduling problem with resource constraints. *IFAC-PapersOnLine*, 49(12):1272–1277, 2016. doi: 10.1016/j.ifacol.2016.07.690. URL <https://doi.org/10.1016/j.ifacol.2016.07.690>.
- [19] S. Counsell, S. Swift, A. Tucker, and E. Mendes. Object-oriented cohesion subjectivity amongst experienced and novice developers. *ACM SIGSOFT Software Engineering Notes*, 31(5):1–10, September 2006. doi: 10.1145/1163514.1163530. URL <https://doi.org/10.1145/1163514.1163530>.
- [20] W. Cunningham. The WyCash portfolio management system. In *Addendum to the proceedings on Object-oriented programming systems, languages, and applications (Addendum) - OOPSLA '92*. ACM Press, 1992. doi: 10.1145/157709.157715. URL <https://doi.org/10.1145/157709.157715>.
- [21] K. Deb and H. Jain. An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part i: Solving problems with box constraints. *IEEE Transactions on Evolutionary Computation*, 18(4):577–601, August 2014. doi: 10.1109/tevc.2013.2281535. URL <https://doi.org/10.1109/tevc.2013.2281535>.
- [22] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multi-objective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, April 2002. doi: 10.1109/4235.996017. URL <https://doi.org/10.1109/4235.996017>.

- [23] H. Dhama. Quantitative models of cohesion and coupling in software. *Journal of Systems and Software*, 29(1):65–74, April 1995. doi: 10.1016/0164-1212(94)00128-a. URL [https://doi.org/10.1016/0164-1212\(94\)00128-a](https://doi.org/10.1016/0164-1212(94)00128-a).
- [24] S. A. Ebad and M. Ahmed. Software packaging approaches —a comparison framework. In *Software Architecture*, pages 438–446. Springer Berlin Heidelberg, 2011. doi: 10.1007/978-3-642-23798-0_44. URL https://doi.org/10.1007/978-3-642-23798-0_44.
- [25] M. O. Elish and M. A. Al-Khiaty. A suite of metrics for quantifying historical changes to predict future change-prone classes in object-oriented software. *Journal of Software: Evolution and Process*, 25(5):407–437, January 2012. doi: 10.1002/smr.1549. URL <https://doi.org/10.1002/smr.1549>.
- [26] L. H. Etzkorn, S. E. Gholston, J. L. Fortune, C. E. Stein, D. Utley, P. A. Farrington, and G. W. Cox. A comparison of cohesion metrics for object-oriented systems. *Information and Software Technology*, 46(10):677–687, August 2004. doi: 10.1016/j.infsof.2003.12.002. URL <https://doi.org/10.1016/j.infsof.2003.12.002>.
- [27] A. Fadhel, M. Kessentini, P. Langer, and M. Wimmer. Search-based detection of high-level model changes. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, September 2012. doi: 10.1109/icsm.2012.6405274. URL <https://doi.org/10.1109/icsm.2012.6405274>.
- [28] G. Fraser and A. Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, Feb 2013. ISSN 2326-3881. doi: 10.1109/TSE.2012.14. URL <https://doi.org/10.1109/TSE.2012.14>.
- [29] A. Ghannem, G. El Boussaidi, and M. Kessentini. Model refactoring using interactive genetic algorithm. In *Search Based Software Engineering*, pages 96–110. Springer Berlin Heidelberg, 2013. doi: 10.1007/978-3-642-39742-4_9. URL https://doi.org/10.1007/978-3-642-39742-4_9.
- [30] A. Ghannem, M. Kessentini, M. S. Hamdi, and G. El Boussaidi. Model refactoring by example: A multi-objective search based software engineering approach. *Journal of Software: Evolution and Process*, 30(4):e1916, November 2017. doi: 10.1002/smr.1916. URL <https://doi.org/10.1002/smr.1916>.
- [31] M. Harman, R. Hierons, and M. Proctor. A new representation and crossover operator for search-based optimization of software modulariza-

- tion. GECCO'02, page 1351–1358, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc. ISBN 1558608788.
- [32] M. Hitz and B. Montazeri. Chidamber and kemerers metrics suite: a measurement theory perspective. *IEEE Transactions on Software Engineering*, 22(4):267–271, April 1996. doi: 10.1109/32.491650. URL <https://doi.org/10.1109/32.491650>.
- [33] C. Jermaine. Computing program modularizations using the k-cut method. In *Sixth Working Conference on Reverse Engineering (Cat. No.PR00303)*. IEEE Comput. Soc, 1999. doi: 10.1109/wcre.1999.806963. URL <https://doi.org/10.1109/wcre.1999.806963>.
- [34] H. Kabaili, R. K. Keller, and F. Lustman. Cohesion as changeability indicator in object-oriented systems. In *Proceedings Fifth European Conference on Software Maintenance and Reengineering*. IEEE Comput. Soc. doi: 10.1109/csmr.2001.914966. URL <https://doi.org/10.1109/csmr.2001.914966>.
- [35] M. Kessentini, W. Kessentini, H. Sahraoui, M. Boukadoum, and A. Ouni. Design defects detection and correction by example. In *2011 IEEE 19th International Conference on Program Comprehension*. IEEE, June 2011. doi: 10.1109/icpc.2011.22. URL <https://doi.org/10.1109/icpc.2011.22>.
- [36] C. K. Kwong, L. F. Mu, J. F. Tang, and X. G. Luo. Optimization of software components selection for component-based software system development. *Computers & Industrial Engineering*, 58(4):618–624, May 2010. doi: 10.1016/j.cie.2010.01.003. URL <https://doi.org/10.1016/j.cie.2010.01.003>.
- [37] J. Kook Lee, S. Jae Jung, S. Dong Kim, W. Hyun Jang, and D. Han Ham. Component identification method with coupling and cohesion. In *Proceedings Eighth Asia-Pacific Software Engineering Conference*. IEEE Comput. Soc. doi: 10.1109/apsec.2001.991462. URL <https://doi.org/10.1109/apsec.2001.991462>.
- [38] R. Mahouachi. Search-based cost-effective software remodularization. *Journal of Computer Science and Technology*, 33(6):1320–1336, November 2018. doi: 10.1007/s11390-018-1892-6. URL <https://doi.org/10.1007/s11390-018-1892-6>.
- [39] R. Mahouachi, M. Kessentini, and M. Ó Cinnéide. Search-based refactoring detection using software metrics variation. In *Search Based Software Engineering*, pages 126–140. Springer Berlin Heidelberg, 2013.

- doi: 10.1007/978-3-642-39742-4_11. URL https://doi.org/10.1007/978-3-642-39742-4_11.
- [40] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No.98TB100242)*. IEEE Comput. Soc. doi: 10.1109/wpc.1998.693283. URL <https://doi.org/10.1109/wpc.1998.693283>.
- [41] A. Marcus, D. Poshyvanyk, and R. Ferenc. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Transactions on Software Engineering*, 34(2):287–300, March 2008. doi: 10.1109/tse.2007.70768. URL <https://doi.org/10.1109/tse.2007.70768>.
- [42] R. C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, USA, 2003. ISBN 0135974445.
- [43] A. Marx, F. Beck, and S. Diehl. Computer-aided extraction of software components. In *2010 17th Working Conference on Reverse Engineering*. IEEE, October 2010. doi: 10.1109/wcre.2010.28. URL <https://doi.org/10.1109/wcre.2010.28>.
- [44] T. M. Meyers and D. Binkley. An empirical study of slice-based cohesion and coupling metrics. *ACM Transactions on Software Engineering and Methodology*, 17(1):1–27, December 2007. doi: 10.1145/1314493.1314495. URL <https://doi.org/10.1145/1314493.1314495>.
- [45] B. S. Mitchell. A heuristic search approach to solving the software clustering problem. 2002. AAI3039424.
- [46] B. S. Mitchell and S. Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering*, 32(3):193–208, March 2006. doi: 10.1109/tse.2006.31. URL <https://doi.org/10.1109/tse.2006.31>.
- [47] B. S. Mitchell and S. Mancoridis. On the evaluation of the bunch search-based software modularization algorithm. *Soft Computing*, 12(1):77–93, June 2007. doi: 10.1007/s00500-007-0218-3. URL <https://doi.org/10.1007/s00500-007-0218-3>.
- [48] M. W. Mkaouer, M. Kessentini, S. Bechikh, and D. R. Tauritz. Preference-based multi-objective software modelling. In *2013 1st International Workshop on Combining Modelling and Search-Based Software Engineering (CMSBSE)*. IEEE, May 2013. doi: 10.1109/cmsbse.2013.6605712. URL <https://doi.org/10.1109/cmsbse.2013.6605712>.

-
- [49] M. W. Mkaouer, M. Kessentini, S. Bechikh, and M. Ó Cinnéide. A robust multi-objective approach for software refactoring under uncertainty. In *Search-Based Software Engineering*, pages 168–183. Springer International Publishing, 2014. doi: 10.1007/978-3-319-09940-8_12. URL https://doi.org/10.1007/978-3-319-09940-8_12.
- [50] M. W. Mkaouer, M. Kessentini, M. Ó Cinnéide, S. Hayashi, and K. Deb. A robust multi-objective approach to balance severity and importance of refactoring opportunities. *Empirical Software Engineering*, 22(2):894–927, March 2016. doi: 10.1007/s10664-016-9426-8. URL <https://doi.org/10.1007/s10664-016-9426-8>.
- [51] 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*, 24(3):1–45, May 2015. doi: 10.1145/2729974. URL <https://doi.org/10.1145/2729974>.
- [52] M. Mohan and D. Greer. A survey of search-based refactoring for software maintenance. *Journal of Software Engineering Research and Development*, 6(1), February 2018. doi: 10.1186/s40411-018-0046-4. URL <https://doi.org/10.1186/s40411-018-0046-4>.
- [53] A. Ouni, M. Kessentini, H. Sahraoui, and M. S. Hamdi. Search-based refactoring: Towards semantics preservation. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, September 2012. doi: 10.1109/icsm.2012.6405292. URL <https://doi.org/10.1109/icsm.2012.6405292>.
- [54] J. Pantiuchina, M. Lanza, and G. Bavota. Improving code: The (mis) perception of quality metrics. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, September 2018. doi: 10.1109/icsme.2018.00017. URL <https://doi.org/10.1109/icsme.2018.00017>.
- [55] M. Di Penta. Evolution doctor: A framework to control software system evolution. In *Ninth European Conference on Software Maintenance and Reengineering*. IEEE. doi: 10.1109/csmr.2005.29. URL <https://doi.org/10.1109/csmr.2005.29>.
- [56] K. Praditwong, M. Harman, and X. Yao. Software module clustering as a multi-objective search problem. *IEEE Transactions on Software Engineering*, 37(2):264–282, March 2011. doi: 10.1109/tse.2010.26. URL <https://doi.org/10.1109/tse.2010.26>.

- [57] M. Revelle, M. Gethers, and D. Poshyvanyk. Using structural and textual information to capture feature coupling in object-oriented software. *Empirical Software Engineering*, 16(6):773–811, March 2011. doi: 10.1007/s10664-011-9159-7. URL <https://doi.org/10.1007/s10664-011-9159-7>.
- [58] D. Romano and M. Pinzger. Using source code metrics to predict change-prone java interfaces. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, September 2011. doi: 10.1109/ic-sm.2011.6080797. URL <https://doi.org/10.1109/ic-sm.2011.6080797>.
- [59] S. Sarkar, G. Rama, and A. Kak. API-based and information-theoretic metrics for measuring the quality of software modularization. *IEEE Transactions on Software Engineering*, 33(1):14–32, January 2007. doi: 10.1109/tse.2007.256942. URL <https://doi.org/10.1109/tse.2007.256942>.
- [60] R. W. Schwanke and S. J. Hanson. Using neural networks to modularize software. *Machine Learning*, 15(2):137–168, May 1994. doi: 10.1007/bf00993275. URL <https://doi.org/10.1007/bf00993275>.
- [61] D. F. Da Silva, L. F. Okada, T. E. Colanzi, and W. K. G. Assunção. Enhancing search-based product line design with crossover operators. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*. ACM, June 2020. doi: 10.1145/3377930.3390215. URL <https://doi.org/10.1145/3377930.3390215>.
- [62] I. Stamelos, L. Angelis, A. Oikonomou, and G. L. Bleris. Code quality analysis in open source software development. *Information Systems Journal*, 12(1):43–60, January 2002. doi: 10.1046/j.1365-2575.2002.00117.x. URL <https://doi.org/10.1046/j.1365-2575.2002.00117.x>.
- [63] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974. doi: 10.1147/sj.132.0115. URL <https://doi.org/10.1147/sj.132.0115>.
- [64] M. Stoer and F. Wagner. A simple min-cut algorithm. *Journal of the ACM (JACM)*, 44(4):585–591, July 1997. doi: 10.1145/263867.263872. URL <https://doi.org/10.1145/263867.263872>.
- [65] M. M. T. Thwin and T. S. Quah. Application of neural networks for software quality prediction using object-oriented metrics. *Journal of Systems and Software*, 76(2):147–156, May 2005. doi: 10.1016/j.jss.2004.05.001. URL <https://doi.org/10.1016/j.jss.2004.05.001>.

- [66] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. Lucia, and D. Poshyvanyk. When and why your code starts to smell bad (and whether the smells go away). *IEEE Transactions on Software Engineering*, PP:1–1, 01 2017. doi: 10.1109/TSE.2017.2653105. URL <https://doi.org/10.1109/TSE.2017.2653105>.
- [67] Z. Wen and V. Tzerpos. An effectiveness measure for software clustering algorithms. In *Proceedings. 12th IEEE International Workshop on Program Comprehension, 2004*. IEEE. doi: 10.1109/wpc.2004.1311061. URL <https://doi.org/10.1109/wpc.2004.1311061>.
- [68] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9):574–586, September 2004. doi: 10.1109/tse.2004.52. URL <https://doi.org/10.1109/tse.2004.52>.
- [69] E. Yourdon and L.L. Constantine. *Structured design: fundamentals of a discipline of computer program and systems design*. Prentice-Hall, Inc., 1979. ISBN 0138544719.
- [70] C. Zhang and H. A. Jacobsen. Prism is research in aspect mining. pages 20–21, 01 2004. doi: 10.1145/1028664.1028676. URL <https://doi.org/10.1145/1028664.1028676>.

Appendix A

Glossary

A.1 Terminology

In this section we give an overview of frequently used terms and abbreviations.

- **Codebase:** The entire source code of a certain piece of software.
- **Dependency structure:** The dependency structure is the total picture that arises from the dependencies from class to class. The way these classes are divided into modules results in dependencies from module to module. Both of these dependency levels can be represented with directed graphs. (see section 2.2.1)
- **(software/code) Architecture:** The structure of a piece of software or codebase. In this thesis, the term is often used to refer to the dependency structure in particular.
- **Modularization:** The problem of dividing classes into modules in such a way that a good module structure is achieved. Re-modularization is the same problem, but from an already existing modularization, preferably with not too many changes.
- **Cohesion:** An abstract concept representing how well a piece of code fits together. (see section 2.1.2)
- **Coupling:** An abstract concept representing how interconnected a piece of code is. (see section 2.1.1)
- **IntraMD:** Intra Module Dependencies. The metric used in to represent cohesion. It is measured by the number of class dependencies in a module divided by the number of possible ones, summed for each module.
- **InterMD:** Inter Module Dependencies. The metric used in to represent coupling. It is measured by the number of module dependencies.

- **CCP**: The Common Closure Principle [42]. This is measured by the number of pairs of classes changed in the same commit over a set period of time.
- **CRP**: The Common Reuse Principle [42]. This is measured by the number of pairs of classes which are both in the same module and used by the same class, per class.
- **(EA) Evolutionary Algorithm**: An optimization algorithm based on the concept of evolution. It contains some sort of population made of solutions, which procreate and mutate to form new solutions. The fittest solutions, which are the ones that optimize the problem the best, are allowed to procreate and mutate with the goal of finding fitter solutions.
- **(MOEA) Multi-Objective Evolutionary Algorithm**: A type of evolutionary algorithm that optimizes for more than one value at the same time. In this type of algorithm, there is more than one "optimal" solution, often described as the pareto-front.
- **Enterprise software**: Software made specifically to satisfy the needs of an organization. Often used in a way to emphasize the size/scale of the software.
- **Build cost**: The cost of building a module, expressed in time.
- **Dead code**: Code that has not been used and changed in a while, and is not part of future plans.
- **Transitive coupling / transitive dependencies**: If module M1 depends on module M2, and module M2 depends on module M3, M1 is said to be transitively coupled to / transitively dependent on module M3.
- **Caching**: In this thesis caching refers to module build caching specifically. Module build caching is often used in enterprise-level software, especially in monolithic repositories. When a module is built, the built state is saved and shared so that any developer that needs the same state of the module does not have to build it. This process saves a lot of time for developers, whenever they want to run their code.
- **Transitive cache breaks**: When some part of the code in a module is changed, its cache is broken. Due to this module being rebuilt, all the modules using this module also need to be rebuilt. This causes a cascading effect through the module dependency structure of cache breaks.
- **(EBCCB) Estimated Build Cost of module Cache Breaks**: The new metric introduced in this thesis representing weighed transitive coupling. For further explanation see section 4.5.

Appendix B

Interview

B.1 Interview Questions

- Interviewee is given an explanation of the interview, the goal of the research, and that they are going to review a part of a solution generated by the algorithm.
- How many years of experience in software development do you have?
- How many years have you worked at Adyen?
- How well would you say that you grasp the module to module dependency structure of the entire codebase, on a scale from 0 to 10?
- How well would you say that you grasp the module to module dependency structure around *the module the suggestion's classes are from*, on a scale from 0 to 10?
- The interviewee is shown the suggestion and it is explained why this is an improvement in terms of the metrics.
- Would you consider this suggestion to be a good change, meaning it has an overall positive impact on the codebase?
- (If the interviewee does not give clear reasoning) Could you explain your reasoning for the previous answer?
- (If the interviewee deems the suggestion to be good) How important would you rate this change in terms of severity and priority, on scales of 0 to 10?
- Has this change given you an idea on how to improve the code structure in this area of the code?

B.2 Interview answers

D1 **How well would you say that you grasp the module dependency structure of the codebase?**

7/10

How well would you say that you grasp the module dependency structure of the module C1 is in?

7/10

Would you consider this C1 to be a good change, meaning it has an overall positive impact on the codebase?

C1: consists of moving one class to a new module. D1 deemed it to be a bad move. Logically the class fits in the module it is in, and moving just this class to a new module would make it too granular.

Has this suggestion given you an idea on how to improve the code structure in this area of the code?

This suggestion did reveal a very strange and possibly unnecessary dependency from this class to another module, causing multiple transitive dependencies. This is the cause of the EBCCB value impact of the suggestion.

How well would you say that you grasp the module dependency structure of the module C2 is in?

6/10

Would you consider this C2 to be a good change, meaning it has an overall positive impact on the codebase?

C2: consists of moving 2 classes to separate new modules, and 2 to existing modules. D1 verified that this move is good, however, there are better modules to move these classes to. Making modules for one class is again too granular, and there exist fitting modules for these 2 classes. Also, one of the classes that is moved to an existing module should be moved to a different one to follow design principles.

How important would you rate this change in terms of severity (impact relative to effort), and priority (compared to the day to day development tasks), on a scale from 0 to 10?

Severity: 4/10

Priority: 7/10

D2 **How well would you say that you grasp the module dependency structure of the codebase?**

6/10

How well would you say that you grasp the module dependency structure of the module C3 is in?

6/10

Would you consider this C3 to be a good change, meaning it has an overall positive impact on the codebase?

C3: consists of moving one class to a different module. D2 deemed this to be a good move. The class is not used and is not using anything in its current module. However, there is a future plan for the module it is moved to that would make it not fit. Given this information, it would be better to construct a new module specifically for this class and related functionality.

How important would you rate this change in terms of severity (impact relative to effort), and priority (compared to the day to day development tasks), on a scale from 0 to 10?

Severity: 4/10

Priority: 4/10

D3 How well would you say that you grasp the module dependency structure of the codebase?

7/10

How well would you say that you grasp the module dependency structure of the module C4 is in?

7/10

Would you consider this C4 to be a good change, meaning it has an overall positive impact on the codebase?

C4: consists of moving 1 class to a new module, and 1 class to a different module. D3 deemed this suggestion to be bad and good at the same time. The suggestion is bad because the class that is moved to a new module is dead code causing unnecessary dependencies, so it would be better to delete it entirely. The move of the other class is good in some sense, as it has been the part of an ongoing discussion where a group of developers is advocating for moving multiple classes to the module this class is suggested to be moved to. However, another group of developers says it is where it should be. What has become clear by this interview is that the class does not adhere to the design pattern properly. Before a move would be made, the class should be fixed first.

How important would you rate this change in terms of severity (impact relative to effort), and priority (compared to the day to day development tasks), on a scale from 0 to 10?

B. Interview

Severity: 2/10

Priority: 2/10

D4 **How well would you say that you grasp the module dependency structure of the codebase?**

4/10

How well would you say that you grasp the module dependency structure of the module C5 is in?

5/10

Would you consider this C5 to be a good change, meaning it has an overall positive impact on the codebase?

C5: consists of moving 2 classes to a different module. D7 was also interviewed on this suggestion. D4 saw this as a bad change. One class is not necessarily in the wrong place. D4 is unsure about the other class, noting that it feels out of place. The module they are moved to is unfitting for both.

Has this suggestion given you an idea on how to improve the code structure in this area of the code?

No, no specific ideas.

D5 **How well would you say that you grasp the module dependency structure of the codebase?**

3/10

How well would you say that you grasp the module dependency structure of the module C6 is in?

5/10

Would you consider this C6 to be a good change, meaning it has an overall positive impact on the codebase?

C6: consists of one class being moved to a new module. D5 verified this to be a bad move. The class has been superseded by another piece of functionality D5 wrote, so it should be removed and usages transferred, instead of being moved to a new module.

Has this suggestion given you an idea on how to improve the code structure in this area of the code?

Remove the class and transfer the usages to the other functionality.

D6 **How well would you say that you grasp the module dependency structure of the codebase?**

5/10

How well would you say that you grasp the module dependency structure of the module C7 is in?

9/10

Would you consider this C7 to be a good change, meaning it has an overall positive impact on the codebase?

C7: consists of one class being moved to another module. D6 said this move makes sense in the current state of the code, however, there are structural plans for new functionality that will use the moved class, making the suggestion bad.

Has this suggestion given you an idea on how to improve the code structure in this area of the code?

It has shown the module to be a catch-all, so splitting it would be an improvement.

How well would you say that you grasp the module dependency structure of the module C8 is in?

4/10

Would you consider this C8 to be a good change, meaning it has an overall positive impact on the codebase?

C8: consists of one class being moved to another (very closely related) module. D6 states the class fits in both modules, so given the algorithm's metrics, the move is good.

How important would you rate this change in terms of severity (impact relative to effort), and priority (compared to the day to day development tasks), on a scale from 0 to 10?

Severity: 2/10

priority: 2/10

D7 **How well would you say that you grasp the module dependency structure of the codebase?**

5/10

How well would you say that you grasp the module dependency structure of the module C5 is in?

8/10

Would you consider this C5 to be a good change, meaning it has an overall positive impact on the codebase?

C5: consists of moving 2 classes to a different module. D4 was also interviewed on this suggestion. D7 states that the first class should be moved

B. Interview

according to the suggestion as it fits better in the suggested module. The second class doesn't fit its current module, partly due to it holds too much differing functionality. It should be fixed first, but in its current state, the move would be good.

How important would you rate this change in terms of severity (impact relative to effort), and priority (compared to the day to day development tasks), on a scale from 0 to 10?

Severity: 8/10

Priority: 3/10

D8 How well would you say that you grasp the module dependency structure of the codebase?

8/10

How well would you say that you grasp the module dependency structure of the module C9 is in?

8/10

Would you consider this C9 to be a good change, meaning it has an overall positive impact on the codebase?

C9: consists of moving 2 classes to different new modules. D8 determined this to be bad. One of the classes is dead code, so instead it should be removed. The other should be moved, but not to a new module, as it fits some existing modules already.

Has this suggestion given you an idea on how to improve the code structure in this area of the code?

Yes, remove the dead code and also move the other class to a fitting existing module

How well would you say that you grasp the module dependency structure of the module C10 is in?

9/10

Would you consider this C10 to be a good change, meaning it has an overall positive impact on the codebase?

C10: consists of moving 1 class to a different module. D8 stated that this move is bad, as the class fits its current module and does not fit the suggested module. However, looking at this suggestion has shown that something is off about the structure around and the usage of this class.

Has this suggestion given you an idea on how to improve the code structure in this area of the code?

Fix the usage of this class.

D9 **How well would you say that you grasp the module dependency structure of the codebase?**

4/10

How well would you say that you grasp the module dependency structure of the module C11 is in?

5/10

Would you consider this C11 to be a good change, meaning it has an overall positive impact on the codebase?

C11: consists of moving 1 class to a different module. D9 has verified this to be a good move, as it is only used in the suggested module and not in the one it is in.

How important would you rate this change in terms of severity (impact relative to effort), and priority (compared to the day to day development tasks), on a scale from 0 to 10?

Severity: 7/10

Priority: 4/10

D10 **How well would you say that you grasp the module dependency structure of the codebase?**

4/10

How well would you say that you grasp the module dependency structure of the module C12 is in?

0/10

Would you consider this C12 to be a good change, meaning it has an overall positive impact on the codebase?

C12: consists of moving 1 class to a different module. D10 has stated that this move is bad, due to the suggested module being in a refactoring process with the goal of splitting it up. This class was moved to its current module in the process, but it's not the best place for it. A better option would be to take this class, related functionality from the suggested module, and related functionality which is in a different module and combine it into a new module. Looking into this also revealed that code from the class' module seems quite stale, so it should be looked into.

Has this suggestion given you an idea on how to improve the code structure in this area of the code?

Create a new module around the class and its related functionality.

B. Interview

D11 **How well would you say that you grasp the module dependency structure of the codebase?**

3/10

How well would you say that you grasp the module dependency structure of the module C13 is in?

10/10

Would you consider this C13 to be a good change, meaning it has an overall positive impact on the codebase?

C13: consists of moving a package of 6 classes to a different module, and 1 other class to a new module. D11 confirmed the package move to be good. However, the other class should be removed instead of moved, because it is dead code.

How important would you rate this change in terms of severity (impact relative to effort), and priority (compared to the day to day development tasks), on a scale from 0 to 10?

Severity: 8/10

Priority: 6/10

Appendix C

Figures & Tables

C.1 LOC correlation with build costs

In the Figures, C.1 and C.2, the correlation between build tasks and LOC, and the formula achieved through regression can be seen. The function shown was achieved by feeding a function in the following form into the *optimize.curve_fit* function of python package *scipy*, and using the relative error instead of absolute:

$$f(x) = a * \log_{10}(x)^b + c \tag{C.1}$$

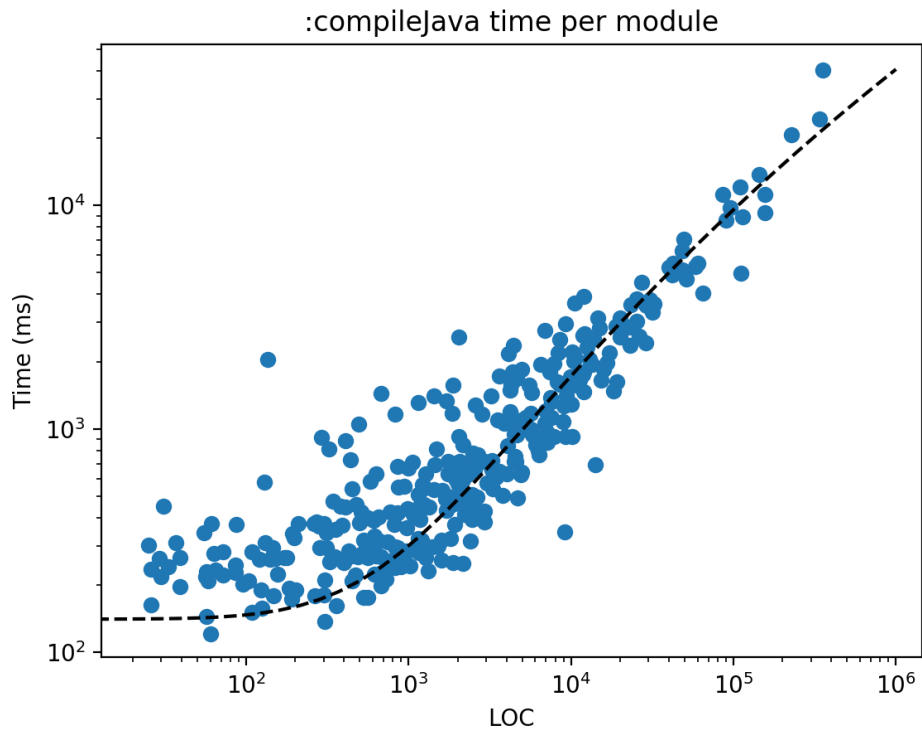


Figure C.1: The runtime of a compile Java task relative to the lines of code in the module

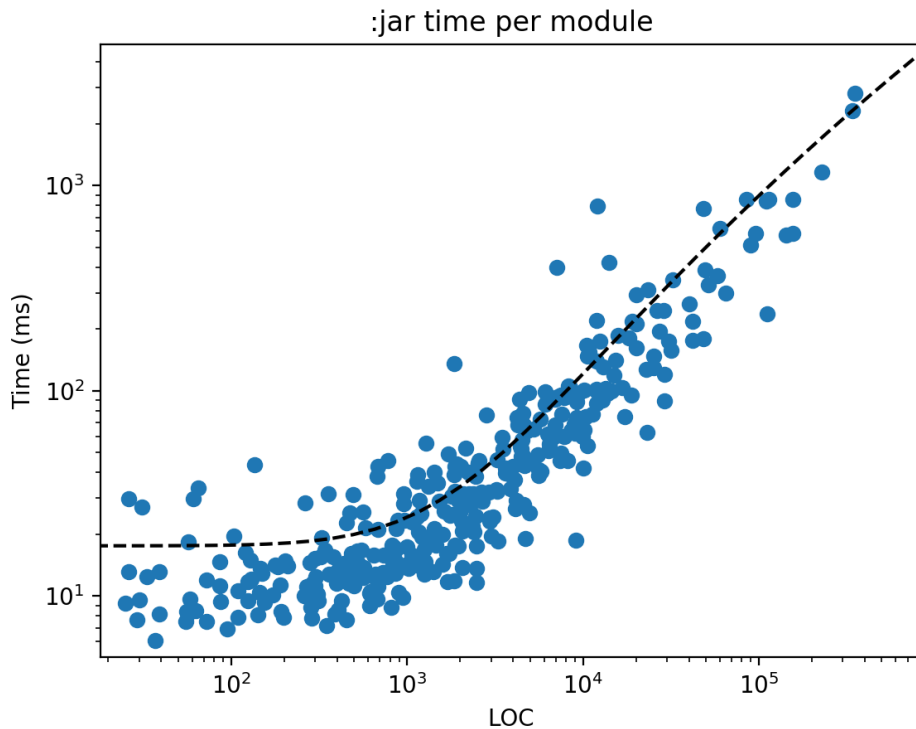


Figure C.2: The runtime of a jar task relative to the lines of code in the module

C.2 Optimization variable tweaking results

The results can be seen in Table C.1 & C.2. These tables contain the highest amount of generations that was achieved in the 5 runs per configuration, the average amount of generations, the best value of each optimization variable for solutions in all runs, and the average value of the best values per optimization variable per run. Note that all optimization variables use the existing architecture as the 0 point and negative values show improvements (the values of CCP, CRP, and Intra Module Coupling are flipped). The optimization variable configurations are as follows:

1. IntraMD, InterMD, EBCCB, # Changes

A combination including all aspects optimized for, with only one Cohesion metric.

2. IntraMD, InterMD

This combination has been used in modularization research before[31]. It is included as a baseline to show the value of EBCCB as an optimization variable, and other combinations tested.

3. InterMD, CRP, CCP

A combination that also optimizes for Coupling and Cohesion, with CCP and CRP used for Cohesion.

4. IntraMD, InterMD, EBCCB, CCP, CRP, # Changes

This combination uses all optimization variables to show its performance compared to the other combinations.

5. IntraMD, EBCCB

This combination optimizes for one of the types of Cohesion and the newly proposed metric, which is also used here as a type of coupling.

6. IntraMD, InterMD, # Changes

A combination to show the effect of optimizing for the number of classes moved compared to the often used cohesion/coupling combination.

7. IntraMD, CCP, CRP

The combination of only Cohesion optimization variables. It's included to show the effect of ignoring Coupling metrics.

8. InterMD, EBCCB

This combination of coupling and the newly introduced metric. It's included to show the effect of ignoring Cohesion metrics.

9. InterMD, CCP, CRP, EBCCB, # Changes

A similar combination to the first one, using the other Cohesion metrics.

10. IntraMD, EBCCB, # Changes

The same combination as #5, with the number of classes moved. This shows the effect of optimizing or ignoring the number of changes with these other optimization variables.

11. CCP, CRP, EBCCB, # Changes

A similar combination of optimization variables as the previous combination, with the other Cohesion optimization variables.

From the tested configurations, we concluded that configuration 1 is the best choice, as it shows a consistent improvement across the different variables tested, with the best improvement in some cases. It also performs well in terms of number of generations reached.

Table C.1: Optimization variable tweaking results (1/2)

Configuration	Highest # generations	Average # generations	IntraMD best	IntraMD average	CCP best	CCP average	CRP best	CRP average
1	3291	2923	-201.19203	-188.056636	-3	-0.6	-358.5	-144.3
2	1627	1332.4	-353.89275	-303.321648	500	2528.7	-213.5	4953.2
3	350	276.6	-9.44654	-7.567176	-5143.5	-4196.1	-45810.5	-37781.6
4	1629	1451.2	-155.68067	-136.209968	-359	-155.1	-2515	-1856.5
5	3060	1682.6	-381.06114	-307.17393	716	2327.9	-323.5	1379.7
6	3422	3242.6	-373.13179	-349.270522	0	0	-87	-71.2
7	246	227	-86.35844	-77.499758	-4318.5	-3716	-54467	-42200.2
8	6138	5700.8	-4.82292	-3.562402	14	51.4	-120.5	-107.5
9	994	479.2	-9.1541	-8.104232	-4866	-3199	-44588.5	-28359.7
10	3310	2981.2	-181.89897	-176.163372	0	0	0	0
11	401	332.4	-9.30071	-6.689142	-3590.5	-3225.6	-48990	-38548.4

Table C.2: Optimization variable tweaking results (2/2)

Configuration	Highest # generations	Average # generations	InterMD best	InterMD average	EBCCB best	EBCCB average
1	3291	2923	-10	-9.4	-8868.766327	-7249.249941
2	1627	1332.4	-10	-7.2	-89.75918085	15462.80009
3	350	276.6	-8	-5.6	37484.10431	345335.6063
4	1629	1451.2	-10	-9	-8868.766327	-5998.596233
5	3060	1682.6	74	265.2	-8696.175143	-6858.179237
6	3422	3242.6	-9	-8.4	-1855.888897	-1662.609525
7	246	227	93	132.8	999999	999999
8	6138	5700.8	-10	-9.8	-8868.766327	-8497.082381
9	994	479.2	-10	-3.6	-8868.766327	-1846.221569
10	3310	2981.2	0	0	-8146.341455	-5516.492598
11	401	332.4	-7	-1.4	-2183.037166	-767.415869

C.3 Parameter tuning results

This section holds the results of the parameter tuning runs. The results can be seen in the tables in this section. One table exists per tweaked parameter. The tables show the highest amount of generations that was achieved in the 5 runs per parameter, the average amount of generations, the best value of each used optimization variable for all solutions in all runs, and the average value of the best values per used optimization variable per run.

The default settings for all parameters are as follows:

- Population size = 500
- Mutation chance = 0.5
- Both mutation operators are used with equal probability
- Constraint breaks are ignored
- Duplicates are deleted
- The crossover operator that conserves building blocks is used
- An elite archive is not used

The options the parameters are tweaked for are as follows, with the optimal options in bold:

- Population size = {50, 100, 200, **500**, 1000, 2000}
- Mutation chance = {0.05, 0.1, 0.2, **0.5**}
- The optimal ratio of mutation operator usage = {0.25, **0.50**, 0.75}
- Whether solutions that break constraints are punished, repaired or whether **constraint breaks are ignored**
- How duplicates should be handled; by ignoring them, **deleting them** or mutating them
- Which crossover operator is used; the single-point operator or **the one that tries to preserve building blocks**.
- Whether an elite archive is used.

Note that for both population size and elite archive there was no clear optimal choice, so we decided to go for parameter values that have a balanced trade-off in performance and solution diversity. This resulted in a population size of 500, and using an elite archive of the size of the population, as it doubled the number of unique solutions while barely impacting performance.

Table C.3: Mutation choice parameter tuning results

Mutation Choice	Highest # generations	Average # generations	IntraMD best	IntraMD average	InterMD best	InterMD average	EBCCB best	EBCCB average
0.25	3047	2735.6	-204.35977	-196.329108	-9	-9	-8868.766327	-7617.684572
0.5	3379	3094.6	-241.85756	-211.027916	-10	-9.2	-8868.766327	-8796.452694
0.75	3439	3315.2	-217.37067	-179.69686	-10	-9.6	-8868.766327	-7504.235915

Table C.4: Mutation chance parameter tuning results

Mutation Chance	Highest # generations	Average # generations	IntraMD best	IntraMD average	InterMD best	InterMD average	EBCCB best	EBCCB average
0.05	3485	3236.8	-230.86222	-203.496192	-10	-9.2	-8507.19816	-8505.507435
0.1	3349	3268.8	-238.8507	-204.025204	-9	-9	-8868.766327	-8650.698277
0.2	3514	3371	-230.85718	-205.87951	-10	-9.6	-8507.19816	-7176.936307
0.5	3379	3094.6	-241.85756	-211.027916	-10	-9.2	-8868.766327	-8796.452694

Table C.5: Dealing with solutions that break constraints parameter tuning results

Constraint Breaking Solutions	Highest # generations	Average # generations	IntraMD best	IntraMD average	InterMD best	InterMD average	EBCCB best	EBCCB average
ignore	3379	3094.6	-241.85756	-211.027916	-10	-9.2	-8868.766327	-8796.452694
punish	2701	2472.6	-191.06344	-179.100518	-10	-7.8	-8696.175143	-7312.543237
repair	2737	2604	-168.18622	-159.756326	-9	-9	-8868.766327	-8723.575485

Table C.6: Dealing with duplicate solutions parameter tuning results

Dealing with Duplicates	Highest # generations	Average # generations	IntraMD best	IntraMD average	InterMD best	InterMD average	EBCCB best	EBCCB average
delete	3379	3094.6	-241.85756	-211.027916	-10	-9.2	-8868.766327	-8796.452694
ignore	3223	3011.4	-236.8605	-212.099254	-10	-9.2	-8868.766327	-8651.261852
mutate	9438	9141.4	-3	-2.60019	0	0	0	0

Table C.7: Crossover operator choice parameter tuning results

Crossover Operator	Highest # generations	Average # generations	IntraMD best	IntraMD average	InterMD best	InterMD average	EBCCB best	EBCCB average
building blocks	3379	3094.6	-241.85756	-211.027916	-10	-9.2	-8868.766327	-8796.452694
single	4495	4010.4	-78.84706	-73.8421	-10	-9.6	-8868.766327	-8650.698277

Table C.8: Using an elite archive parameter tuning results

Use Elite Archive	Highest # generations	Average # generations	IntraMD best	IntraMD average	InterMD best	InterMD average	EBCCB best	EBCCB average
false	3379	3094.6	-241.85756	-211.027916	-10	-9.2	-8868.766327	-8796.452694
true	3480	3251.6	-220.85683	-202.05801	-10	-9.6	-8868.766327	-7655.468213

Table C.9: Population size parameter tuning results

Population Size	Highest # generations	Average # generations	IntraMD best	IntraMD average	InterMD best	InterMD average	EBCCB best	EBCCB average
50	19744	18392	-413.53268	-362.800888	-9	-8.4	-8696.175143	-8541.220927
100	13333	11722.8	-365.02763	-341.461612	-9	-8.4	-8868.766327	-8688.493674
200	8253	7671.2	-330.70093	-288.728334	-10	-8.6	-8696.175143	-8501.486691
500	3379	3094.6	-241.85756	-211.027916	-10	-9.2	-8868.766327	-8796.452694
1000	1363	1201.8	-128.85548	-110.552806	-10	-9.2	-8868.766327	-8577.821069
2000	407	362.8	-53.84141	-50.14562	-10	-9.2	-8868.766327	-8723.011911

Appendix D

Pseudocode Estimated Build Cost of module Cache Breaks Difference

This appendix chapter contains the pseudocode that represents the code used to calculate the EBCCB metric.

Algorithm 1 Functionality used to calculate the difference in EBCCB value between the given solution and the base solution

```

1: function CalculateEBCCBDiff( $G, Sol, TDepS, ROCpM, LOCpM, ModsCT$ )
2:    $G = (V, E)$                                 ▷ Graph representing module dependencies, weighted by the number of class dependencies
3:    $Sol$                                           ▷ The solution
4:    $ROCpM$                                        ▷ The rate of change per module, based on commit history, adjusted for changes in this solution
5:    $LOCpM$                                        ▷ the number of lines of code per module adjusted for changes in this solution
6:    $ModsCT$                                        ▷ A mapping from module to module, containing the number of times they were changed together, used to correct the ROC to accurately represent the number of cache breaks
7:    $TDepS$                                        ▷ A mapping from each module to all its transitive dependencies
8:    $Er \leftarrow \emptyset$                         ▷ Set of removed edges
9:    $Ea \leftarrow \emptyset$                         ▷ Set of added edges
10:   $Ea, Er \leftarrow \text{GetChangedEdges}(G, Sol)$ 
11:   $TDa \leftarrow \emptyset$                         ▷ Transitive module dependencies that have been added by the changes in the solution
12:   $TDr \leftarrow \emptyset$                         ▷ Transitive module dependencies that have been removed by the changes in the solution
13:   $TDa, TDr \leftarrow \text{GetChangedTransDeps}(G, Ea, Er, TDepS)$ 
14:   $EBCCB \leftarrow \text{CalcEBCCB}(TDa, TDr, ROCpM, LOCpM, ModsCT)$ 
    return  $EBCCB$ 
15: end function

```

Algorithm 2 Determines the module dependencies (edges) which are added or removed by the class moves of a solution

```
1: function GetChangedEdges( $G, Sol$ )
2:    $Ea, Er \leftarrow \emptyset$ 
3:    $Ec \leftarrow Map$  ▷ Map of changed edges to their weights
4:   for  $class$  in  $Sol.changes$  do
5:      $Ec \leftarrow Ec \cup module\_edge\_changes$  ▷ Add every module edge affected by
      this class change to  $Ec$ , and weigh
      them by the effect of the change on
      the edge weight
6:   end for
7:   for  $edge$  in  $Ec$  do
8:     ▷ If the edge is a newly introduced one
9:     if  $edge \notin G.E$  AND  $Ec.edge > 0$  then
10:       $Ea \leftarrow Ea \cup edge$ 
11:    end if
12:    ▷ If all class dependencies between the
      vertices of this edge no longer exist
13:    if  $edge \in G.E$  AND  $G.E.edge - Ec.edge == 0$  then
14:       $Er \leftarrow Er \cup edge$ 
15:    end if
16:  end for
  return  $Ea, Er$ 
17: end function
```

Algorithm 3 Finds the changed transitive module dependencies which are caused by the added and removed module dependencies of a solution

```

1: function GetChangedTransDeps( $G, Er, Ea, TDep$ s)
2:    $vToCheck \leftarrow trans\_deps\_Er\_Ea$        $\triangleright$  All transitive dependencies of destination vertices in  $Er$  and  $Ea$ 
3:    $knownRevDeps \leftarrow Map$                $\triangleright$  A mapping from modules to their known transitive reverse dependencies
4:    $Vv \leftarrow \emptyset$                                  $\triangleright$  Visited vertices
5:    $Vcircular \leftarrow \emptyset$                      $\triangleright$  Nodes that can be found through its own transitive reverse dependencies
6:    $circular \leftarrow false$ 
7:    $res \leftarrow Map$                                  $\triangleright$  Mapping of all checked vertices to a mapping of the found reverse dependencies and whether they are reachable or not
8:    $TDa, TDr \leftarrow \emptyset$ 
9:   for  $v$  in  $vToCheck$  do
10:     $vRes, thisCircular, knownRevDeps, Vcircular \leftarrow$  RecursiveEdgeCheck( $v, G, Er, Ea, Vv,$   $knownRevDeps, reachable \leftarrow true,$   $edgePrevExists \leftarrow true, Vcircular$ )
11:     $circular \leftarrow circular$  OR  $thisCircular$ 
12:     $res.v \leftarrow vRes$ 
13:  end for
14:   $res \leftarrow updated\_res$                          $\triangleright$  Update the reachable vertices of each vertex in  $Vcircular$  with the reachable vertices of all other vertices in  $Vcircular$ 
15:   $res \leftarrow updated\_res$                          $\triangleright$  Update the reachable vertices of all vertices that can reach a vertex in  $Vcircular$  with that vertex' reachable vertices
16:  for  $v$  in  $res$  do
17:    for  $vr$  in  $res.v$  do
18:      if  $res.v.vr == true$  AND  $v \notin TDep$ s. $vr$  then
19:         $TDa \leftarrow TDa \cup (vr, v)$ 
20:      else if  $res.v.vr == false$  AND  $v \in TDep$ s. $vr$  then
21:         $TDr \leftarrow TDr \cup (vr, v)$ 
22:      end if
23:    end for
24:  end for
  return  $TDa, TDr$ 
25: end function

```

Algorithm 4 Recusively checks the reverse transitive dependencies of a module (vertex). Returns the found dependencies, whether they are reachable, whether there exists a circular dependency and which modules are part of that circle. It saves the results for each modules in a dynamic programming way.

```

1: function RecursiveEdgeCheck( $v, G, Er, Ea, Vv, knownRevDeps, reachable, edgePrevExists, Vcircular$ )
2:    $vRes \leftarrow \emptyset$ 
3:    $thisCircular \leftarrow false$ 
4:   if  $v \in Vv$  then ▷ A potential circular dependency is found
5:      $vRes \leftarrow v : edgeToPreviousExists$ 
6:      $thisCircular \leftarrow reachable$ 
7:      $Vcircular \leftarrow Vcircular \cup v$ 
8:     return  $vRes, thisCircular, knownRevDeps, Vcircular$ 
9:   end if
10:  if  $v \in knownRevDeps$  then ▷ v's reverse dependencies are known
11:    return  $knownRevDeps.v.vRes, knownRevDeps.v.thisCircular, knownRevDeps, Vcircular$ 
12:  end if
13:   $Vv \leftarrow Vv \cup v$ 
14:   $revDeps \leftarrow$  all  $vo$  where  $(vo, v) \in G.E$  OR  $(vo, v) \in Ea$ 
15:  for  $vo \in revDeps$  do ▷ Recursively check all reverse dependencies
16:     $depExists \leftarrow (vo, v) \notin Er$ 
17:    if  $depExists$  then
18:       $voRes, otherCircular, oknownRevDeps, oVcircular \leftarrow$  Recur-
19:       $siveEdgeCheck(vo, G, Er, Ea, Vv, knownRevDeps, reachable, edgePrevExists \leftarrow true, Vcircular)$ 
20:       $vRes.vo \leftarrow true$ 
21:      for  $voo \in voRes$  do
22:        if  $voo == v$  then
23:           $Vcircular \leftarrow Vcircular \cup v$ 
24:        end if
25:        if  $voo \notin vRes$  then
26:           $vRes.voo \leftarrow voRes.voo$ 
27:        else
28:           $vRes.voo \leftarrow vRes.voo$  OR  $voRes.voo$ 
29:        end if
30:      end for
31:       $voRes, otherCircular, oknownRevDeps, oVcircular \leftarrow$  Recur-
32:       $siveEdgeCheck(vo, G, Er, Ea, Vv, knownRevDeps, reachable \leftarrow false, edgePrevExists \leftarrow$ 
33:       $false, Vcircular)$ 
34:      if  $vo \notin vRes$  then
35:         $vRes.vo \leftarrow false$ 
36:      end if
37:      for  $voo \in voRes$  do
38:        if  $voo \notin vRes$  then
39:           $vRes.voo \leftarrow false$ 
40:        end if
41:      end for
42:       $thisCircular \leftarrow otherCircular$  OR  $thisCircular$ 
43:    end for
44:    if  $v \notin vRes$  then ▷ Add reachability of v for the previous vertex
45:       $vRes.v \leftarrow edgePrevExists$ 
46:    else
47:       $vRes.v \leftarrow vRes.v \cup edgePrevExists$ 
48:    end if
49:     $Vv.remove(v)$ 
50:     $knownRevDeps.v \leftarrow vRes, thisCircular$ 
51:    return  $vRes, thisCircular, knownRevDeps, Vcircular$ 
52:  end function

```

D. Pseudocode Estimated Build Cost of module Cache Breaks Difference

Algorithm 5 Calculates the EBCCB metric (relative) value, given the changed transitive dependencies, Rate of Change per module, Lines of Code per module and the amount of times module pairs are changed together.

```
1: function CalcEBCCB(TDa, TDr, ROCpM, LOCpM, ModsCT)
2:   costChange  $\leftarrow$  0
3:   for (v1, v2)  $\in$  TDa do
4:     cacheBreaks  $\leftarrow$  ROCpM.v2 - ModsCT.v1.v2
5:     cost  $\leftarrow$  LOCtoCost(LOCperModule.v2) * cacheBreaks
6:     costChange+ = cost
7:   end for
8:   for (v1, v2)  $\in$  TDr do
9:     cacheBreaks  $\leftarrow$  ROCpM.v2 - ModsCT.v1.v2
10:    cost  $\leftarrow$  LOCtoCost(LOCperModule.v2) * cacheBreaks
11:    costChange- = cost
12:   end for
13:   return costChange
13: end function
```
