Enterprise-level search-based software remodularisation using domain knowledge

Master's Thesis

Rolf de Vries

Enterprise-level search-based software remodularisation using domain knowledge

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Rolf de Vries
born in Alkemade, the Netherlands

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

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

Enterprise-level search-based software remodularisation using domain knowledge

Author: Rolf de Vries
Student id: 4452518
Email: r.b.devries@student.tudelft.nl

Abstract

As software systems evolve over time, the quality of its structure and code degrade unless developers regularly maintain it, requiring significant effort. Automated tools to help developers maintain software have been well-studied in the past. In particular, software remodularisation tools focus on improving the code structure quality with minimal effort by suggesting changes to the developers to obtain an improved modularisation. While there has been considerable research on automated software remodularisation, it often faces one or more of the following three shortcomings. First, the approach is applied to small or medium-size codebases, raising the question of whether it scales to large codebases. Second, the results are not validated by the developers of these codebases. Last, the algorithm optimises only from a code quality metrics point of view, not considering the perspective and knowledge of developers. In this thesis, we propose an approach to capture developers' domain knowledge of a large-scale object-oriented codebase, which uses an NSGA-III algorithm to suggest remodularisations that improve code structure quality and adhere to developer knowledge. Additionally, the results of the algorithm are validated by the developers. The results in this thesis show that with little effort, the domain knowledge of developers can be captured and used to improve the suggestions made by the algorithm.

Thesis Committee:

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

Preface

To start, I would like to thank everyone who made my study and thesis possible. I look back on the past seven years at the TU Delft with fond memories. The first people I want to thank are Dieuwer, Jaap and Jip, going back to the start of our bachelor Computer Science, for the fun and friendship we had, both in and outside the college classrooms. I am extra grateful for Maurício, both for suggesting the possibility of doing my Thesis at Adyen and for the amazing guidance and feedback he has given me during it. Additionally, I want to thank Arie van Deursen and Neil Yorke-Smith for their time and effort in my thesis committee.

I want to thank everyone at Adyen for making it possible to do my thesis and research there, especially the teams that participated in the case studies. To everyone in the Containerization team, thank you for your support during the last nine months and for all the fun activities we did. To my company supervisor Daan in particular, thank you for your help, feedback and insights and for not getting bored of our weekly discussions. I want to thank Casper for the regular discussions about his thesis and for taking the time to explain his code and thought process when needed.

To finish off, I want to thank my family, friends and girlfriend for their love and support over the years.

Rolf de Vries
Hilversum, the Netherlands
May 12, 2022

Contents

List of Figures

Chapter 1

Introduction

Software systems evolve as time progresses. As new features are added and bugs are removed, the system's architecture deviates, eroding the initially planned design [34]. When a software system faces many changes but little maintenance for the quality of the code, the quality of the code decreases and the technical debt of the system increases [21]. This decrease in quality makes the system susceptible to faults, harder to maintain and reduces the quality of software modules [30]. Additionally, it becomes more difficult for developers to understand and add new functionality to the code, especially for enterprise-level codebases, which are too large for any developer to understand completely. As a result of this non-complete understanding of the codebase, developers can make sub-optimal decisions when developing, requiring an effort to maintain software quality from degrading [68, 55].

Technical debt is introduced into the code when software systems are not adequately maintained to uphold their quality [21]. The technical debt can, over time, lead to increased costs for maintaining, running, testing and deploying code. For example, duplicate code is not removed, and modules become tightly coupled or loosely cohesive, making the job for developers harder to perform. One approach developers can use to restore the quality of their software systems is to remodularise the codebase.

Software modularisation is the practice of grouping related classes into modules to achieve a good code structure. Coupling and cohesion [58] are two abstract concepts by which the quality of a software system can be expressed. Cohesion is the degree to which classes in a module belong together. When two or more classes in a module use one another to serve a shared responsibility, these classes are considered cohesive. When classes in the same module do not rely on each other to perform, they are considered incohesive. Generally, the cohesiveness of a module is defined in terms of its number of intra-dependencies. When a group of classes in the same module has strongly related functionality, they likely belong in the same software module, as the cohesiveness among them is high. Coupling is the degree to which classes from different modules rely on each other to function. The more a class relies on classes outside its own module, the more tightly coupled the module of that class is with other modules. High coupling is undesirable, as a change to one class can propagate to many different modules, thereby affecting a large part of the system and causing unwanted functionality changes outside the original change's intended scope. In general, the coupling of a module is defined in terms of its number of inter-dependencies.

When modules are highly cohesive and loosely coupled, they serve a distinct purpose and reduce unwanted propagation of changes during software development.

Metrics for coupling and cohesion are popular choices for automated software remodularisation approaches. At the heart of these approaches is an algorithm that uses metrics to improve the modularisation of software systems. A considerable amount of research has been done to study and propose software modularisation approaches. While some used a manual approach [67, 44], it is more common for an automated approach to be used, involving clustering [46, 47, 78, 7, 61, 49, 6, 56, 74] or genetic algorithms [46, 47, 62, 55, 65, 12, 63, 68, 53, 27, 9, 33, 34]. Regardless of the underlying technique, the goal of these approaches is to provide developers with a better modularisation of their software systems. Automated techniques can be beneficial, as software maintenance is an expensive part of software development [11]. However, a shortcoming of most of these techniques is that they only measure how good a software system is from the perspective of code quality metrics such as coupling and cohesion, without considering whether the suggested modularisation makes sense from the perspective of developers. As the results of these approaches are not validated by the developers of the software system, it raises the question of how well the approaches perform in practice. To quote Alizadeh et al. [3]: "Manual refactoring is time-consuming, error-prone and unsuitable for large-scale, radical refactoring. On the other hand, fully automated refactoring yields a static list of refactorings which, when applied, leads to a new and often hard to comprehend design." For automated tools to provide useful modularisations, their results must be validated by developers, and their domain knowledge should be taken into account.

In this thesis, we continue on the work of Schröder et al. [68], who proposed a software remodularisation approach that scaled to enterprise-level codebases. In their work, they used an NSGA-II algorithm to optimise the Adyen codebase for the following metrics: Intra-Module Dependency Coupling (Cohesion), Inter-Module Dependency Coupling (Coupling), Estimated Build Cost of Cache Breaks (EBCCB) and the number of *move-class* refactorings. Their work showed that the software remodularisation approach scales to large software systems, and they validated the suggested remodularisations with developers of the codebase. From a metrics perspective, the approach improved the quality of the codebase significantly. When validating the results with developers, it was found that the algorithm successfully identified classes that should be moved but was inaccurate in determining to which modules the classes should be moved. As a suggestion for further research, the authors suggest including the perspective of developers in the algorithm. Including this developer perspective for automated software modularisation has been done in previous research [53, 27, 9, 33, 34]. However, these approaches either require a considerable amount of effort from developers, or the reusability of the developer input is limited.

We formulate the software remodularisation problem as a multi-objective problem, to which we apply an NSGA-III algorithm that remodularises object-oriented software systems through *move-class* refactorings. A *move-class* refactoring moves a class in a software system from one module to another. As we continue on the work of Schröder et al., we will also apply our algorithm to the Adyen codebase. Among the objective functions are traditional code quality metrics, such as coupling and cohesion, and a novel metric that expresses developer knowledge. When automated software remodularisation approaches only consider code quality metrics, solutions often do not make sense from the perspective of developers [34]. The NSGA-III algorithm can be configured to consider a complete codebase while only focusing on improving a specific part of it. This configuration allows development teams to receive suggestions related to the code they are familiar with without introducing unwanted cyclic dependencies throughout the entire codebase and adhere to the domain knowledge the developers have of their part of the codebase. Developer knowledge is expressed in rules and freezes, which can vary in granularity from being specific to classes to covering multiple modules, allowing the algorithm to understand whether *move-class* refactoring suggestions violate domain rules developers have, e.g. classes in a *front-end* module may not depend on a *database* module. **The goal of this approach is to suggest a remodularisation that both improves code quality and adheres to developer domain knowledge, with a limited number of move-class refactorings to remain understandable.**

To validate this approach, we consulted various developer teams at Adyen for a series of interviews. During each interview, the teams gave feedback on the remodularisation suggestions of the algorithm. From this feedback, new domain rules were established and used as input for the algorithm. Throughout these interviews, a knowledge base representing the domain knowledge of the developers was established, and the results show that including this domain knowledge positively influences the algorithm's decision-making when suggesting remodularisations.

The results of this thesis show the importance of validating remodularisation results with developers and including their perspectives during the remodularisation process. Before consulting our algorithm's remodularisation suggestions with Adyen developers, we performed an empirical study to analyse the impact the algorithm can make from a code structure quality perspective. The empirical study showed that the algorithm could improving *Intra-Module Dependency Coupling (Cohesion)* by 8.6%, *Inter-Module Dependency (InterMD) Coupling (Coupling)* by 6.67% and the *Largest module size* by 1.97%. While these results show that the algorithm could significantly improve the code structure quality of the Adyen codebase, the results were not validated by developers. The case studies following the empirical study showed that improvements to the metrics are not sufficient for developers to always accept the remodularisation suggestions, as the suggestions violated several domain rules that developers follow during development. By including this developer domain knowledge, the algorithm could suggest remodularisations that improved the code quality and adhered to these domain rules, which positively affected developer acceptance of the suggestions.

For the remainder of this thesis, we will explain the company Adyen and the contributions this thesis makes. Next, we further discuss software modularisation, code structure quality metrics and genetic algorithms and dive deeper into the work of Schröder et al. [68]. The chapter after describes related work. This chapter is followed up by a detailed description of the functionality of our proposed NSGA-III algorithm and the domain knowledge input. In the chapter after that, we perform an empirical study to understand the algorithm's performance from a metrics point of view. The chapter following the empirical study focuses on case studies that include Adyen development teams to analyse the impact of developer knowledge on the algorithm's performance. Finally, the approach and results are discussed, conclusions are drawn, and future work beyond this thesis is proposed.

1.1 Adyen

Adyen was founded in 2006, aiming to revolutionise the payments technology industry. Adyen provides a payments platform for merchants and provides payment and points of sale terminal services. In addition, Adyen is a gateway, acquirer and more recently obtained banking licenses in various countries [1].

Since its founding, both the company and its codebase have grown rapidly, now counting over 2000 employees and millions of lines of code, respectively. As the company grew, the services it provided increased as well, resulting in new software being added to the existing codebase, thus increasing the complexity of the code. Furthermore, as codebases are subject to feature enhancements, they deviate from their original architecture and become increasingly harder to manage [67]. The codebase of Adyen is no exception to this, as shown by the efforts made during a full week by a group of 20 developers spent on refactoring the largest module in the mono repository.

Originally, much of the code related to the core business was located in this module, which grew in size and complexity as the company grew. Over time increasing complexity resulted in the code being heavily intertwined within the module, making it difficult to introduce changes that would not cascade throughout the module and to other modules. Since then, efforts have been made to extract clusters of code into specialised modules, reducing the complexity of the large module and making the code easier to understand. Isolating these modules ensures that changes introduced to their code are less likely to propagate to other modules. Despite the efforts, the large module accounts for a significant portion of the repository, as it offers functionality that relies on code also located in the same module. The work performed in the refactoring week mentioned above resulted in the largest module being reduced in size by 5%, indicating the amount of work required for such an effort and showing potential for the application of automated software maintenance tools.

1.2 Contributions

This thesis makes the following contributions:

- A multi-objective genetic algorithm to tackle the software remodularisation problem for enterprise-level large-scale codebases, able to significantly improve such codebases from a metrics perspective while adhering to its developers' domain knowledge.

- The results of the algorithm are validated with multiple development teams. This approach shows both the importance of validating results with the actual developers and generates feedback that can be used to obtain better remodularisation suggestions.

[1] `https://www.adyen.com`

- A framework to represent developer domain knowledge is introduced, allowing for developer feedback of varying granularity. The feedback is reusable, allowing it to be used for future runs of the algorithm.

Chapter 2

Background

In this section, we discuss software modularisation, code structure quality and genetic algorithms, and the work by Schröder et al. [68]. The work of Schröder et al. is the predecessor of this thesis, which showed that genetic algorithms could be used for the modularisation problem on large-scale enterprise-level software systems. This was shown on the codebase of Adyen. This section lays the groundwork for the topics above and explains the genetic algorithm, NSGA-III, used in this thesis.

2.1 Software modularisation

Software modularisation is the process of dividing classes into modules such that each module works independently. Experts agree that software systems that are well modularised are easier to develop, maintain and understand for developers [71, 69, 64], while poorly modularised systems are associated with decreased testability and increased efforts to maintain and comprehend them [62]. Modularisation efforts can be manual, automatic or a hybrid approach.

With manual modularisation, developers identify clusters of classes that belong together and divide the software system into modules. Ideally, each module relies as little as possible on other modules, while the classes within a module use other classes in the same module to work. The measure of connectivity between modules is called coupling, while the measure of connectivity of classes within a module is called cohesion. Both of these concepts are further explained in Section 2.2. Automatic modularisation is the process of automatically partitioning a software system based on metrics that quantify how well partitioned the software system is. Metrics for coupling and cohesion are popular choices for such automated approaches. However, it is significantly more difficult for automatic modularisation to consider code connectivity that is hard to express in such metrics. Lastly, developers can choose a combined approach where they can interact with the automatic tool to obtain to steer the modularisation process to a desired outcome. Several examples of this last approach are discussed in Section 3.2.4.

2.1.1 Software remodularisation

Software remodularisation is a software modularisation approach that minimises the number of changes. The remodularisation approach is the one we use in this thesis, though we use the terms modularisation and remodularisation interchangeably. While large modularisations can significantly impact structural quality, they are also harder to understand and implement. When modularisation suggestions are harder to understand, it is likely that they are either rejected more often or have an increased probability of developers introducing mistakes when implementing them. A small list of changes will not drastically improve structure quality but requires far less work by improving the structure gradually. As Sam Newman said [57]: "Think of our monolith as a block of marble. We could just blow the whole thing up, but that rarely ends well. It makes much more sense to just chip away at it incrementally." While this was said in the context of breaking down monoliths into micro-services, this approach is also applicable to gradually improving the modularisation of a large-scale codebase.

2.2 Code quality

Various measures can express the quality of code. For example, one can measure how fast the code runs, whether the output is correct, how quickly it compiles or whether it is properly tested. One can use two distinct notions to assess code quality: functional and structural quality. Functional quality expresses how well the software fits the functional requirements such as quality of results, while structural quality expresses how well non-functional requirements such as maintainability are met. Code structure quality metrics such as coupling and cohesion fall under the latter category. As we propose an algorithm that suggests move-class refactorings to improve the structure of software systems in this thesis, we use code structure metrics to produce modularisations with improved structural quality.

2.2.1 Code structure quality

The concepts of coupling and cohesion are not new, being introduced as far back as 1968 by Constantine [18]. Using these metrics, software developers can improve the structural quality of their software systems. In the decades since 1968, researchers have spent a significant amount of time defining metrics to express the quality of software systems, some of which will be discussed in detail in Section 3.1. For now, we will discuss the concepts of coupling and cohesion.

Coupling and cohesion both represent how one software unit interacts with other units in a software system. These units can vary in granularity, ranging from methods and functions to classes or modules in object-oriented software systems. At module level, in the context of a software system that has been divided into modules with each module serving a specific purpose, cohesion expresses how well the classes within a module work together. In contrast, coupling expresses how much a module relies on other modules to function.

Both high coupling and high cohesion have the effect that changing one class will result in changes in other classes. However, the difference between coupling and cohesion is that with high coupling, these changes can unintentionally propagate to other parts of the system. In contrast, for high cohesion, those changes are related to the same module. These two aspects are often used in determining the code structure quality of software systems, as we will see in Chapter 3.

2.3 Genetic algorithms

Genetic algorithms (GA) are search heuristics inspired by the process of natural selection. They operate by simulating a population that evolves over time, where each member of the population represents a solution to the problem to which the GA is applied. As time progresses, good solutions have a better chance of survival than bad solutions. To determine whether solutions are good or bad, genetic algorithms use objective functions to assess the fitness of a solution. Using the fitness of solutions, a GA can compare solutions. Therefore, objective functions must be chosen such that their values represent (un)desirable traits. GAs improve the fitness of solutions in their population by drawing on operators inspired by the biological concepts of reproduction, mutation and selection. These operators are applied in a loop, where each iteration is called a generation. At the end of each generation, the algorithm can either stop, based on some stopping criterion, or perform another generation. The general idea behind such an approach is that the solutions in the population improve over time in terms of the objective functions, which represent desired traits of solutions. We will further discuss the applicability of GAs for software modularisation in Section 2.4 and Chapter 3. To illustrate how a GA operates, we propose a simple problem to solve: "*Find a binary string of length five such that the sum of its bits is 5*". While the problem can easily be brute-forced to obtain the answer 11111, this example is used to help understand how a GA would attempt to solve a problem.

2.3.1 Population

Genetic algorithms start by initialising their population, usually by randomly generating it. The size of the population influences the performance of the algorithm. A population that is too small is less likely to have a diverse outcome, while a large population requires more calculations per generation, thus slowing down the algorithm. Each member of the population is referred to as a solution or chromosome. Every solution consists of a number of genes, which represent features of the solution. Figure 2.1 illustrates these different concepts for a population of size four.

2.3.2 Crossover

Crossover is the conventional term used to describe reproduction for GAs. The algorithm selects two solutions from its population and recombines them to create two new solutions. This selection and recombination continues until the population has doubled in size, which is eight in the current example. One of the simplest crossover operators is *single-point*

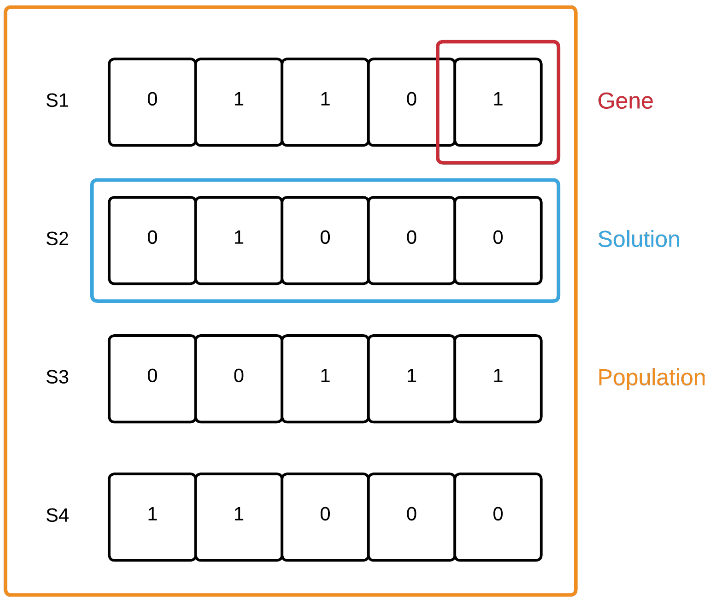


Figure 2.1: Genetic algorithm population

crossover. Single-point crossover splits a parent into two sections and assigns one section to the one child and the other to the other child at random, and vice versa for the other parent. Figure 2.2 illustrates single-point crossover for solutions S1 and S2. Child C1 receives the first part from parent S1 and the second part from parent S2, indicated by their respective colours. Child C2 receives the first part from parent S2 and the second part from parent S1, also indicated by their colours. The location of the crossover point can vary every time the operator is applied, allowing the same pair of parents to result in many different pairs of children.

2.3.3 Selection

The use of a selection operator is inspired by the mechanism behind natural selection: "survival of the fittest." Solutions in the population that are "better" should have a higher chance of surviving and reproducing than those that are "worse". The "goodness" of a solution is called its fitness. For the example problem statement, the fitness of a solution is simply the sum of its bits. The fitness of each solution in the population shown in Figure 2.1 is listed in Table 2.1. The GA uses these fitness values to pick solutions for crossover randomly. A popular selection operator is *roulette wheel selection*, which assigns a probability for a solution to be picked equal to its contribution to the total fitness. The total fitness of the population in this case is $3+1+3+2=9$, meaning that solutions S1 and S3 both account for $\frac{1}{3} \approx 33.3\%$ of the population fitness, solution S4 for $\frac{2}{9} \approx 22.2\%$ and solution S2 for $\frac{1}{9} \approx 11.1\%$. Each solution has a probability equal to its fraction of the total fitness of being picked as a parent to create offspring.

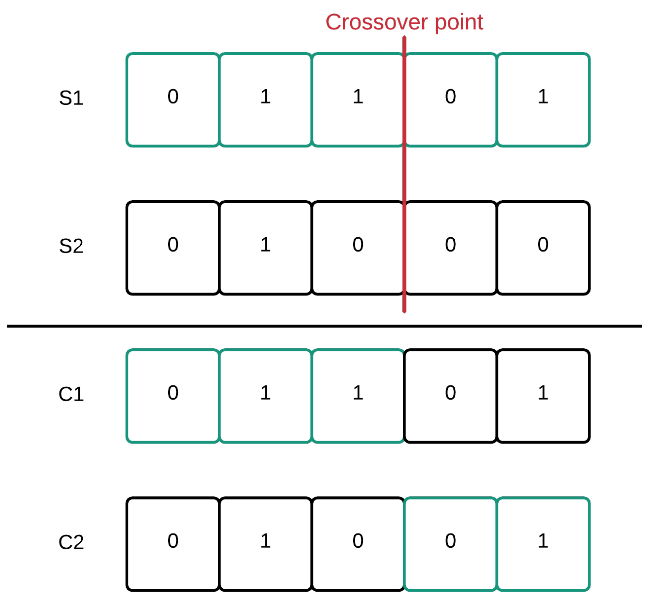


Figure 2.2: Single-point crossover example

Solution	Fitness
S1	3
S2	1
S3	3
S4	2

Table 2.1: Figure 2.1 solution fitness values

Selection can also be used to determine survivors in the population. As mentioned in Section 2.3.2, crossover doubles the population in size due to each pair of parents creating a pair of offspring. A number of solutions equal to the original population size must be picked from this population. This can also be done using a selection operator, of which various types are discussed in Sections 2.4 and 4.7. The main idea behind selection is that by giving solutions that are deemed good a higher chance of surviving and reproducing, they, in turn, create new offspring that is even better, thus gradually improving the fitness of solutions in the surviving population.

2.3.4 Mutation

Mutation is the ability of a GA to introduce changes to a solution that could take a long time or even might never happen through crossover. While crossover allows the algorithm to create new solutions from existing ones, it can never assign a value to a gene that does

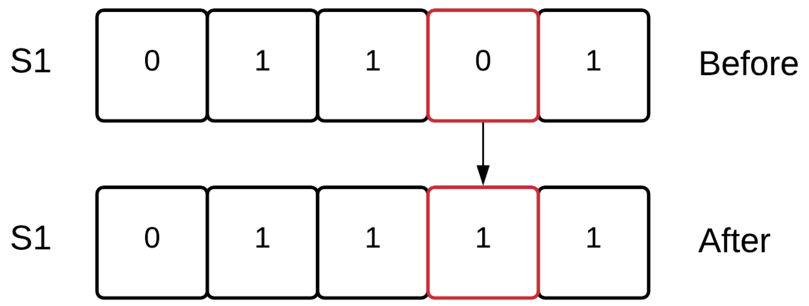


Figure 2.3: Genetic algorithm mutation example

not already exist in the population. For example, in the population displayed in Figure 2.1, only solution S3 has a 1 as value for its fourth gene. If S3 were not to survive a generation, it could happen that none of the solutions in the population had a 1 for their fourth gene. In that case, all subsequent solutions would always have a zero as value for that gene, making it impossible for the algorithm to produce a solution where the sum of its genes is five. Through mutation, the GA can change the value of genes in solutions, mimicking the mutation of genes in nature. For example, Figure 4.11 shows how solution S1 is mutated such that its fourth gene becomes a 1.

2.3.5 Stopping criteria

Even though the algorithm can create new solutions from existing ones with crossover and to introduce never-before-seen changes to solutions with mutation, it can still take the algorithm a significant amount of time to produce a solution that satisfies the problem. To prevent the algorithm from potentially running indefinitely, one can specify stopping criteria so that the algorithm terminates even if it does not find an optimal solution. Usually, these stopping criteria define some maximal number of generations the algorithm can iterate over or a certain amount of time before the algorithm must stop. Once such criteria are met, the algorithm can return the best solutions it has found so far. Continuing on the scenario from Section 2.3.4, where all solutions in the population of the algorithm have a 0 as value for the fourth gene, due to the random nature of the mutation operator, it can take a substantial amount of time until the population once again contains solutions with value 1 for the fourth gene. If the problem must be solved in a limited amount of time, the algorithm might terminate before producing a solution 11111. In this case, the best solution the algorithm could have produced is 11101, which has a fitness of 4.

2.3.6 Multi-objective genetic algorithms

For the example problem mentioned above, a single objective function is sufficient for the algorithm to create a solution that satisfies the problem. However, more complex problems require multiple objective functions to enable the algorithm to produce satisfactory solutions, as one fitness value cannot represent all the desired traits. When multiple objective functions are used, the genetic algorithm is known as a Multi-Objective Genetic Algorithm (MOGA), a type of Multi-Objective Evolutionary Algorithm (MOEA). MOGAs have been shown to be widely applicable for large search-based problems. Since the problem at hand in this thesis is a multi-objective one, the changes required for a genetic algorithm to deal with multiple objectives are discussed.

Instead of having one fitness value for each solution, the fitness of a solution becomes represented by a vector of size equal to the number of objective functions. This transforms the problem from one-dimensional to n-dimensional, with n being the number of objective functions. As a result, the selection operator must now order solutions based on all their fitness values. For example, for a two-dimensional maximisation problem, if there are two solutions with respective fitness vectors $(1,1)$ and $(2,0)$, one cannot say whether the first is better than the second. For the first fitness value, the second solution performs better than the first, but the opposite holds true for the second fitness value. In such cases, the two solutions are said to be non-dominating. For maximisation problems, a solution Sol_A is said to dominate another solution Sol_B if for each objective function $F : F(Sol_A) \geq F(Sol_B)$ and there exists at least one objective function such that $F(Sol_A) > F(Sol_B)$. In other words, all fitness values of solution Sol_A are at least as high as those of solution Sol_B, and at least one fitness value is strictly higher. Every solution in the population that is non-dominated is called a Pareto-optimal solution. With this concept of domination, the GA can divide the population into fronts.

All Pareto-optimal solutions in the population form the Pareto-optimal front. The next front contains all solutions that are only dominated by solutions in the Pareto-optimal front. The front after that contains all solutions that are only dominated by solutions in the previous fronts, and so on until all solutions are allocated to a front. Using these fronts, the selection operator can select solutions based on the front they are in, favouring solutions that are not dominated over those that are.

2.4 Expanding Search-Based Software Modularisation to Enterprise-Level Projects: A Case Study at Adyen, by Schröder et al.

The work in this thesis continues on the research of Schröder et al. [68], which showed that search-based software modularisation could be scaled to enterprise-level software systems. In their work, Schröder et al. describe a graph-based approach to modularising the codebase of Adyen. To explore the solution space of modularisations, an NSGA-II [25] algorithm is employed as a search-based approach. The algorithm modularises by performing move-class refactorings to move classes from one module to another.

Instead of having each solution represent a complete modularisation by assigning each class to a module, the NSGA-II algorithm represents a list of move-class refactorings that, when applied, result in a new modularisation of the Adyen codebase. Modelling solutions in such a way improves performance by reducing the cost of evaluating solutions.

2.4.1 Building block preserving crossover

Traditional crossover operators such as single-point, two-point or uniform crossover cannot respect the module structure of solutions when creating offspring. This is because these operators consider solutions as a list of classes, where each class location is copied from a parent to a child. The resulting modules in the offspring can therefore differ significantly from those of their parents, causing vastly different modularisations. To attempt to maintain the module structure of a parent, the authors' NSGA-II algorithm employs a building block preserving crossover operator inspired by Harman et al. [35]. This operator aims to maintain module structure by performing uniform crossover at module level rather than class level. However, since modules can consist of different sets as classes per parent, this can cause conflict during crossover.

An example is visualised and covered in more detail in Section 4.8, but the general idea is that a child C_1 can receive a module from parent P_1, where that module contains a class that C_1 earlier received in a different module from parent P_2. This would cause child C_1 to contain that class in two different modules, while the class would not be located in child C_2. To resolve this, the second time C_1 receives a class, that class instead goes to C_2.

2.4.2 Selection

For selection, the authors' NSGA-II algorithm uses tournament selection of size 5, using partial order and crowding distance to decide on draws. A traditional tournament selection of size n picks n solutions from the population at random. It returns a solution with a probability corresponding to the ratio of its fitness to the total fitness in the tournament [52]. Variations to this method can occur by allowing a solution to appear multiple times in a tournament or always returning the best solutions in a tournament. When the fitness of solutions is multi-objective, solutions can be assigned to fronts using the domination principle as explained in Section 2.3.6.

The selection operator of Schröder et al. uses partial order and crowding distance to deal with these solutions. Partial order favours solutions of lower rank, meaning they are in a better front. When solutions are of the same rank, thus in the same front, the crowding distance is used to order these solutions, choosing solutions with high crowding distance over those with lower values. This is the same selection approach as used in the original NSGA-II paper by Deb et al. [25].

2.4.3 Mutation

Move class refactorings are introduced when a solution is mutated. The authors' NSGA-II algorithm uses two mutation operators, each having an equal probability of being applied to a solution. The first operator is a min-cut based operator. Starting at a random class, the dependency graph of that class within the module is created up to a certain size. Next, the min-cut of the graph is determined using the Stoer-Wagner algorithm [72], cutting the graph into two sub-graphs. The sub-graph that contains the originally picked class is then moved to a related module. The size of this sub-graph varies on the location of the min-cut.

The second operator works by moving a neighbourhood of classes, inspired by the approach of Fraser and Arcuri [31]. First, a class C is picked at random. Then, all classes that directly depend on C or that C directly depends on are determined. Then, with decreasing probability, another class from the directly related classes is grouped with C, and the related classes of the group are determined. Once no more classes are added to the group, a related module of the group is picked at random as the destination for the group of classes.

For both operators, to allow the algorithm to create new modules, a probability of creating and substituting a new module for the related module can be provided.

2.4.4 Fitness evaluation

To evaluate and compare the performance of solutions, Schröder et al. used four optimisation variables:

- Coupling (Inter-Module Dependency Coupling) between modules.
- Cohesion (Intra-Module Dependency Coupling) within modules.
- The number of move-class refactorings (#moves).
- The Estimated Build Cost of module Cache Breaks (EBCCB).

For coupling, the average amount of Inter-Module Dependency Coupling (InterMD) was used. For cohesion, the number of class dependencies within a module divided by the maximum possible amount of class dependencies within that module, Intra-Module Dependency Coupling (IntraMD), was used. The number of move-class refactorings indicates how much work is required by developers to implement all the suggested changes. The Estimated Build Cost of module Cache Breaks indicated how often the cache of a module and its transitive dependencies breaks due to changes in that module. With a codebase of a size like Adyen's, compiling all code for every change slows down development speed considerably. Adyen caches the compiled modules to combat this, only recompiling if the module changes or any module that the module (transitively) depends on changes. High EBCCB values indicate high transitive coupling. As a result, changing a module will break many caches, which increases compilation time.

Coupling, #moves and EBCCB are minimisation variables, while cohesion is a maximisation variable. By minimising coupling, modules become less dependent on others. Maximising cohesion means that classes in the same module rely on each other to function. Minimising the number of move class refactorings ensures that the algorithm does not produce a long list of refactorings since those are unlikely to be accepted by developers. Instead, by keeping the number of refactorings low, the list of refactorings is more concise and better understandable. Finally, minimising EBCCB reduces transitive coupling in the codebase and ensures that compilation time does not grow out of hand for future changes.

2.5 NSGA-III

NSGA-III [23, 40] is an evolutionary algorithm that follows the NSGA-II framework [25] and applies a reference-point-based approach. Like NSGA (Non-dominated Sorting Genetic Algorithm) and NSGA-II, NSGA-III emphasises non-dominated population members. Additionally, NSGA-III emphasises those non-dominated population members close to supplied reference points. NSGA-II algorithms have proven efficient in solving two- and three-objective optimisation problems but have difficulty dealing with many-objective optimisation problems, which have more than three objectives.

2.5.1 Many-objective optimisation problems

As the number of objectives grows, the proportion of non-dominated solutions scales exponentially [23]. This slows down the search process of the GA, as the crowding distance operator becomes computationally more expensive, and most of the population is located in the Pareto front [24, 26]. Most multi-objective optimisation algorithms, like NSGA-II, emphasise non-dominated solutions. As the number of dimensions increases, so does the distance from a solution to others in the population [2]. When two parents that are distant from each other create offspring through crossover, those children will also be distant from their parents, which is counter-intuitive to the idea behind crossover and harms the algorithm's efficiency.

Visualising the search space becomes difficult, or even impossible, for problems with many objectives. While NSGA-III does not solve the visualisation problem, it can address the other problems stated by updating its selection procedure. Rather than selecting solutions based on crowding distance [25] or clustering [79], NSGA-III uses reference points to maintain diversity among the population.

2.5.2 Selection

While NSGA-III uses the NSGA-II framework, it functions significantly different when comparing their selection operator. NSGA-III uses reference points to maintain diversity among solutions in the population, even for a large number of objectives. Similarly to NSGA-II, solutions are sorted into fronts based on which solutions dominate them. The first front, also known as the Pareto-optimal front, contains all non-dominated solutions. However, where NSGA-III differs is how it decides on draws for solutions in the same front.

When the algorithm has a population of size N, the total number of solutions after crossover is $2N$. The selection operator is used to select N solutions that survive to the next generation, favouring good solutions in terms of fitness and maintaining diversity among the population. The $2N$ solutions are assigned to fronts based on the domination principle. The first front, F_1, contains all non-dominated solutions, F_2 contains the solutions only dominated by F_1. This goes on until front F_M, which contains the solutions dominated by solutions in F_{M-1} and previous fronts. Then, the operator assigns solutions to the surviving population P based on their front, starting with F_1 and continuing until the size of P is larger or equal to N. When $|P| = N$, the operator returns P, and the algorithm can continue.

However, if $|P| > N$, the operator must decide which solutions are not added to P such that $|P| = N$. Say that front F_L is the front that, when completely added to P, results in $|P| > N$. The selection operator uses the reference points to decide which solutions in F_L are added to P. Each solution so far in P, s.t. $P = F_0 \cup F_1 \cup \ldots \cup F_{L-1}$, is assigned to its closest reference point, resulting in an association count for each reference point. The solutions in F_L are also assigned to their closest reference point without increasing the association count of their reference point. The selection operator then picks a solution in F_L associated with the reference point with the lowest association count, breaking ties at random. This solution is then added to P, and the association count of its reference point is incremented. This is repeated until $|P| = N$, where the solutions picked from F_L are chosen because they have the biggest impact on the diversity of P.

2.5.3 Reference points

NSGA-III allows users to adopt two strategies for supplying reference points: reference points can be supplied structurally or defined manually by the user. In the original NSGA-III paper [23], a structural manner is used, called the "Das and Dennis systematic approach" [22], which uniformly distributes the reference points on a normalised $(M-1)$-dimensional hyperplane, with M equal to the number of objectives. By distributing the reference points uniformly, the algorithm is not biased to any objective function, allowing the algorithm to pursue maximal diversity across the objective space.

Chapter 3

Related Work

Over the past decades, software modularisation and maintenance have been well-studied topics for software engineering research. Software modularisation is the process of restructuring a codebase to improve its structure. This chapter discusses various ways to measure the structural quality of software systems. These metrics have been used in various algorithms and approaches to tackle the software modularisation problem in the past, which the following section outlines. Lastly, techniques to model user preferences and knowledge are described.

3.1 Software structure quality metrics

There are many metrics available to measure the structural quality of software systems. This section will outline metrics that have been used by related work to improve software structure quality.

Chidamber and Kemerer proposed a set of six metrics [17], which have been used and extended extensively in software quality research. These metrics are listed below:

1. Weighted Methods per Class (WMC): The sum of the weighted complexity of all methods in a class.

2. Depth of Inheritance Tree (DIT): The length from the class to the root of its inheritance tree.

3. Number of Children (NOC): The number of immediate sub-classes a class has in its inheritance tree.

4. Coupling Between Object classes (CBO): The number of classes a class is coupled with.

5. Response for a Class (RFC): The number of methods a class can call.

6. Lack of Cohesion in Methods (LCOM): Measures the lack of cohesion between pairs of methods in a class depending on whether they share class variables.

Basili et al. [8] studied the performance of these metrics as predictors of fault-prone classes by testing them on eight information systems. They showed that all metrics except LCOM outperformed traditional code metrics as predictors. In addition, the metrics from Chidamber and Kemerer were usable earlier on in the software development process.

While much work has been done on developing software quality measures, Briand et al. [14] argue that there is little understanding of which application many measures are helpful for. Therefore, they propose a unified framework to measure cohesion for software systems. Additionally, they criticise the lack of empirical validation in many of the works that proposed new measures and the quality of some works that do have empirical evidence. This work emphasises the importance of thorough empirical validation to show the use-cases and usability of software quality metrics.

Metrics for coupling and cohesion are some of the most well-known measures for software quality. Numerous researchers have used them to measure the class or module level structural quality of software systems, representing intra- and inter-connectivity. For example, Mancoridis et al. [46] proposed a metric, Modularisation Quality (MQ), that combines intra- and inter-connectivity into a single value by using a weighted sum of the two values. The resulting value for MQ is bounded between -1 and 1, where -1 means no cohesion while 1 means no coupling.

Allen et al. proposed information-theory approaches to measure coupling and cohesion, for an entire system [4] and at module level [5]. In their work, they refer to the framework of Briand et al. and adopt their definitions in their work. They propose module level measures based on information theory, one for cohesion and two for coupling; inter-module coupling and intra-module coupling. They show that these measures can be applied to measure the quality of modular software systems.

In addition to coupling and cohesion, Mahouachi [45] considered the semantic cohesiveness of packages and the refactoring effort as optimisation goals for software modularisation. They argue that "the name of a variable, method, or class, reveals the developer intent and it corresponds on internal indicators for the meaning of a program." This intention is not easily captured by traditional metrics; therefore, semantic evaluation is used. Using WordNet[1] information content, they calculated the semantic similarity between a class and packages to determine whether a class belongs to a package on a semantic level. In addition to improving the modularisation quality while minimising the needed effort, the algorithm was also able to accommodate for the domain knowledge captured in the semantics of the software elements. If software elements are not appropriately named, the semantic analysis may negatively affect the algorithm's performance.

Bavota et al. [10] combined structural and semantic measures to decompose packages into smaller, more cohesive packages, as no single metric captures all aspects of cohesion [50]. They used Information-Flow-based Coupling (ICP) [43] and Conceptual Coupling Between Classes (CCBC) [60] to measure the cohesion of packages. ICP measures the structural relation between classes, while CCBC measures the semantic relation. These metrics were chosen as they were shown to work well together while not correlating, meaning they measure different aspects. Their results show promise for using a combination

[1] `https://wordnet.princeton.edu/`

of semantic and structural measures to identify modularisations of packages that improve cohesion.

In addition to MQ, inter-cluster and intra-cluster connectivity, Bavota et al. [9] used the number of clusters and the number of isolated clusters as metrics to improve the quality of software systems. This approach was inspired by the Maximising Cluster Approach (MCA) proposed by Praditwong et al. [62]. In addition to MCA, Praditwong et al. also proposed the Equal-Size Cluster Approach (ECA), which is similar to MCA. However, instead of the number of isolated clusters, it uses the difference in size of the largest and smallest cluster as a metric. Both approaches showed promise for improving code structure quality, with ECA having the better performance of the two.

While software modularisation recommender tools often use cohesion and coupling to propose modularisations that are a balance between the measures, Candela et al. [15] argue that there is no empirical evidence that such balances are what developers need or even want. The authors performed two studies to assess the role of cohesion and coupling in modularising software systems. The first study objectively investigates by analysing 100 open-source systems whether their design balances coupling and cohesion or whether it favours one of the two metrics. The second study is more subjective, where the authors interviewed 29 developers that contributed to some of the 100 projects to understand what drives developers when they perform modularisations. The first study shows that almost all open-source projects have a design far from optimal in terms of coupling and cohesion. This is confirmed by the results of the second study, in which developers claimed that they are usually guided by more properties than just coupling and cohesion when modularising. The authors conclude by stating that these results indicate that cohesion and coupling are likely not enough to produce good modularisations from a developer perspective.

Chhabra et al. [16] proposed an approach that considers lexical dependencies in addition to structural dependencies for modularisation. Considering lexical dependencies allows the approach to optimise cohesion and coupling on a lexical level, such as class and parameter names, as well as on a structural level, which results in modularisation solutions that capture more of the developer's perspective. The authors use four metrics: Intra-Module Connectedness Index (IMCI), Between-Module Connectedness Index (BMCI), Module Count Index (MCI), and Module Size Index (MSI) (MCI), where the first two make use of both structural and lexical analysis to determine the connectedness of modules. The results of the work show that combining lexical and structural dependencies yield modularisations that are most useful from a developer perspective compared to using only one of the two.

3.2 Software modularisation approaches

In the past, much research has been done to propose and study approaches for software maintenance, an activity that many developers are familiar with, as studies by Pigoski [59] and Erlikh [28] show that software maintenance takes up over half of all programming effort in large corporations. In this section, various research on software modularisation approaches is discussed. Additionally, an overview of this research can be found in Table 3.1.

Authors [reference]	Problem	Approach	Validated by developers
Mancoridis et al. [46, 47]	Modularisation	Clustering GA	No
Wiggerts [78]	Remodularisation	Clustering	No
Anquetil and Lethbridge [7]	Remodularisation	Clustering	No
Pourasghar et al. [61]	Modularisation	Clustering	No
Maqbool and Babri [49]	Modularisation	Clustering	No
Andritsos and Tzerpos [6]	Modularisation	Clustering	Yes
Naseem et al. [56]	Modularisation	Clustering	Yes
Praditwong et al. [62]	Modularisation	GA	No
Mkaouer et al. [55]	Remodularisation	GA	Yes
Saidani et al. [65]	Microservice extraction	GA	No
Boukharata et al. [12]	Web interface splitting	GA	Yes
Sarkar et al. [67]	Modularisation	Manual	Yes
Teymourian et al. [74]	Modularisation	Clustering	Yes
Levcovitz et al. [44]	Microservice extraction	Manual	No
Prajapati and Chhabra [63]	Module clustering	PSO	No
Schröder et al. [68]	Remodularisation	GA	Yes
Mkaouer et al. [53]	Automated model transformation	GA	No
Di Penta et al. [27]	Library splitting and removing redundant code	GA	Yes
Bavota et al. [9]	Modularisation	GA	No
Hall et al. [33, 34]	Remodularisation	GA	No

Table 3.1: Overview of problems optimised for, approaches used and whether results were discussed with domain experts.

3.2.1 Clustering

Mancoridis et al. [46, 47] created an automated technique called Bunch to create modularisations of C++ software systems. The modules in the source code are clustered to maximise Modularisation Quality (MQ). The authors leveraged the clustering algorithms provided by Bunch, which provides an optimal algorithm for small systems, while for larger systems, users must resort to sub-optimal algorithms that rely on the Neighbouring Partitioning Strategy. The sub-optimal algorithms provided by Bunch are a clustering algorithm based on hill-climbing and a genetic algorithm. Both eventually converge to a local optimum but are not guaranteed to reach the global optimum.

Wiggerts [78] outlined several clustering approaches to remodularise legacy software systems. From the perspective of the system, modularisation means splitting the system into smaller chunks, while from the perspective of a class, it means grouping it with other classes to form related chunks. In the case of legacy systems, which already have a defined structure, completely overhauling the structure can be too costly or complicated. Instead, Wiggerts advocates remodularising legacy systems to reduce their complexity. The work presents an overview of clustering approaches for both perspectives that are suitable for software modularisation and is used by researchers in later clustering-based (legacy) software modularisation research [7, 46, 47, 48, 61].

Pourasghar et al. [61] propose a Graph-based Modularisation Algorithm (GMA) that considers transitive dependencies to modularise software systems. The algorithm clusters classes in the Artifact Dependency Graph (ADG) in modules and updates the assignment of classes to modules repeatedly to optimise the sum of similarity between artifacts (SSA). The output of GMA is the modularisation of the original ADG with the highest SSA. By considering the transitive dependencies in addition to direct dependencies, GMA could produce better modularisations than state-of-the-art search-based algorithms from the perspective of developers.

The architecture of software systems can become lost over time, for example, due to poor documentation or loss of expert knowledge. To regain architectural-level understanding, the source code can be used to reverse engineer the architecture by grouping classes. Maqbool and Babri [48] reviewed hierarchical clustering approaches for software architecture recovery. They applied various approaches to four legacy software systems and found that algorithms that do not retain information related to entities perform more arbitrary decisions, reducing the clustering quality. Additionally, they compared two state-of-the-art approaches, WCA [49] and LIMBO [6], to understand how the number of arbitrary decisions can be reduced. The results show that besides the choice of algorithm, the software system itself largely influences which clustering approach is most optimal.

Naseem et al. [56] propose a novel Consensus-Based Technique (CBT) for software modularisation, called Cooperative Clustering Technique (CCT). The proposed CCT uses an agglomerative hierarchical algorithm in addition to a new feature vector (NFV) algorithm to modularise five software systems, after which the results are compared to clustering approaches that use individual similarity measures to investigate whether CCT is valuable for software modularising. The CCT approach achieved much higher MoJoFM [77] values than single similarity measures, indicating that a cooperative approach can give better modularisation solutions.

3.2.2 Search-based

In addition to clustering, search-based approaches have been prevalent for software modularisation. A study by Praditwong et al. [62] showed that search-based approaches outperform traditional clustering approaches for software modularisation. Praditwong et al. used two novel search-based approaches to find software module clusters: Maximising Cluster Approach (MCA) and Equal-Size Cluster Approach (ECA). Both approaches are compared to a traditional Hill-Climbing (HC) algorithm that optimises Modularisation Quality (MQ). Using a multi-objective search-based approach has the advantage that developers need not concern themselves with the optimal weights for the best calculation of MQ. This is important because it is often impossible to determine exactly how much gain in coupling is needed to justify the loss in cohesion or vice versa. Instead, a multi-objective approach returns a set of non-dominated results that account for every objective function. Developers can choose from this set the solution that best fits their needs in terms of cohesion and coupling. Both multi-objective approaches were compared to HC, and it was found that they outperformed HC both in terms of the traditional MQ value and the multi-objective values. Most importantly, Praditwong et al. have shown that multi-objective formulations show significant promise in solving the modularisation problem.

Search-based software modularisation approaches can optimise multiple objective functions to create modularisations, but research has shown that these approaches do not scale well for high-dimensional objective spaces [42, 76]. Deb and Jain proposed an evolutionary algorithm inspired by the NSGA-II framework [25] to handle high-dimensional problems efficiently, called NSGA-III [23]. NSGA-III makes use of non-dominated sorting, similar to NSGA-II. The problem with a higher number of objectives is that the proportion of non-dominated solutions increases exponentially. This causes the performance of NSGA-II algorithms to degrade, as the diversity of the population can no longer be maintained. NSGA-III can maintain a diverse population, even when dealing with many-objective problems, by using a set of reference points throughout the objective space. Each generation, every solution in the population is associated with its closest reference point, giving each reference point a niche count equal to the number of associated solutions. When performing selection to select surviving solutions, those associated with lower niche count reference points are favoured over those associated with higher niche count reference points. In doing so, the NSGA-III algorithm prioritises solutions that improve the diversity of the population while also favouring non-dominating solutions. While this comes at the cost of additional computational complexity, NSGA-III has been shown to outperform NSGA-II and other search-based approaches for many-objective problems [23, 40, 54].

Using an NSGA-III algorithm, Mkaouer et al. [55] created a modularisation approach that optimised for seven objectives, among which are structural quality metrics, semantic coherence and change history consistency while minimising the number of changes. Using the number of changes as an objective ensures that the NSGA-III algorithm does not propose "big-bang" modularisations whose vastly different design would be difficult to understand for developers, but it can offer modularisations with various balances between size and quality. The algorithm was able to fix 92% of code smells in several open-source projects and one industrial project, showing that NSGA-III can be applied to many-objective software remodularisation problems.

Saidani et al. [65] formulated the microservices extraction problem as a two-objective optimisation problem, applying an NSGA-II algorithm to extract microservices automatically. The NSGA-II algorithm, called MSExtractor, minimised inter-service dependencies (coupling) and maximised intra-services dependencies (cohesion) to identify and extract microservices from legacy applications. The performance of MSExtractor was compared to three state-of-the-art approaches for four commonly used quality measures for web service interfaces. MSExtractor was found to outperform them all at identifying microservice candidates.

Third-party web service interfaces are usually the only part of the system visible to outside users. As such interfaces are the main interaction point for users of the underlying system, they should be kept comprehensible, but as maintenance and changes are applied to the interface, their complexity increases, which negatively affects its usability and structure. Boukharata et al. [12] proposed a multi-objective approach, called WSIRem, to split web service interfaces into smaller interfaces that are more cohesive and loosely coupled. In addition to maximising cohesion and minimising coupling, WSIRem also minimises the modification degree, meaning the required modularisation effort. The results of WSIRem applied on a benchmark of 22 services highlight its ability to extract smaller and more cohesive web interfaces from large ones.

3.2.3 Remodularising at scale

In most of the related work discussed up to now, the authors have provided empirical evidence for the performance of their proposed tools and approaches. To allow reproducibility of their work, researchers often gather empirical evidence on open-source software systems. However, these open-source software systems are rarely of the same order of size as enterprise software systems, thus raising the question of whether the proposed tools and approaches scale to such sizes. In the remainder of this section, we will discuss research focused on modularising enterprise-level large-scale software systems.

Sarkar et al. [66] criticise software modularisation research for barely distinguishing between a class and a module, considering the two as synonymous concepts. They claim that as software systems grow in size, the level of granularity of classes is too low to be an effective unit for the purpose of software modularisation, as modules in large software systems can consist of hundreds of classes. Their work proposes 12 metrics to assess the quality of Application Programming Interfaces (APIs) of large-scale software systems. These metrics operate at module level granularity and allow for a higher-level understanding of the quality of the modularisation in comparison to class level granularity metric suites like MOOD [1]. This work is followed up by the modularisation of a banking system [67] consisting of 25 million LOC, using their 12 metrics to monitor the quality of the modularisation. The banking system, which was over a decade old, did not have any clear domain boundaries or architectural structure, which significantly increased development time and maintenance effort. The authors spent over two years modularising half of the system, requiring expert knowledge to determine the domain boundaries of modules and refactoring the code to support the new modularisation, with much of the effort being done manually.

Teymourian et al. [74] proposed a fast clustering algorithm (FCA) for large-scale software systems, showing its performance on ITK[2] and Chromium[3], which contain thousands of files and consist of 1 million and 10 million lines of code, respectively. The number of operations needed to calculate similarity scales quadratically with the number of files for typical clustering algorithms like LIMBO [6], which would be slowed down significantly by large software systems. To cluster these large systems, FCA derives additional matrices from the dependency matrix based on a set of features. These additional matrices allow FCA to produce modularisations of large-scale software systems in a timely manner. The results of FCA on ITK and Chromium were deemed acceptable by experts of the respective system, proving that FCA can scale to cluster large software systems efficiently.

[2]`https://itk.org/`

[3]`https://www.chromium.org/chromium-projects/`

Levcovitz et al. [44] proposed a technique to identify microservice candidates in monolithic enterprise systems. First, a dependency graph of the system is created, with facades, functions and database tables as nodes and calls as edges. The dependency graph is split into subsystems which are candidates to become microservices. Through manual inspection of the code and business logic, candidates are judged whether they will become microservices. The authors applied this technique to a banking software system containing 750 thousand LOC. They found several candidates that could be extracted to become a microservice, thereby concluding their study a success. However, the authors did not declare themselves experts on the software system, nor did they consult actual developers of the system to evaluate their results.

Prajapati and Chhabra [63] propose a particle swarm optimisation (PSO) algorithm for large scale software clustering, called PSOMC. PSOMC optimises four objectives to improve the clustering of software systems: intracluster dependency, intercluster dependency, the number of clusters and the number of modules per cluster. PSOs are a different type of evolutionary algorithm compared to genetic algorithms. The solutions in a PSO are called particles, and the population of particles is called a swarm. Where the solutions in the population of a GA are independent of each other, except for crossover, the state of the swarm in a PSO affects each particle. Like GAs, PSOs are search-based algorithms that reiterate over many generations to optimise for a given (multi-objective) problem. PSOMC uses a weighted combination of the four objectives to result in a single fitness function. The performance of PSOMC is evaluated against six real-world software systems and compared to three search algorithms: group genetic algorithm, hill climbing and simulated annealing. The resulting clusterings of all algorithms are assessed on MQ, coupling, cohesion and non-extreme distribution (NED). NED is a metric to measure the distribution of modules that are too small or too large in software systems. The results of PSOMC compared to the other algorithms show that PSOMC is able to generate reasonable clustering solutions.

Schröder et al. [68] used an NSGA-II algorithm to remodularise a software system consisting of several millions of lines of code by moving classes between modules. This algorithm and its objective functions have already been extensively discussed in Section 2.4. Their work showed that enterprise-level software systems could be remodularised using a multi-objective search-based approach. An empirical study shows that their proposed algorithm can significantly improve the code structure quality of the codebase, showing promise that remodularisations of enterprise-level systems can be generated automatically. To evaluate whether the changes suggested by their algorithm were helpful from the perspective of developers, the authors interviewed developers on their perception of the quality of the suggested changes. The results from these interviews showed that the algorithm can successfully identify problematic classes in modules but is inaccurate in suggesting the suitable module for classes. These findings highlight the necessity for software modularisation research to validate their results with developers that have expert knowledge of the research software systems and show that code quality metrics alone are not always sufficient to generate modularisations that developers accept.

3.2.4 Incorporating developer knowledge

The Pareto front of a multi-objective problem can consist of many solutions, well-distributed in the objective space. In terms of the objectives, none of the solutions dominates the others, but in many cases, the decision-maker only needs one solution. Solutions in the Pareto front that are far from the decision-maker's preferred region are less likely to be chosen than those located near or even in the preferred region. The decision-maker could look at all solutions resulting from a multi-objective algorithm and choose one. However, considering that this set may be enormous, it drastically increases the workload on the decision-maker. This section describes various pieces of literature on how developer knowledge can be used to improve modularisation suggestions.

Mkaouer et al. [53] proposed a reference solution-based NSGA-II (r-NSGA-II) algorithm for automated model transformation. The decision-maker can define multiple reference points before starting the algorithm, which indicate preferred areas in the objective space. Through these defined reference points, the algorithm can prioritise specific solutions in a population of non-dominating solutions. Compared to the performance of a standard NSGA-II algorithm, r-NSGA-II produces many more solutions at or near the reference points, showing promise for the application of reference-based evolutionary algorithms to incorporate user preferences for automated model transformation.

Gong et al. [32] implemented an interactive evolutionary algorithm for interval multi-objective optimisation problems that considers decision-maker a priori preferences to determine the preferred region. These preferences steer the algorithm coarsely initially, but as more generations have passed, the preferences steer the algorithm more finely. A decision-maker can state that they prefer one objective over another, that they are equal in preference or that they are incomparable without needing the decision-maker to determine weights for these relations. This results in a preference region that allows the algorithm to distinguish between solutions with identical ranks. The algorithm favours solutions in the preferred region rather than aiming at a uniformly distributed Pareto front. Similarly, Di Penta et al. [27] proposed a Software Renovation Framework (SRF) that considers decision-maker preferences for automated software renovation. SRF uses an interactive genetic algorithm (IGA) to identify duplicate or unused code and find candidate libraries to split into smaller ones, which a decision-maker can then give feedback on. This feedback is used in the algorithm's fitness function, allowing the decision-maker to provide feedback to steer the algorithm interactively. This feedback is used in the fitness function of the algorithm, allowing the decision-maker to interactively provide feedback to steer the algorithm. The authors applied SRF on an actual business application and used the feedback of developers of that application as input for the algorithm. As a result, the developers identified and confirmed various structural problems, after which these problems were solved. These results show the value of including the perspective of developers for automated software refactoring.

Bavota et al. [9] proposed an IGA that allows a decision-maker to steer the algorithms modularisation approach. This is done by partially evaluating the fitness automatically, such as code quality metrics, and partially by a human decision-maker. The decision-maker can penalise a solution that contains software components in the wrong place, thus conflicting with their domain knowledge (e.g. a database class outside of the database module).

Bavota et al. proposed both single-objective as well as multi-objective GAs inspired by the MCA algorithm from Praditwong et al. [62], comparing the performance against a non-interactive version of the MCA algorithm on two software systems. For the interactive algorithms, decision-maker feedback was requested on the best solution found so far, where the decision-maker was presented with pairs of classes from that solution and could provide two constraints; either the pair belongs to the same module, or they should be kept apart. After each feedback round, that solution was used to create a new population for the algorithm to continue. There are two main drawbacks to this approach. Firstly, a decision-maker must specify constraints for pairs, while they might not have an opinion on the relationship of each pair, at least at that moment. Secondly, by creating a new population based on mutations of the best solution, much of the variety of the population coming from other solutions is lost. For every algorithm, the solution with the highest MQ value was selected as the best solution. The authors report that with a limited amount of feedback, the IGAs are better able to propose modularisations that are useful from the perspective of developers than traditional GAs. However, it is worth noting that no actual developers were included in the study to provide feedback, and instead, authoritative modularisations were used to simulate developer feedback.

Hall et al. [33, 34] proposed their supervised software modularisation (SUMO) algorithm, which remodularises a software system consistent with constraints specified by a decision-maker. SUMO starts out unsolved and proposes a modularisation of a target software system. If a decision-maker is not satisfied with the current modularisation, they can indicate for one or more pairs of classes whether the pair belongs together. SUMO then produces a new modularisation consistent with these constraints or returns an error message if some constraints conflict. The modularisation is solved once the decision-maker no longer declares new constraints. The authors validated SUMO on several small software systems and have shown that the feedback increases the quality of the suggested modularisations. However, they simulated the feedback using an authoritative modularisation rather than involving actual developers.

3.3 What is missing?

As can be seen from the literature discussed in this chapter, much work has been done in the past to tackle the modularisation problem, proposing various approaches that have been applied to software systems to study their potential benefits. However, as can be seen from Table 3.1, the potential of many of these approaches is only evaluated empirically and these approaches do not consider if the results are useful in real-life applications. Over half of the work in Table 3.1 did not validate their results with developers, which is in line with the work of Praditwong et al. [62], which found literature on software modularisation lacking when it comes to evaluating suggestions from the perspective of developers.

In a similar sense, many of the search-based approaches discussed do not consider the perspective of developers during decision-making and instead only consider code quality metrics to determine whether solutions are good. Including this perspective is a complex challenge. However, the resulting solutions would be more in line with the expectations of developers, as can be seen in the literature on supervised and interactive software modularisation in Section 3.2.4. It is worth noting that in that literature, developers were rarely included and instead, their feedback was simulated using authoritative modularisations.

Lastly, many of the software systems used for research are not large-scale, raising the question of how well the results carry over to enterprise-level systems and if the approach performs sufficiently. The research that evaluated on large-scale systems, such as Schröder et al. [68], did not consider the perspective of developers during decision-making. The exception to this statement is the work of Di Penta et al. [27], which reduced code duplication and split large libraries into smaller, more cohesive ones. Understanding whether modularisation approaches scale to larger software systems is essential to determine whether they can be applied for real-life business applications, which can be orders of magnitude larger than the open-source systems that are often used for research purposes.

In this thesis, we propose a search-based remodularisation approach that considers the domain knowledge of developers to produce remodularisations of a large-scale codebase. These modularisations will be validated by the developers to determine whether the suggestions are good or if more domain knowledge is needed for improvement. To reduce the workload of developers, feedback should be of various granularity and be reusable. Stating whether pairs of classes should be together or not, such as the approach by Bavota et al. [9], is reusable but not scalable. Indicating the number of differences between a suggested modularisation and the refactoring proposed by developers, such as the approach by Di Penta et al. [27], is scalable but not reusable. For the proposed approach in this thesis, developers can provide fine-grained feedback at class level as well as coarse-grained feedback at module level or module-type level. The work in this thesis focuses on the following aspects:

- Create remodularisation solutions in line with developer knowledge.
- Make developer knowledge reusable.
- Make stating developer knowledge scalable to reduce the workload of developers.
- Evaluate remodularisation solutions with developers.
- Make this domain-aware remodularisation approach scalable to large-scale software systems.

Chapter 4

A domain-aware search-based algorithm for software remodularisation

Every codebase has its own specific rules that developers know the code must adhere to. Some were thought of in the initial design phase, and others later as the codebase grew in size and complexity. For example, for a web application, developers may have decided that code in front-end modules may not directly interact with database modules, only with back-end modules. With this knowledge in mind, developers can identify illegal dependencies and resolve them by either refactoring or rewriting parts of the code, especially when these rules are known beforehand and regularly enforced.

However, when such domain rules are not enforced or defined later in the life of a codebase, once it has grown to considerable size and complexity, the effort required to resolve the codebase to a state where all rules are adhered to is considerable. Furthermore, large refactoring projects are often challenging to perform in one go, especially if normal development happens in parallel. Manually determining all cases that violate a domain rule, finding the right solution to resolve it, and performing the fix is time-consuming and error-prone. On the other hand, using fully automated refactoring tools results in long lists of refactorings that often lead to a significantly different and hard to understand design of the codebase [3]. **To provide the best of both strategies, we propose an algorithm that remodularises a codebase, improving its quality while adhering to the domain knowledge of developers.**

The idea behind the algorithm is to present a set of move-class refactoring suggestions that provide improvements compared to the current state of a codebase, while adhering to the domain knowledge of the developers. Instead of presenting an utterly different modularisation consisting of hundreds or thousands of changes, the algorithm presents the developers with a small set of changes. This is because the effect of a small set of changes is more easily understood, accepted and implemented by developers. In contrast, completely overhauling the codebase structure is likely not feasible to be done in a reasonable amount of time, may take the effort of many different developers and is more likely to introduce errors as part of the large operation. By gradually improving the codebase, developers understand

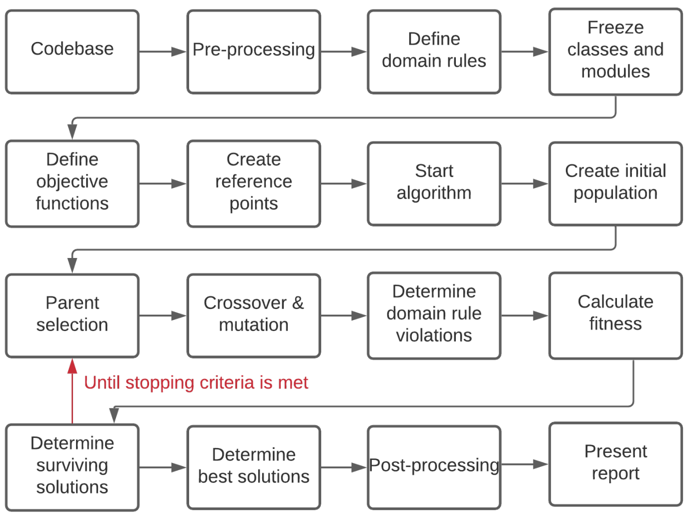


Figure 4.1: Algorithm pipeline. Starts at the codebase and ends after presenting the report.

the steps they take from start to end, giving them confidence that their work improves the codebase and ensures that they do not spend too much time refactoring. It is essential that the algorithm complies with the domain knowledge that the developers have so that the suggested changes are acceptable and usable.

This thesis proposes an NSGA-III algorithm to modularise software systems by suggesting move-class refactoring operations. Moving classes between modules can improve the code quality at module level, for example, by increasing the cohesion of modules or reducing their coupling. Additionally, moving classes can reduce the number of violated domain rules on class and module level. The algorithm does not perform refactorings on code level granularity, as this would exponentially increase the size of the search space, which makes it infeasible to converge for enterprise-level codebases.

Figure 4.1 illustrates the step-by-step workflow of the algorithm, starting at the codebase. In the rest of this section, we will take a closer look at these steps and how the algorithm operates.

4.1 Modelling solutions

To allow the algorithm to perform move-class refactorings on the targeted codebase, it must represent modularisations of the codebase as solutions that are members of its population to perform crossover and mutation on them. Traditionally, solutions are represented as fixed-length vectors. In the case of modularisation, the length of a solution could be the number of classes, with the value of each cell signifying in which module it is located. This causes issues when working with traditional crossover methods such as single point, two-point and uniform crossover [36] for modularisation, because these operators consider classes rather than modules, thus making it likely to split modules into separate parts when creating offspring solutions. For this reason, the building block preserving crossover operator mentioned in Section 2.4.1 is used to allow the algorithm to perform crossover on solutions while respecting their module structure. Additionally, solutions are modelled as change-list compared to the original modularisation for performance reasons, similar to the approach by Schröder et al. [68].

As an example, Figure 4.2 shows a dependency graph of some codebase and Figure 4.3 shows a remodularisation solution of that codebase. The solution comprises a single change, moving class *Authenticator* to module *bank*. Therefore the corresponding solution would consist of a single mapping of the class *Authenticator* to its new module location *bank*.

4.2 Pre-processing the codebase

In order for the algorithm to generate remodularisation solutions, it must understand the structure of the codebase in the form of a directed dependency graph. In this graph, the classes are represented as nodes, while dependencies as represented as directed edges. Before the algorithm can run, this dependency graph is created using the following information for each class:

- The fully qualified domain name of the class.
- The name of the module where the class is currently located in.
- The list of classes the class depends on.

This directed dependency graph is then stored and can be used as input for the algorithm.

4.3 Defining domain knowledge rules

In order for the algorithm to generate solutions that do not violate the domain knowledge of developers, the developers need to provide the algorithm with the domain rules they want to have enforced. To achieve this, a domain-specific language (DSL) is proposed that serves two purposes. First, the DSL allows developers to state domain rules. These domain rules can vary in granularity, targeting specific classes or modules, or covering entire module types. Second, developers can configure which parts of the codebase the algorithm should focus on by freezing classes or modules that should not be affected.

4.3.1 Domain rule approach

As developers can have domain rules that apply to specific parts of the codebase, the DSL should allow them to specify which classes and modules should adhere to specific rules. To do so, developers can define domain rules specific to certain parts of the codebase. For example, taking the codebase represented by Figure 4.2, a domain rule could be that classes in module *authentication* cannot depend on classes in module *transaction*. The illegal dependency from *Authenticator* on *Transaction* is highlighted in red in Figure 4.4. In terms of the DSL, a developer would define a domain rule for all classes in module *authentication*

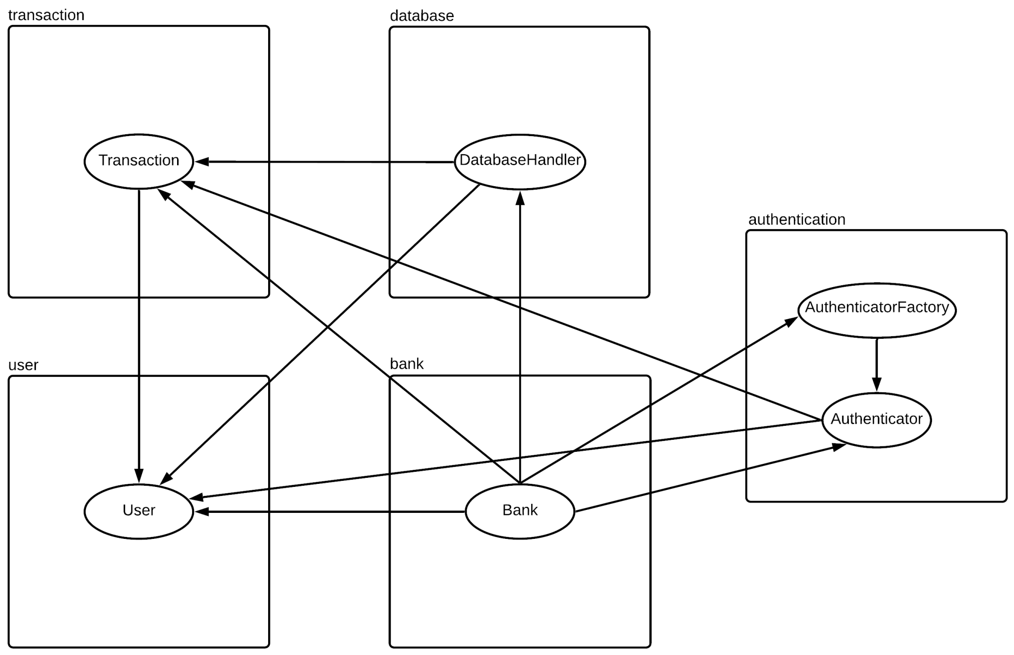


Figure 4.2: Dependency graph of example codebase

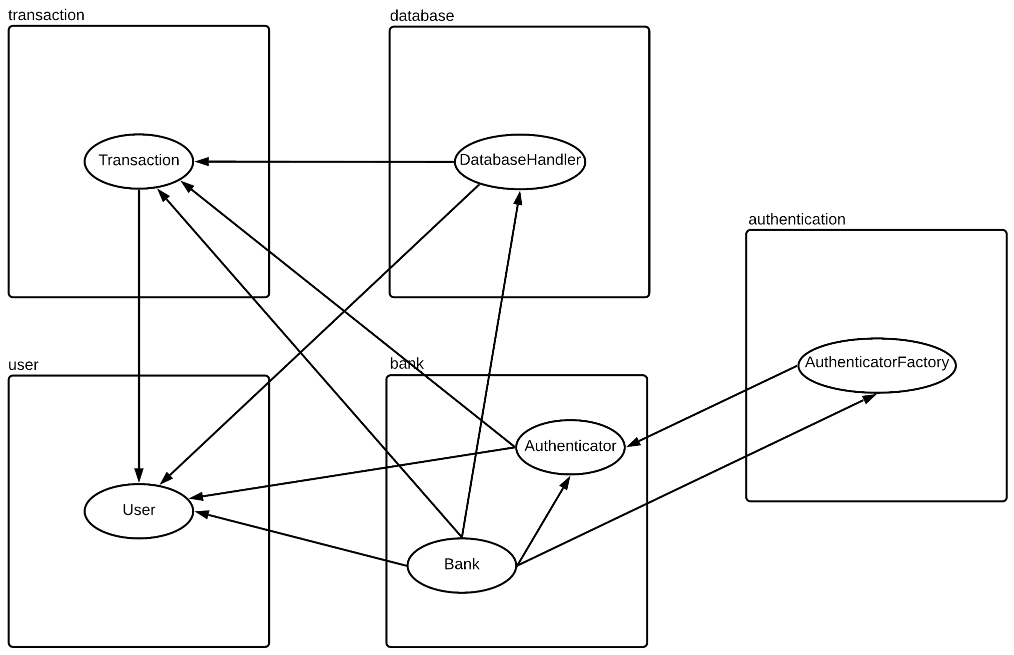


Figure 4.3: Dependency graph of example solution

that forbids dependencies on module *transaction* for this example. Developers can vary the granularity of domain rules, ranging from targeting a specific class to several modules at once.

While the goal of the algorithm is to generate modularisations that are in line with the domain knowledge of developers, some solutions may introduce new violations instead of resolving them. There are several potential strategies to deal with this behaviour:

1. Do not allow the algorithm to create solutions that violate domain rules.
2. Allow the algorithm to generate violating solutions up to some number of generations, after which these solutions must be repaired or discarded, so the resulting final population does not contain violating solutions.
3. Allow the algorithm to generate violating solutions and penalise solutions based on the severity of their violations, reducing the likelihood of violating solutions surviving through generations.

Strategy 1 is the simplest to implement. Whenever the algorithm attempts to move a class to a module, it must check whether that would introduce new violations. If it would, the move cannot happen. The downside of this strategy is that it can severely hinder the exploration potential of the algorithm, depending on the number of specified rules. Furthermore, due to the way the mutation operator works, to achieve a modularisation that does not violate domain rules and improves the original codebase, an illegal intermediate modularisation may be needed to reach it.

Strategy 2 is more nuanced than strategy 1 as it does not hinder the exploration of the algorithm during the generations where violations are accepted but can result in poor results afterwards. In order to repair solutions that violate domain rules, an effort must be made to move classes such that they no longer cause violations. In the worst case, this can mean that all changes in a solution must be discarded, resulting in a solution that proposes the original codebase or such a large number of changes are needed that developers are hesitant to accept the resulting solution. Additionally, the amount of time needed to repair solutions can significantly influence the time needed for the algorithm to reach convergence.

Strategy 3 gives the algorithm the most flexibility to try various modularisations. The algorithm can freely explore the objective space by allowing solutions to introduce new violations. Furthermore, by penalising solutions based on their violations, the algorithm prioritises solutions with few or no violations over solutions with a large amount, which improves the probability of the former surviving through generations. This means that while the algorithm is not hindered in exploring the objective space, it will attempt to generate solutions that violate as few domain rules as possible. As a result, strategy 3 is used to deal with the domain rule violations of solutions.

By default, every rule violation is considered equally bad, with a penalty score of 1. However, developers may prioritise some rules over others, so to accommodate for this, the penalty for each domain rule can be configured to a different amount. Setting different penalties allows the algorithm to understand the priority developers have assigned to the domain rules. In line with strategy 3, the *total domain rule violation penalty score* of a solution is used to determine which solutions outperform others and, therefore, should have a higher probability to survive to the next generation.

For genetic algorithms, their population members are ranked on their fitness. A single value or multiple values can express this fitness. For example, the software modularisation algorithm proposed by Schröder et al. [68] used four objective functions to determine the fitness of solutions: *Inter-Module Dependency Coupling (Coupling)*, *Intra-Module Dependency Coupling (Cohesion)*, *Number of class moves (#moves)* and *Estimated Build Cost of Cache Breaks (EBCCB)*. When posing the problem as a minimisation problem, one solution Sol_A dominates another Sol_B if for every objective function $F : F(Sol_A) \leq F(Sol_B)$ and there exists at least one objective function such that $F(Sol_A) < F(Sol_B)$. In other words, Sol_A dominates Sol_B if Sol_A outperforms Sol_B in at least one objective function and performs at least as well in all others. Like the algorithm of Schröder et al., our NSGA-III in this thesis is a Multi-Objective Genetic Algorithm (MOGA), with the objective functions defined in Section 4.4.

To account for its domain rule penalty in the survivability of a solution, there are two potential strategies to let the penalty affect the fitness of a solution:

1. Add the domain rule penalty as an objective function and rank solutions in a multi-objective manner.

2. Normalise and sum the fitness values and add the domain rule penalty to that amount, using the single new value to rank solutions.

Strategy one increases the number of dimensions of the objective space, which gives the algorithm the ability to optimise for other objective values at the cost of increasing the domain rule penalty or vice versa. This means that the Pareto front will include solutions both with high and low domain rule penalty values. Additionally, it will increase the number of reference points used by the algorithm, as will be discussed in Section 4.5, which affects the population size and thereby increases the time needed to reach convergence.

Strategy two requires optimising the ratio between the normalised and summed fitness values and a solution's domain rule penalty score. For example, suppose the domain rule penalty score considerably outweighs the fitness values. In that case, solutions with good fitness values but a high domain rule penalty are dominated by solutions with terrible fitness values but a low domain rule penalty score. A similar argument can be made for the case where the normalised and summed fitness values outweigh the domain rule penalty score. To solve these scenarios requires tuning the assigned weights for the penalty score and the summed fitness values, which are likely specific to a single codebase and therefore do not generalise.

While both strategies have their downsides, we prefer strategy one because it does not require additional weight tuning but rather leaves it to the algorithm to trade off by using it as an objective function. The rules are composed of three components in the DSL: a filter, a rule and a penalty. The filter expresses against which classes or modules the rule must be evaluated. The rule itself expresses some condition that the classes or modules must adhere

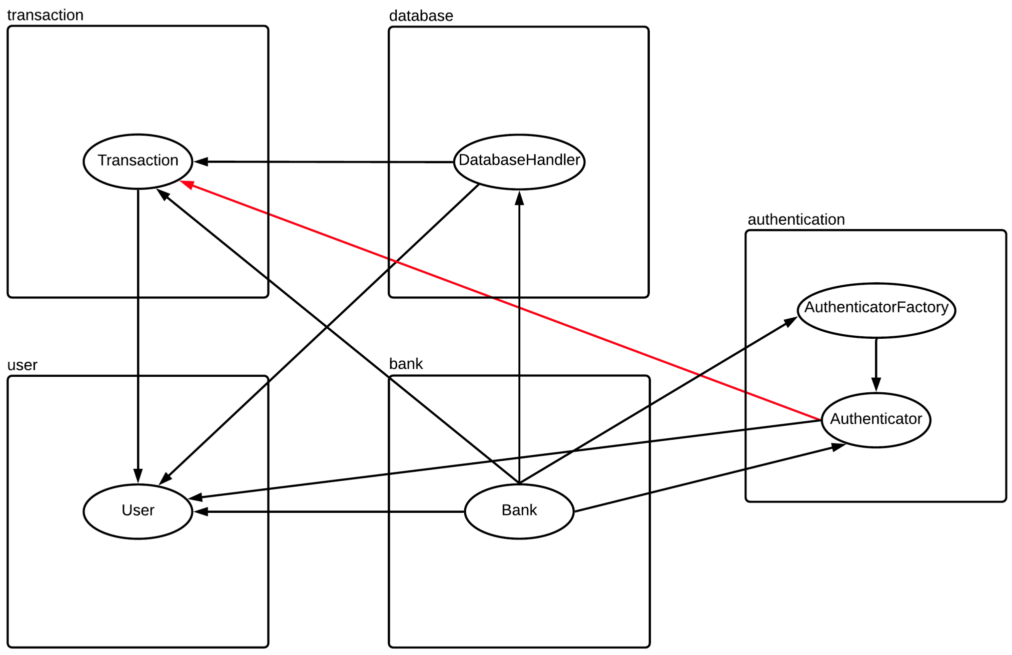


Figure 4.4: Dependency graph with highlighted illegal dependency

to, such as forbidding a dependency. Lastly, the penalty indicates the relative priority from the developers' perspective. Using the example domain rule stated above, the filter would be *classes in module authentication*, the rule would be *cannot depend on classes in module transaction*, and the penalty would be 1 by default.

4.3.2 Freezing the codebase

As codebases grow in size and complexity, it becomes increasingly more difficult for any developer to understand every aspect of the codebase. Developers will often focus on specific areas, becoming more knowledgeable in some parts than others. Presenting a developer with refactoring suggestions on code that they are unfamiliar with requires more effort from the developer to understand and have confidence in these suggestions than suggestions made for parts of the codebase they are familiar with. The algorithm of Schröder et al. operated on an entire codebase, which resulted in refactoring suggestions on all parts of the codebase. This required a post-processing filtering of the solutions to present developers only with move-class refactoring suggestions for the code they often work with.

Providing developers with the ability to generate suggestions on the areas of the codebase they are most familiar with requires that the algorithm can be configured to focus on those areas. This configuration is done by freezing modules or classes in the original state of the codebase. The algorithm can be configured for different ways of freezing:

- Freezing a class - Frozen classes cannot be moved.

- Inwardly freezing a module - Inwardly frozen modules can not have classes added to them that are not originally in the module, but the classes in the module can be moved out.

- Outwardly freezing a module - Outwardly frozen modules can have new classes added to and moved out of them, but the classes that are originally in it cannot be moved out. Essentially, only the classes originally in the module are frozen.

- Totally freezing a module - Totally frozen modules cannot have any classes moved in or out of them.

For example, consider the dependency graph from Figure 4.2 again. Say that a particular developer for this codebase primarily works on code in all modules except *authentication*. Additionally, the developer believes that the class *User* should not be moved out of the *user* module. Therefore, he configures the algorithm by freezing the class *User* and totally freezing the module *authentication*. This is illustrated in Figure 4.5 with frozen classes and inward frozen modules outlined in blue. In this example, the classes *Transaction*, *Database-Handler* and *Bank* can still be moved, and the modules *transaction*, *database*, *user* and *bank* accept new classes.

With the types of freezes available, a developer can configure the algorithm such that the search space can range from modularising the entire codebase to just a few classes or modules. While the algorithm can move classes even if those moves introduce new domain

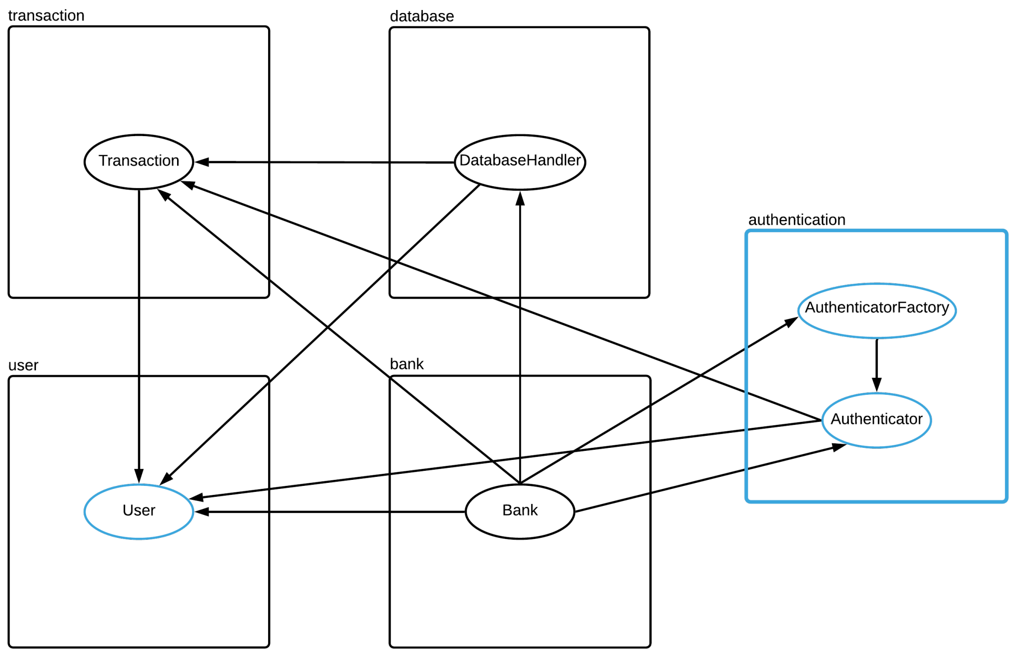


Figure 4.5: Dependency graph with frozen classes and modules

rule violations, it is not possible to move classes that are frozen nor move classes to modules that are inwardly frozen. Consequently, by freezing a codebase such that only a handful of classes or modules are not frozen, the size of the search space is orders of magnitude smaller compared to when nothing is frozen, thereby significantly reducing the number of possible solutions in the objective space.

4.4 Objective functions

Objective functions are used to determine the fitness values of solutions. These fitness values are used to rank the solutions in fronts, based on the solutions they dominate and are dominated by. Every objective function corresponds to a dimension in the search space, which the algorithm will try to optimise for. A developer using the algorithm can choose objective functions to their liking. For example, some objective functions involve measuring the code quality of a codebase, while others ensure the number of refactorings suggested by a solution does not grow out of hand. The following objective functions are inspired by the thesis of Schröder et al.: *Change impact*, *Inter-Module Dependency Coupling (Coupling)* and *Intra-Module Dependency Coupling (Cohesion)*, where *Change impact* is identical to the notion of *work* in the work of Schröder et al. In addition to those, three objective functions have been formulated to minimise the number of circular module dependencies, the domain rule violation penalty and the size of the largest module. All objective functions are formulated such that the algorithm will attempt to minimise the corresponding fitness values.

A brief overview of the objective functions is listed below, and a more detailed description can be found in the rest of this section:

- **Change impact -** Represents the number of classes moved in a solution.
- **Circular module dependencies -** Estimates the number of cycles in the module dependency graph.
- **Inter-Module Dependency Coupling (InterMD Coupling) -** Represents the total coupling in the codebase, expressed as the sum of module dependencies for all modules.
- **Intra-Module Dependency Coupling (IntraMD Cohesion) -** Represents the total cohesion in the codebase, expressed by the sum of module cohesion for all modules.
- **Domain rule violations -** Represents the total number of domain rule violations for a solution.
- **Largest module -** Represents the size of the largest module in the codebase, expressed as the number of classes in that module.

4.4.1 Change impact

A solution's *change impact* is defined as the number of changes a solution suggests. If the algorithm does not minimise for the change impact, this number of changes could be equal to the number of classes in the codebase in the worst case. To deal with this, the algorithm must optimise solutions such that the number of changes does not grow out of hand. Doing so helps ensure that developers receive a list of suggestions that is manageable without taking days or weeks to actualise.

4.4.2 Circular module dependencies

A circular module dependency occurs when a module depends on other modules such that transitively, the original module depends on itself. This holds true for all modules in the circular dependency. Having circular module dependencies can cause code to become unbuildable; therefore, the solutions created by the algorithm should not contain any circular module dependencies. Figure 4.6 shows a dependency graph with a circular module dependency. Class *Authenticator* has been moved from *authentication* to *bank*. However, class *Bank* depends on class *AuthenticatorFactory*, meaning module *bank* depends on module *authentication*. Additionally, class *AuthenticatorFactory* depends on class *Authenticator*, meaning module *authentication* depends on module *bank*. As a result, modules *bank* and *authentication* form a circular dependency. The edges causing the cycle are coloured red in the figure.

Since all solutions with a circular dependency are unwanted, the easiest way to optimise for solutions without circular dependencies is to check if the module dependency graph of a solution contains any cycle, returning 1 as fitness and 0 otherwise. The downside of such an approach is that a mutation must resolve all cycles in the dependency graph to be considered an improvement in terms of circular dependency. If the number of cycles is decreased but still above 0, the fitness of that solution would remain 1.

Instead, we have chosen the more expensive approach of estimating the number of cycles in the dependency graph. This encourages the algorithm to mutate solutions with circular dependencies such that the number of circular dependencies decreases and discourages mutations that increase the number of circular dependencies. Solutions with circular dependencies can never dominate solutions without circular dependencies, so over many generations, it can happen that the solutions with circular dependencies did not survive or were mutated such that they no longer contain circular dependencies. In both cases, solutions without circular dependencies are favoured by the algorithm.

To calculate the number of cycles in a dependency graph, a depth-first-search (DFS) algorithm is used, inspired by Cormen et al. [19]. A brief explanation is listed here:

Each node is originally coloured white. While the edges of a node are explored, that node is coloured grey. After all its edges are explored, the node is coloured black. The DFS algorithm starts at a random node r in the graph, colouring it grey and exploring its edges if the node at the end of the edge is coloured white. Any time the DFS algorithm encounters a grey node, a cycle has been detected, as this grey node has been visited earlier while exploring the edges of node r. Once the DFS algorithm finishes exploring the edges of node *r*, another white node in the graph is explored until all nodes in the graph have been explored. After the DFS algorithm finishes, the number of found cycles is returned by the objective function.

Because modules that have already been visited are not explored by the DFS algorithm anymore, the number of cycles found may be lower than the total number of cycles in the dependency graph. Figure 4.7 shows an example of such a graph. The graph shows various cycles, such as $[2,3]$ and $[3,4]$. The number of found cycles varies depending on which node the DFS algorithm starts. For example, let the algorithm start at node 3, go to node 2 and then back to 3, discovering a cycle. At this point, node 2 is coloured black since all its edges are explored. The algorithm then traverses the edge from 3 to 4 and back to 3, discovering a cycle. The algorithm then traverses the edge from 4 to 1, after which it cannot go to 2 because it is already coloured black. The DFS algorithm has found two cycles.

There are however a total of 4 cycles: $[1,2,3,4]$, $[2,3], [3,4], [2,3,4]$. The total number of cycles in a graph can be calculated by algorithms such as Tiernan [75], Tarjan [73] and Johnson [41], but these algorithms are slower than the DFS algorithm mentioned above. During preliminary experiments, the algorithm by Johnson, which is the fastest of the three that calculate all cycles, was often at least an order of magnitude slower than the DFS algorithm for the Adyen codebase, significantly slowing down the NSGA-III algorithm. As a result, the DFS algorithm is used as it indicates the number of cycles in a solution, prioritising speed over accuracy. Ultimately, this objective function aims to allow the algorithm to create solutions without circular dependencies, which all the algorithms mentioned above facilitate.

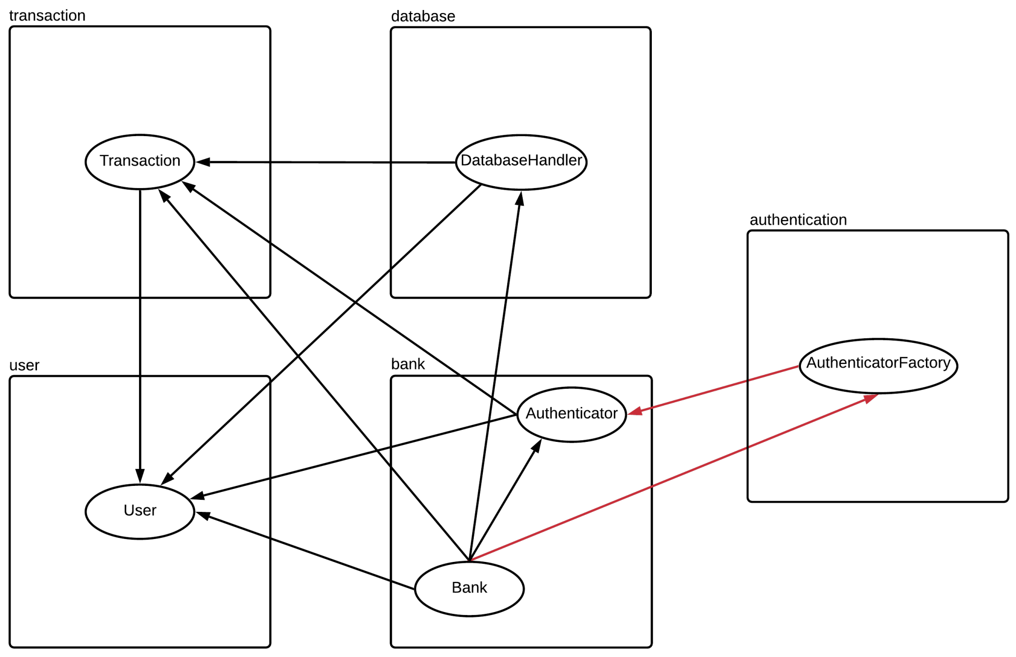


Figure 4.6: Dependency graph with circular module dependency

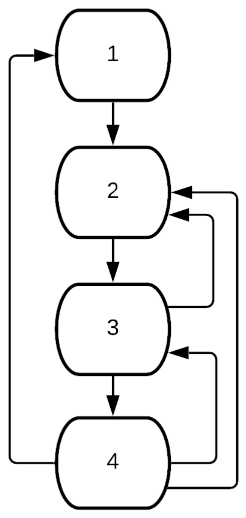


Figure 4.7: Directed cyclic graph

4.4.3 Inter-Module Dependency Coupling (Coupling)

At module level, coupling represents the degree to which a module depends on other modules. Reducing coupling in a codebase means that modules become less dependent since they rely less on other modules to function. There are various levels of granularity to determine coupling, such as class level and module level [13]. This objective function calculates module-level coupling since our approach moves classes between modules.

Inter-Module Dependency (InterMD) Coupling is used to measure the coupling of a solution, the same metric used by Schröder et al. [68] for the Adyen codebase. From the results of their work, InterMD Coupling showed promising results as a metric for improving coupling in resulting solutions. The coupling is defined as the total number of module dependencies of a codebase.

For example, the module dependencies of the codebase represented by the dependency graph in Figure 4.2 are shown in Table 4.1. For this codebase, the total number of module dependencies, and thus the InterMD Coupling of the codebase, is 9. The codebase resulting from moving *Authenticator* from *authentication* to *bank* shown in Figure 4.3 removes the dependencies from *bank* on *authentication* and from *authentication* on *transaction* and *user*, and adds a dependency from *authentication* on *bank*. The module dependencies of this solution are shown in Table 4.2, showing that the total InterMD Coupling is 7. This means that in terms of InterMD Coupling, moving class *Authenticator* to module *bank* improved the quality of the codebase.

Module name	Module dependencies
transaction	user
database	transaction, user
user	-
bank	transaction, user, database, authentication
authentication	transaction, user

Table 4.1: Example codebase module dependencies

Module name	Module dependencies
transaction	user
database	transaction, user
user	-
bank	transaction, user, database
authentication	bank

Table 4.2: Example codebase module dependencies of solution

4.4.4 Intra-Module Dependency Coupling (Cohesion)

Where coupling measures how dependent modules are on other modules, cohesion measures how well classes in a module belong together [15, 20, 29]. From the perspective of modules, classes should use other classes in the same module to serve a common purpose. The more classes in a module work together, the higher the cohesion of that module. Modules with low cohesion may indicate that the module serves multiple purposes and should be split up into smaller modules with high cohesion that serve a single purpose.

There are various ways to calculate cohesion. In their work for remodularising the Adyen codebase, Schröder et al. experimented with the following three cohesion metrics: Common Closure Principle [51], Common Reuse Principle [51] and Intra-Module Dependency (IntraMD) Coupling [35]. Their experiments show that of those three metrics Intra-Module Coupling (IntraMD) outperformed Common Close Principle and Common Reuse Principle. Based on their findings, our proposed algorithm will also use IntraMD Coupling as the cohesion metric for the algorithm. In order to prevent confusion between the terms InterMD Coupling and IntraMD Coupling, we refer to IntraMD Coupling as IntraMD Cohesion in the remainder of this thesis.

We calculate IntraMD Cohesion as follows, from the paper of Harman et al. [35]: Let $A(m)$ be a function that returns the number of class associations within a module m. Let $N(m)$ be a function that returns the number of classes in a module m and let K be the number of modules. The IntraMD Cohesion C of a module m is calculated as follows:

$$C(m) = \begin{cases} 1 & \text{If } N(m) = 1 \\ \frac{A(m)}{N(m)*(N(m)-1)} & \text{Otherwise} \end{cases} \tag{4.1}$$

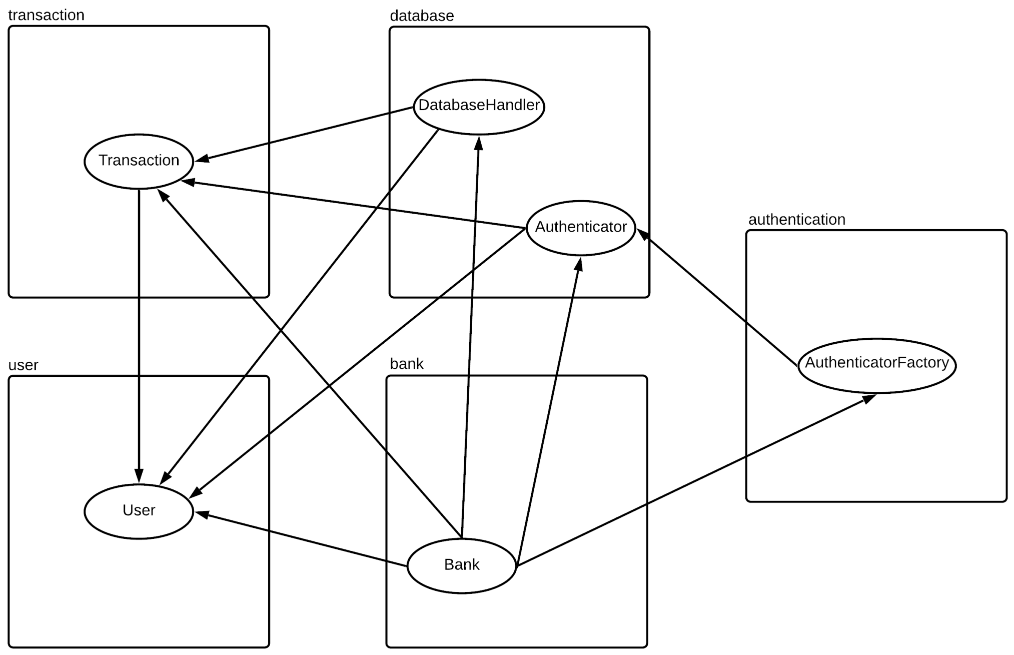


Figure 4.8: Dependency graph of example solution with worse cohesion

Then, the total IntraMD Cohesion of a codebase is calculated as follows:

$$\text{Cohesion} = \frac{\sum_m C(m)}{K} \tag{4.2}$$

Equation (4.2) shows that the IntraMD Cohesion of a codebase lies between 0 and 1, with 1 meaning that all classes in all modules are associated with all other classes in the same module. An example of such a codebase is shown in Figure 4.2. All modules either contain a single class, or all classes in a module are associated with with all other classes in the same module, thus the total IntraMD Cohesion is 1. This is also true for the codebase represented in Figure 4.3. An example of a codebase with worse IntraMD Cohesion is shown in Figure 4.8. In this solution, *Authenticator* is moved to module *database*. One can see that neither *DatabaseHandler* nor *Authenticator* depend on the other, so the IntraMD Cohesion of module *database* is 0, following Equation (4.1). From Equation (4.2) it shows that the total IntraMD Cohesion is 0.8.

The previous examples show that for IntraMD Cohesion, higher values are desired. This conflicts with the attempt of the algorithm to minimise the fitness values of solutions for each objective function. In order to ensure that the algorithm optimises for solutions with better IntraMD Cohesion rather than worse, the resulting value should reflect that lower fitness values correspond with desired solutions. This is achieved by multiplying the IntraMD Cohesion value by -1. The objective function for IntraMD Cohesion is shown in Equation (4.3).

$$\text{Cohesion} = -1 * \frac{\sum_m C(m)}{K} \tag{4.3}$$

4.4.5 Domain rule violations

Developers can use the DSL mentioned in Section 4.3 to define their domain knowledge such that the algorithm can detect whether suggested refactorings introduce or remove domain rule violations. The algorithm should use solutions' domain rule violation penalty as a minimisation objective function to optimise for solutions with minimal domain rule violations. Given all domain rules, the objective function calculates the total penalty for a solution.

4.4.6 Largest module

Not all codebases have modules of roughly equal size and purposes. As mentioned in Section 1.1, in the Adyen codebase, the largest module is an artefact of the time when much of the core business code was located in a single module. Since then, efforts have been made to extract other, more specific modules from this module, but this task is not complete due to its size and complexity. To emphasise the need of Adyen developers to reduce the size of the module, we define an objective function that returns the size of the largest module in terms of the number of classes. When optimising for this objective function, the algorithm can prioritise moving classes out of the largest module. In addition to this objective function, developers can define an inward freeze for such modules, ensuring that no classes are moved into them, which would counteract the purpose of the objective function. Such a metric may not be useful to all codebases and in such cases, developers can choose to not use this metric.

With these available objective functions, the algorithm can be configured to generate modularisations that improve the quality of the codebase while minimising the number of domain rule violations in the least amount of changes possible.

4.5 Reference points

The purpose of the *reference points* is to help the NSGA-III algorithm maintain a diverse population spread out over the objective space. By prioritising the survival of solutions associated with reference points with low association counts, the algorithm attempts to prevent all solutions from being in the same region in the objective space. Reference points can be supplied beforehand or by any predefined structured placement strategy. For our algorithm, we use the same placement strategy as the original NSGA-III paper by Deb and Jain [23], mentioned in Section 2.5.3. This approach places the reference points uniformly distributed over an $(M-1)$-dimensional hyperplane, with M equal to the number of objective functions, without bias to any of the objective functions, thus allowing the algorithm to pursue maximal diversity across the objective space. Let M be the number of objective functions and p the number of divisions along each dimension. The number of reference points H is then given by Equation (4.4).

$$H = \binom{M+p-1}{p} = \frac{(M+p-1)!}{p!((M+p-1)-p)!} = \frac{(M+p-1)!}{p!(M-1)!} \tag{4.4}$$

The number of reference points is essential for the functionality of the algorithm. The population size N is directly related to the number of reference points, as $N \approx H$. A low number of reference points means few solutions and little diversity, while a higher number means that there is much diversity. However, a high number of reference points also means the population size is large, which increases the time required for the algorithm to reach convergence.

4.6 Initial population

The *initial population* of the algorithm heavily influences the first generations and, therefore, the results after the algorithm finishes. To ensure that difference between the solutions and the initial state of the codebase is not too large, the population should not be initialised with random modularisations. Initialising the population with random modularisations could either require many generations to reduce the amount of work required to implement a solution or result in a list of changes that is too large for developers to accept and implement within a reasonable amount of time. Instead, a similar approach to the one used by Schröder et al. [68] is used, where the population is initialised by mutating the initial codebase several times. This approach results in an initial population of unique modularisations while ensuring that the number of changes in each initial solution does not grow out of hand. Like the approach by Schröder et al., each solution in the initial population is mutated 20 times.

4.7 Parent selection

Parent selection is determines which pairs of solutions will generate offspring through the crossover operator. The functionality of the crossover operator is discussed in Section 4.8. The idea behind selection is to mimic natural selection, which means that solutions with desirable traits (i.e. high fitness) have a higher chance of surviving and generating offspring than solutions with lower fitness. Tournament selection of tournament size 5 is used to determine pairs of parents that will generate offspring, the same tournament size as the tournament selection approach used by Schröder et al. [68]. Solutions in the tournament are sorted in fronts based on domination. Solutions in the first front are non-dominated, solutions in the second front are dominated by the first front and so on. If the non-dominated front contains at least two solutions, two solutions are randomly picked from that front and used as parent pair for crossover. If the non-dominated front contains only one solution, that solution and a randomly picked solution from the second front are used as parent pair for crossover. The introduction of an elite archive is later discussed in Section 4.9.1. Essentially, solutions in a tournament are ranked on domination, thus favouring solutions that are not dominated over those that are. Solutions in the non-dominated set of the population will always appear in the non-dominated front of a tournament. However, by chance, a dominated solution may win a tournament if none of the solutions it is dominated by are in the same tournament.

4.8 Crossover operator

Crossover is the operation used by genetic algorithms to create offspring from existing solutions in the population. As parent solutions with desirable traits have a high probability of surviving and creating offspring, the general idea is that their resulting offspring has slightly different but also desirable traits. This offspring, in turn, has a high probability of surviving and therefore creating new offspring, repeating the cycle inspired by natural selection. The crossover operator used by the algorithm to create modularisations is inspired by Harman et al. [35] and is the same operator as the building block preserving crossover used by Schröder et al. [68], as described earlier in Section 2.4.1. As mentioned in Section 4.1, traditional crossover methods such as single point, two-point and uniform crossover would result in offspring that do not preserve their parents' module structure. The building block preserving crossover operator attempts to preserve module structure by doing module-level uniform crossover instead of at class level. The order of modules is random for each pair of parents. By attempting to move an entire module from a parent to a child, the changes that happened in the parent have a high probability of being preserved in the offspring.

Figures 4.9 and 4.10 illustrate how the operator may fail to preserve module structure, thereby splitting the changes of a parent for a module. Figure 4.9 shows the module structure of two parents. Both parents have a refactoring suggestion involving class *AuthenticatorFactory*. Parent 1 suggests moving the class to module *transaction*, and parent 2 suggests moving the class to module *user*. Assume the order the operator moves the module in is *transaction, user, database, bank, authentication*. Figure 4.10 shows the children resulting from performing the crossover operator on the parents. Modules and class coloured green originate from parent 1, while those from parent 2 are coloured red. The first module is *transaction*, which child 1 receives from parent 1 and child 2 from parent 2. As this is the first module, no conflicts can happen. However, by chance, child 1 is to receive the module *user* from parent 2, but this module contains class *AuthenticatorFactory*, which child 1 already received in module *transaction* from parent 1. Since the class cannot exist twice in one child and zero times in the other, this conflicting class is instead moved to child 2. This can be seen by the red outline of class *AuthenticatorFactory* in the *user* module of child 2. In this example, it is as if child 1 instead received module *user* from parent 1 and child 2 from parent 2, but for a larger codebase with more classes and therefore more potential changes in parent solutions, one can imagine that such a conflict instead results in spreading the changes of one parent over two children. However, if the number of changes in solutions is kept at a reasonable amount, the number of conflicts should not be significant, and most modules can be moved entirely from a parent to the target child solution.

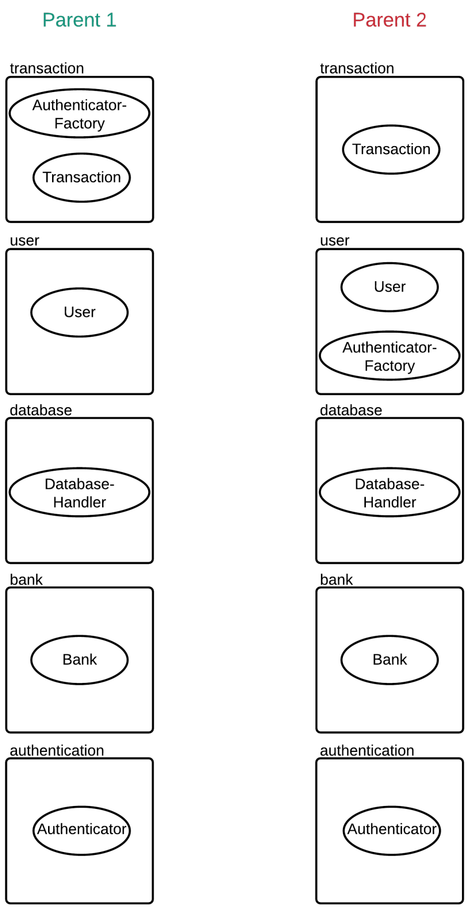


Figure 4.9: Example of parent pair for crossover

4.9 Mutation operator

The *mutation* operator allows the NSGA-III algorithm to introduce changes to solutions that affect their fitness. Every mutation corresponds to moving a class to a different module. As mentioned in Sections 4.1 and 4.8, solutions represent changes compared to the modularisation of the codebase, rather than a complete mapping of every class to a module. Because of this representation, mutations are necessary to introduce any changes that propagate to the offspring created by the crossover operator.

The algorithm uses a single-class mutation operator to mutate solutions, which moves one class per mutation. However, the algorithm proposed by Schröder et al. provided two different mutation operators that were able to move small and large sets of classes from one module to another.

The first operator is based on picking a random class in a module and creating its dependency graph within that module, up to a certain size. The operator then performed a min-cut on the dependency graph, moving the sub-graph that contained the originally picked class

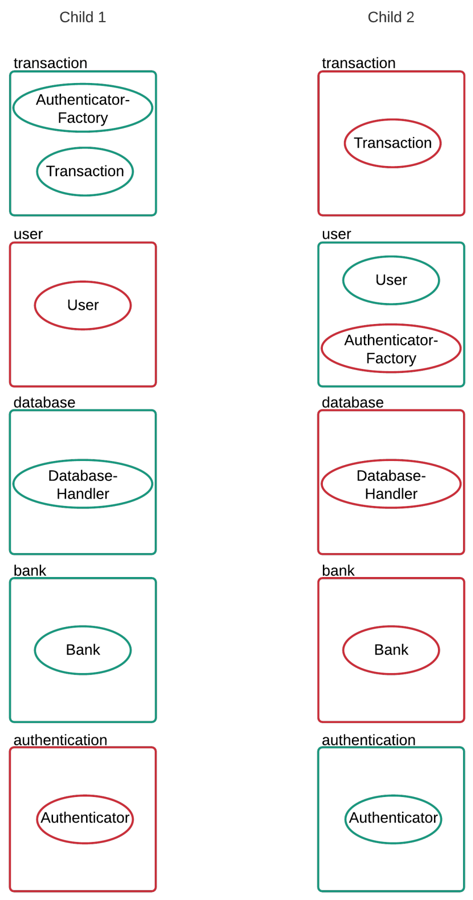


Figure 4.10: Example of crossover conflict

to a different module.

The second operator added a random class in a module to a set. Then with decreasing probability, classes related to the classes in the set and in the same module as the initially picked class were added to that set. The operator picks a random number r between 0 and 1, and adds a related class to the set as long as $r < (c)^n$, where c a constant between 0 and 1 and n an integer starting at 0 that is incremented every time a class gets added to the set. Finally, all modules related to the set are determined and one is randomly picked as the destination for the set.

These operators allow the algorithm to move groups of clusters of varying sizes from a module, but these groups are not necessarily very cohesive. The groups resulting from both operators have some cohesion. However, both operators do little to determine if a group is highly cohesive, which would require extra effort, such as semantic analysis of the class names to determine how related the classes are, thereby slowing down the algorithm's speed. Additionally, both operators pick a random class, without checking where that class lies in the dependency graph in the module. Checking this is important, as moving a class that has both dependencies and is depended on by classes in its module will cause a circular dependency to and from its new destination. To illustrate this, the codebase corresponding to Figure 4.2 has been enhanced such that the module *user* contains such classes. Figure 4.11 shows this dependency graph, where class *Role* in module *user* depends on classes *User* and *Admin*, and those two both depend on class *PasswordChecker*. If class *User* were to be moved to a related module like *transaction*, modules *user* and *transaction* would form a circular dependency, since *Role* depends on *User* and *User* depends on *PasswordChecker*. For both operators, it could happen that the set of classes to be moved only contains class *User*, which would introduce circular dependencies.

Instead, for our NSGA-III algorithm, a single-class mutation operator is chosen that only moves *roots* and *leaves* of a module. A *root class* in a module is defined as a class that is not depended on by other classes in the same module, while a *leaf class* is defined as a class that does not depend on other classes in the same module. Classes *Role* and *PasswordChecker* in Figure 4.11 are examples of roots and leaves, respectively. Only moving roots and leaves does not guarantee that circular dependencies are prevented. An example can be seen in Figure 4.11. Class *PasswordChecker* is a leaf of module *user*, and its only related module is *authentication*. If *PasswordChecker* were to be moved to *authentication*, however, it would introduce a circular dependency. Module *authentication* would still depend on module *user* because of the dependency from *Authenticator* on *User*. At the same time, module *user* now also depends on *authentication* because of *Admin* and the dependency from *User* on *PasswordChecker*. In this example, the circular dependency is between two modules, but the example could be extended such that the cycle involves more than two modules. As a result, the entire dependency graph may need to be checked for cycles to detect whether a mutation introduces circular dependencies.

A version of the single-class mutation operator was tried during preliminary testing that moves a random root or leaf only if the move does not result in a circular dependency, which used the DFS algorithm as mentioned in Section 4.4.2, to check for cycles in the module dependency graph. While this prevented circular module dependencies from being introduced during mutation, it also slowed down the performance of the mutation operator by an order of magnitude. Circular dependencies could still be introduced when using this mutation operator due to how the crossover operator works. The crossover operator moves all classes from a module to a child that the child did not receive earlier from the other parent. Otherwise, the child would receive a class twice while the other child would not receive it at all. In that case, the second time a child receives a class, that class instead goes to the other child. This can also result in circular dependencies, which means that slowing down the mutation operator with a DFS algorithm to check for circular dependencies affects the algorithm's speed but does not entirely prevent circular dependencies. Instead, we have chosen to optimise for circular dependencies with the objective function described in Section 4.4.2.

It is worth nothing that we have chosen to prevent the algorithm from creating new modules in the codebase, since our algorithm uses a mutation operator that moves a single class for a solution per generation. This approach contrasts with that of Schröder et al. [68], where the algorithm could create new modules. The mutation operators used by their algorithm could move more than a single class per mutation, and the authors still noted that

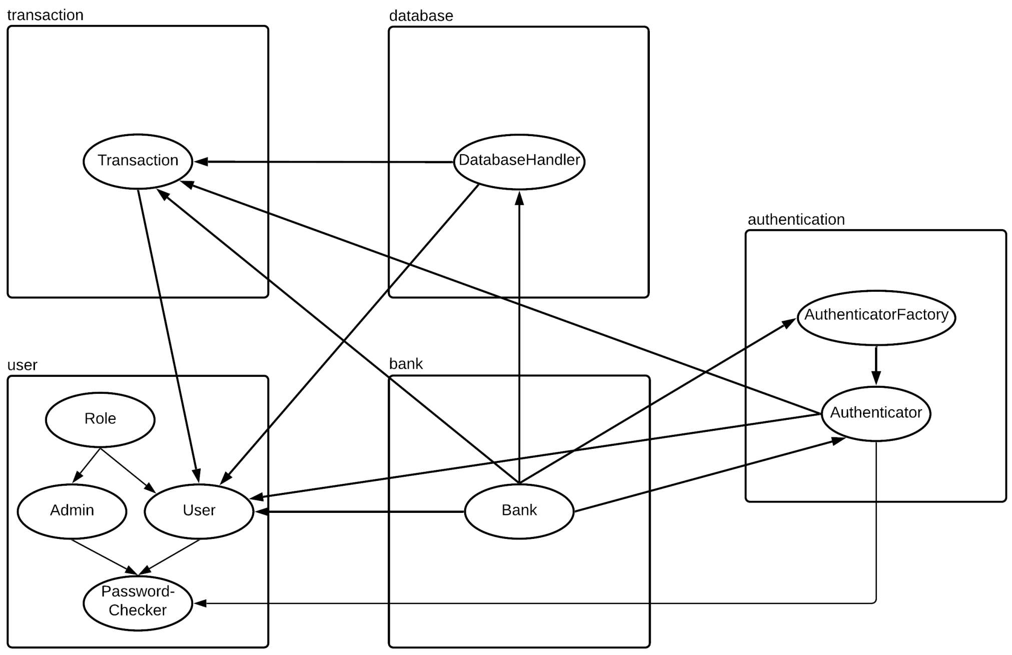


Figure 4.11: Extended dependency graph

this often resulted in modules that were too granular for large-scale codebases like the one of Adyen. From their findings, combined with the fact that our mutation operator moves one class at a time, we conclude that allowing the algorithm to create new modules would also result in modules that are too granular for large-scale codebases. Thus, we do not include this functionality for our NSGA-III algorithm.

4.9.1 Elite archive

NSGA-III uses a set of reference points to ensure that solutions are well-spread through the objective space. During the niche-preservation operation, solutions in the last considered front F_l that are associated with a reference point with the minimum niche count are added to the surviving population until the surviving population reaches its maximum size. This operation helps spread the solutions over the objective space because it attempts to keep solutions in the direction of as many reference points as possible. Still, this does not ensure that every reference point has a solution associated to it because reference points with no associated solutions in F_l cannot result in a related solution being added to the surviving population. Therefore, over time, solutions may be associated with a small numbers of reference points, thus resulting in a limited spread over the entire objective space.

In order to prevent the number of reference points with at least one associated solution from decreasing, an elite archive is implemented. Ibrahim et al. [39] propose an elite archive for NSGA-III that maintains an elite solution for every reference point. First, each solution in the initial population is associated with its closest reference point. Then, for each reference point with at least one associated solution, the solution with the smallest Euclidean distance to the ideal point is chosen as the elite solution for that reference point.

The solutions in the elite archive are used during parent selection. Without the elite archive, two solutions, r_1 and r_2 from the current population are selected, by any means of selection operation, such as tournament selection or random picking with or without replacement. These solutions become a parent pair (p_1, p_2) used during crossover.

From the elite archive, two solutions r_3 and r_4 are selected at random, such that $r_3 \neq r_4$. Then with 50% probability, r_3 replaces r_1 as parent p_1. Similarly, with 50% probability, r_4 replaces r_2 as parent p_2. This ensures that the diversity of the offspring is improved. The elite archive also improves the diversity of parents. In early generations of the algorithm, the population is associated with a few reference points, meaning it is likely that parents associated with the same reference point are selected, resulting in solutions in the offspring population that are close to their parents, thus limiting the diversity. Because there is at most one solution associated with a reference point, the probability that both parents belong to the same reference point is limited, which improves the diversity of the population.

After the niche-preservation operation at the end of each generation, the elite archive is once again updated. Every solution in the surviving population is associated with a reference point. Suppose the Euclidean distance of a solution is smaller than that of the elite solution associated with the same reference point. In that case, that solution becomes the new elite solution for that reference point.

It can still happen that a reference point never has a solution associated with it. However, unlike normal NSGA-III, the number of reference points with at least one associated solution does not decrease. Ibrahim et al. [39] show that over time for WFG6 test problems [37, 38] the amount of reference points with at least one associated population member decreases for NSGA-III, while it increases for EliteNSGA-III, thereby showing that an elite archive helps NSGA-III maintain and improve the diversity of the population.

Chapter 5

An empirical study on the convergence of the algorithm

Having discussed how the algorithm operates in the previous chapter, it is now time to analyse the performance depending on the parameters of the algorithm, through an empirical study. **This empirical study will analyse the performance and convergence of the algorithm.** The chapter starts by listing the research questions that the empirical study will answer. Next, the methodology used for the empirical study is described, followed by explaining how the algorithm's parameters were tuned to obtain the best performance of the algorithm. Lastly, the results of the empirical study are analysed and used to answer the research questions of this chapter.

5.1 Research questions

To understand the performance and convergence of the algorithm, we want to answer the following research questions:

RQ1: What are the improvements to the codebase in terms of the objective values? How well does the algorithm generate modularisations that improve the code structure quality metrics?

RQ2: How fast does the algorithm converge? We want to know how many generations it takes for the algorithm to reach a local optimum. By analysing the improvement of the fitness values and the Pareto optimal front over generations, we can detect whether the algorithm has converged and how many generations it needed.

5.2 Methodology

The algorithm is run on 16 cores at 3.3GHz and 128GB of RAM, two times for each research question. The algorithm runs on the entire codebase, and the Pareto front is saved every 50 generations. The algorithm is run for 24 hours for **RQ2**, and 72 hours for **RQ2**. In order for the algorithm to perform appropriately for the research questions, the algorithm's parameters must first be tuned, as will be described in Section 5.3. The objective function for *Domain rule violations* is not used for the study in this chapter, as it will be evaluated during the case studies with Adyen developers performed in the next chapter, because the Adyen developers provide the domain rules for the algorithm. However, the study in this chapter is performed without any Adyen developer input and therefore, without any domain rules.

5.3 Parameter tuning

The algorithm's parameters must be appropriately configured to ensure that the algorithm produces good modularisations in a timely manner. Because a significant amount of runs are required for tuning, a reduced version of the codebase is used to ensure the parameter tuning finishes promptly. This smaller version is obtained by running the algorithm on the codebase with a large portion frozen. Freezing a large part of the codebase reduces the algorithm's search space but still allows it to consider all dependencies so it can still detect any cycles. The version consisted of 105 modules, totalling 5476 classes that were not frozen. These chosen modules represent various functionality present in the codebase. For each parameter, various values were tried to determine which works best. The details of this tuning are described in Appendix B, where six parameters were tuned for a total of 17 configurations. Each configuration was run five times, for three hours per configuration by the algorithm, on the partially frozen codebase. The results and details are outlined in Appendix B, and the final parameter values are listed below:

- Number of reference points: 10x the number of objective functions
- Population size: 1x the number of reference points
- Mutation chance: 50%
- Elite archive: is used
- Number of cycles: is optimised for
- Largest module: is optimised for

5.4 Results

This section shows the results of the empirical study to answer **RQ1** and **RQ2** regarding the algorithm's performance and convergence. Similarly to the parameter tuning results in Appendix B, the relative improvements are shown rather than the absolute values, per the request of Adyen for confidentiality. We only consider the metric improvements of the solutions generated by the algorithm without validating the usefulness of the solutions with Adyen developers during this empirical study.

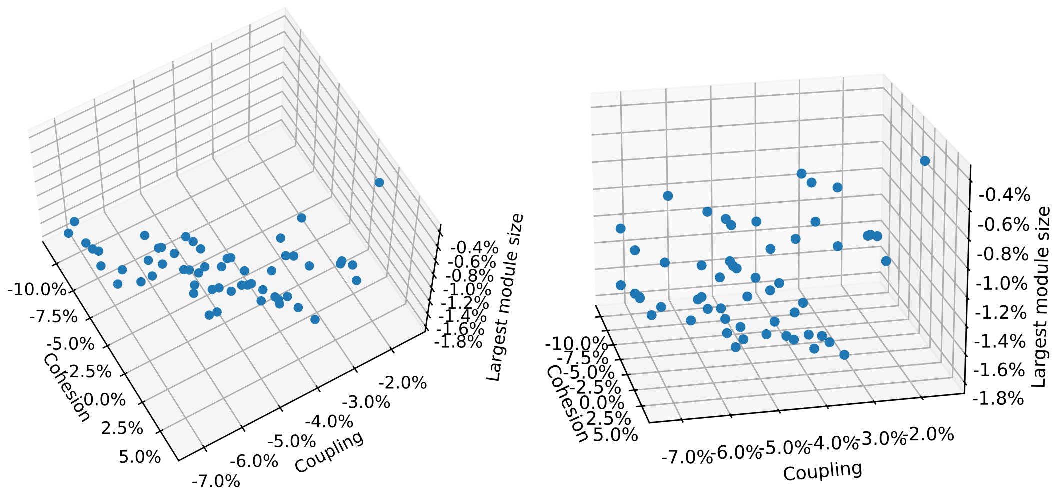


Figure 5.1: First run Pareto front

Best solution	IntraMD Cohesion	InterMD Coupling	Largest module size	Change impact
IntraMD Cohesion	8.60%	-4.12%	-1.06%	198
InterMD Coupling	-7.12%	-8.10%	-1.48%	301
Largest module size	-1.95%	-6.57%	-1.97%	291

Table 5.1: Relative improvement of best solutions for each objective function

5.4.1 RQ1: What are the improvements to the codebase in terms of the objective values?

Figures 5.1 and 5.3 show the Pareto fronts for both runs, for the objective functions *IntraMD Cohesion*, *InterMD Coupling* and *Largest Module Size*. Figures 5.2 and 5.4 show the solutions in the Pareto front that improve on all metrics for both runs. The graphs do not show the fourth objective function (Circular Module Dependencies). However, since only solutions without cycles are usable, any solutions that contained a cycle in the module dependency graph were discarded for the purpose of answering RQ1. Table 5.1 shows for each objective function which metric values the best solution for that objective achieved. Table 5.2 shows the same for solutions that improved on all metrics. While InterMD Coupling and Largest Module Size are a minimisation effort, IntraMD Cohesion should be maximised, as mentioned in Section 4.4.4.

Observation 1: The algorithm can produce modularisations that significantly improve the code structure quality in terms of the objective functions. Considering solutions that improve all metrics, as displayed in Table 5.2, one can see that IntraMD Cohesion can be increased by 8.6%, InterMD Coupling decreased by 6.67%, and the largest module size could be reduced by 1.94%. When the best solutions for each objective function that

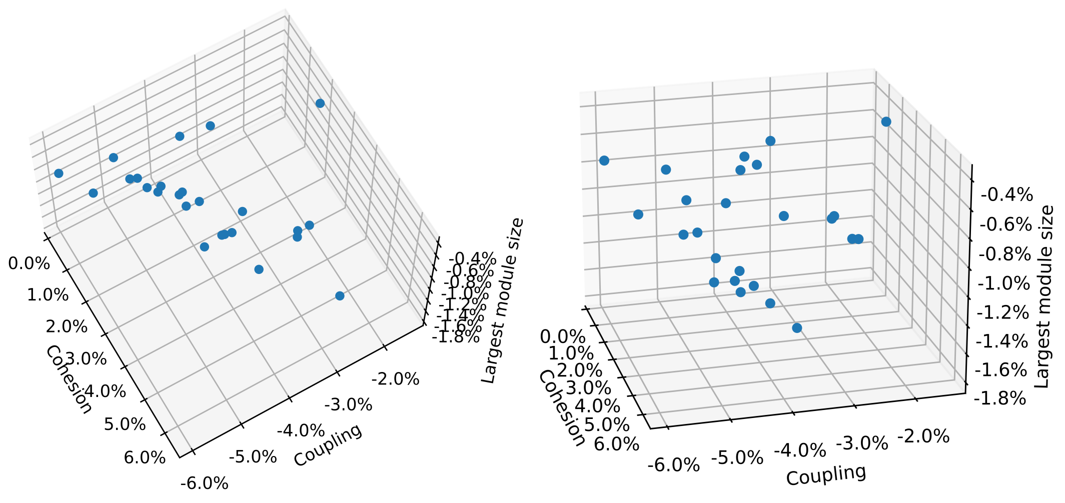


Figure 5.2: First run Pareto front solutions that improve all metrics

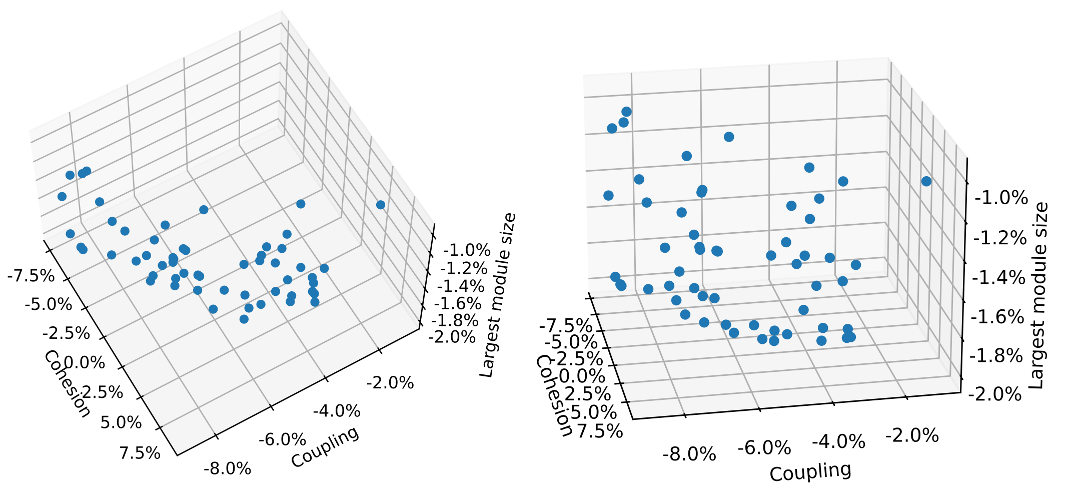
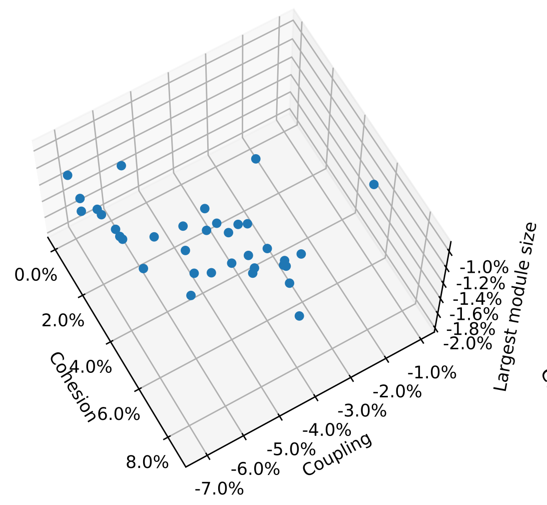


Figure 5.3: Second run Pareto front

Best solution	IntraMD Cohesion	InterMD Coupling	Largest module size	Change impact
IntraMD Cohesion	8.60%	-4.12%	-1.06%	198
InterMD Coupling	0.57%	-6.67%	-1.06%	238
Largest module size	3.38%	-6.57%	-1.94%	271

Table 5.2: Relative improvement of best solutions for each objective function that improve all metrics



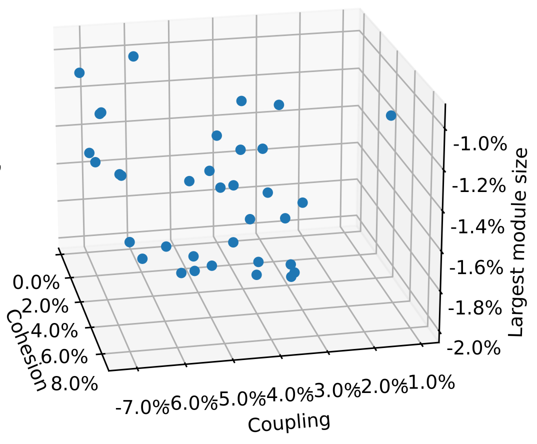


Figure 5.4: Second run Pareto front solutions that improve all metrics

improve some metrics at the cost of others are considered, as shown in Table 5.1, even better improvements can be seen for InterMD Coupling and the size of the largest module. InterMD Coupling can be reduced by 8.1%, while the largest module could be reduced in size by 1.97%. These tables show that the algorithm is able to produce significantly improved modularisations from the perspective of code structure quality metrics.

Observation 2: The best solution in terms of IntraMD Cohesion also improves the other two metrics. The best IntraMD Cohesion solution in Table 5.1 is the same as in Table 5.2, showing that the best IntraMD Cohesion improvement of 8.6% is obtained while also decreasing InterMD Coupling by 6.67% and the size of the largest module by 1.06%. Additionally, when comparing Figures 5.1 and 5.3 to Figures 5.2 and 5.4, one can see that IntraMD Cohesion is also the only metric that is not always improved upon in the Pareto front of solutions without cycles, as all solutions in the figures improved both InterMD Coupling and the size of the largest module. The reason for this is unclear and would require more runs to determine why this occurred.

Observation 3: The optimal solution for IntraMD Cohesion needs fewer changes than the optimal solutions for InterMD Coupling and the size of the largest module. The best solution for IntraMD Cohesion requires the least changes to the codebase, while the best solutions for InterMD Coupling and largest module size require approximately the same amount of changes. An explanation for this could be that every move can improve IntraMD Cohesion, but only when every class in a module M_i that depends on another module M_j are moved out of M_i does the InterMD Coupling of M_i decrease, which could require a large amount of moves depending on the number of classes in M_i that depend on M_j. Additionally for the largest module size, there are many more classes not in the largest module than there are in it. As the mutation operator randomly picks classes to move, it can take many moves of other classes before the algorithm moves a class out of the largest module, which could explain why the best solutions for InterMD Coupling and largest module size require more moves than the best solution for IntraMD Cohesion.

5.4.2 RQ2: How fast does the algorithm converge?

The algorithm performed two runs of 72 hours each to analyse the algorithm's convergence for the objective functions. For each objective function, the best and average results are shown in Figures 5.5 to 5.8.

Observation 4: The average solution improves InterMD Coupling and the largest module size at the cost of IntraMD Cohesion. Looking at Figures 5.5 to 5.7, one can see that the average value for IntraMD Cohesion gets worse while the average values for InterMD Coupling and the largest module size improve.

Observation 5: The algorithm quickly reached a local optimum for IntraMD Cohesion. Figure 5.5 shows that the best improvement for IntraMD Cohesion, roughly 9.6%, is achieved after approximately 50,000 generations. For the remainder of the runs, no solution achieves a better value, indicating that the algorithm likely is stuck in a local optimum in terms of IntraMD Cohesion.

Observation 6: The algorithm does not seem to converge for InterMD Coupling and the largest module size. As shown in Figures 5.6 and 5.7, the best values for both objective functions steadily decrease throughout the 72-hour runs. The algorithm appeared to converge after around 350,000 generations for InterMD Coupling. However, near the end of the runs, the value once again decreases. The algorithm converged quickly for the best value for IntraMD Cohesion, but the average value does not seem to converge within the 72 hours.

Observation 7: Some generations do not contain solutions without cycles. As the number of cycles is one of the objective functions, it is expected that every generation contains some solutions with cycles as the result of the algorithm trading off between fitness values. While the exact number estimated by the algorithm does not matter, one can see from the best values in Figure 5.8 that even the best solution contains cycles for some generations.

Observation 8: The algorithm computed approximately 460,000 generations after 72 hours. During both runs, the number of generations computed was approximately 460,000. The speed of the algorithm is affected by various parameters, such as the number of objective functions and the population size. Additionally, the machine that performed the runs is an important aspect for the actual speed of the algorithm. Considering that the best improvement for IntraMD Cohesion was achieved after 50,000 generations, the algorithm ran for approximately 8 hours to reach this improvement.

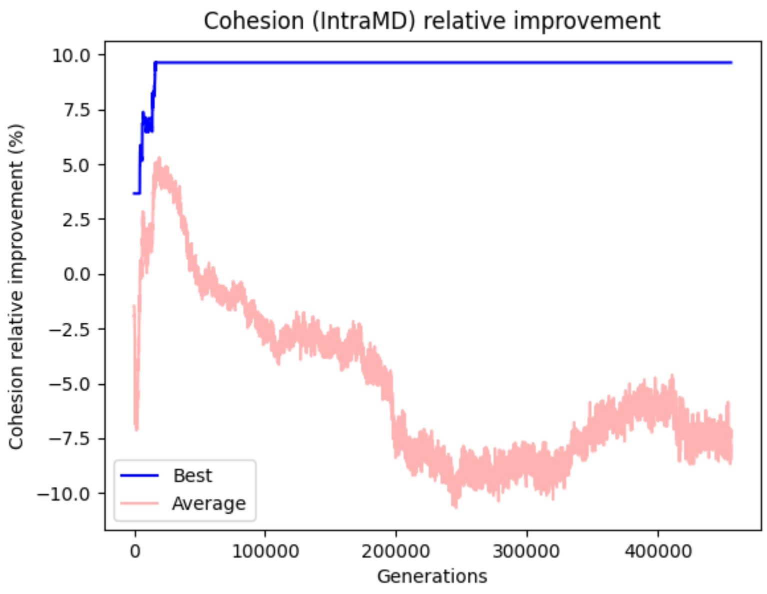


Figure 5.5: IntraMD Cohesion convergence, best and average values of 72 hour runs.

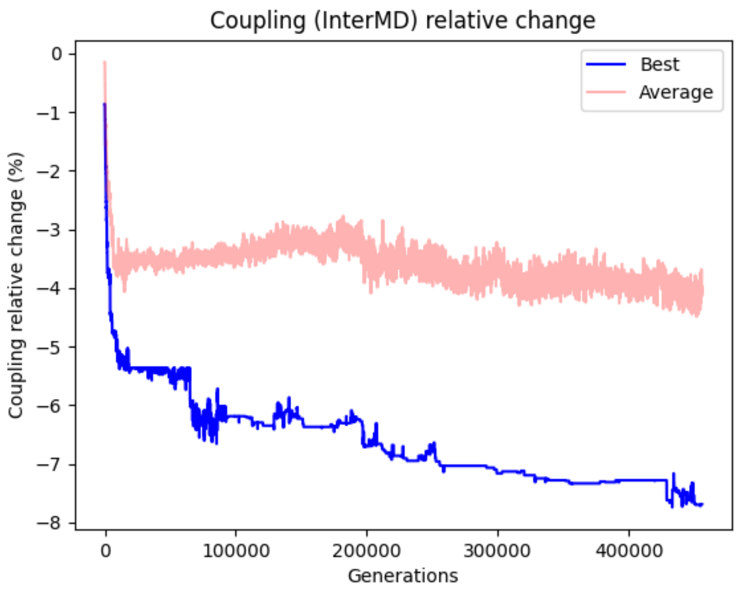


Figure 5.6: InterMD Coupling convergence, best and average values of 72 hour runs.

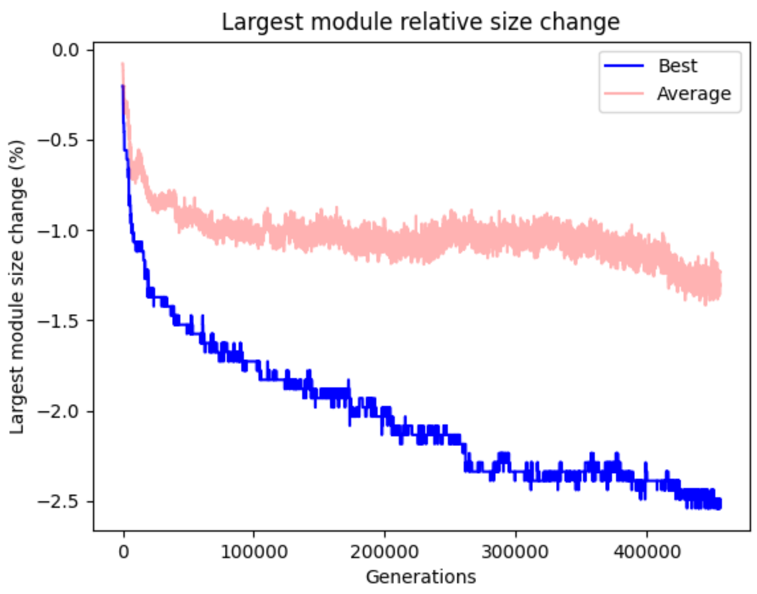


Figure 5.7: Largest module size convergence, best and average values of 72 hour runs.

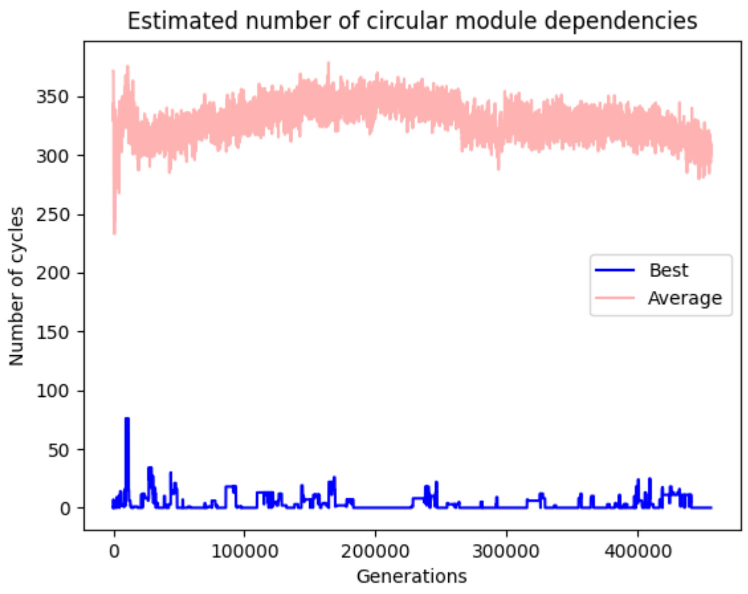


Figure 5.8: Circular module dependency convergence, best and average values of 72 hour runs.

5.5 Conclusion

Considering the results in Figures 5.1 to 5.4 and Tables 5.1 and 5.2, we can answer the following for RQ1: The algorithm shows improvements of 8.6% for IntraMD Cohesion, 6.67% for InterMD Coupling and 1.94% for the size of the largest module, considering modularisations that improve all three metrics and contain no cycles.

To analyse the algorithm's convergence for each objective function, we let the algorithm run twice, for 72 hours, with the best and average results for those runs shown in Figures 5.5 to 5.8. Considering these results, we can answer for RQ2 that the algorithm does not seem to converge for any objective function except IntraMD Cohesion. However, the average IntraMD Cohesion value did not seem to converge. Thus we recommend performing more runs in the future to further analyse the convergence of the algorithm for IntraMD Cohesion.

Chapter 6

Case studies at Adyen

This chapter performs three case studies by running the algorithm on different parts of the Adyen codebase. **The goal of performing these case studies is to understand the developers' domain knowledge and how this knowledge can be used to generate better modularisation suggestions.** Each case study focuses on a different team and the modules they are responsible for. This chapter starts by describing which case study focuses on which parts of the Adyen codebase. Next, the methodology for the case studies is explained, along with the developer selection and interview process. Finally, the chapter concludes by listing the results of each case study and evaluating the effect of domain knowledge on the algorithm's performance.

6.1 Case studies

Case study 1 (CS1): The first case study focuses on a core feature of Adyen from the start of the company, making it one of the oldest parts of the codebase. It has been developed, maintained and extended over the years, resulting in an architecture that differs from when it was initially created. For this reason, modules related to this core feature are chosen for the first case study.

Case study 2 (CS2): To complement the first case study on an older part of the codebase, the second case study considers the modules related to a relatively new addition to the codebase. Because it is a younger part of the codebase, the domain knowledge developers have for this part of the codebase may differ from that of the developers for the first case study.

Case study 3 (CS3): The last case study considers an internal framework. While the modules of each of the other two case studies serve their own specific purpose, the modules belonging to this case study are used in various parts of the codebase. Furthermore, while the modules are maintained by one team, multiple domains make use of it, thus making these modules an intriguing choice for the last case study.

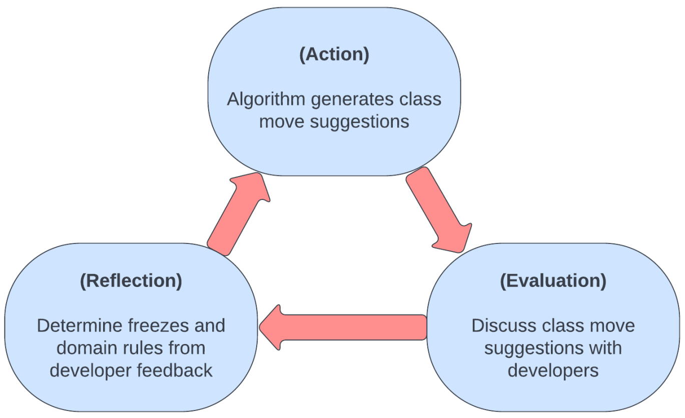

(Action)
Algorithm generates class move suggestions
(Evaluation)
Discuss class move suggestions with developers
(Reflection)
Determine freezes and domain rules from developer feedback

Figure 6.1: Case study action research cycle diagram

It is worth noting that the case studies do not have overlapping classes or modules.

6.2 Methodology

In the previous chapter, we performed an empirical study on the performance and convergence of the algorithm. The results obtained in that chapter show that the algorithm can significantly improve a large-scale codebase from a metrics perspective. However, an important question is whether the remodularisations created by the algorithm are actually useful from a real-world point of view. To answer this question, we performed three case studies inspired by the canonical action research cycle described in **Action research in software engineering** [70]. We combine the *Action planning* and *Action taking* steps into a single *Action* step, follow the *Evaluation* step and combine the *Learning* and *Diagnosing* steps into a single *Reflection* step:

- **Action -** Run the algorithm with the established domain knowledge as input.
- **Evaluation -** Discuss the solutions with the developers.
- **Reflection -** Determine new domain knowledge from the feedback of the evaluation phase.

After the reflection phase, the algorithm is rerun with the new domain knowledge as input to create solutions that are better in line with the knowledge of the developers. Performing the three steps multiple times allows us to reflect on the experience and feedback of the developers, which in turn improves the solutions that the algorithm generates. Figure 6.1 visualises the action research cycle used by each case study.

While developers are familiar with their domain knowledge when developing, it is unlikely that they can exhaustively formulate it all from the top of their heads. However, they can identify when a move-class suggestion goes against their knowledge. Therefore to establish a knowledge base relevant to the developers, they will be interviewed on suggestions made by the algorithm, which can contain violations to yet unspecified domain knowledge. During these interviews, developers can state why they disagree with specific move-class suggestions. From this feedback, new freezes or domain rules can be determined, which can then be used as input for future runs of the algorithm. Therefore, performing several iterations of the cycle in figure 6.1 fits very well with our iterative interview approach.

For each case study, the following objective functions are optimised for by the algorithm: *Change impact (number of classes moved)*, *IntraMD Cohesion*, *InterMD Coupling*, *Cycles (number of circular module dependencies)* and *Rule penalty (number of domain rule violations)*. The algorithm was run for 12 hours before each iteration.

6.2.1 Developers

At Adyen, developer teams own sets of modules. To select the developers for each case study, the developers in the teams that own the modules were asked to take part in the case study via the internal chat platform. As a result, three developers accepted for the second case study, while two accepted each for the first and third case studies.

6.2.2 Initial interviews

Each of the initial interviews was done in person or through an online conference application. Before discussing any results of the algorithm, the developers of each case study are explained which metrics the algorithm optimises for, how these metrics are calculated and how the algorithm can choose to which modules classes can be moved. In addition, they were asked which modules should be considered for the case study to specify the initial freezes that narrow the algorithm's search space.

Additionally, they were explained how **the goal of the case study is to capture their domain knowledge to improve the remodularisation solutions of the algorithm through the use of freezing parts of the codebase and defining domain rules.** They were also explained how this process happens throughout several runs and iterations to iteratively improve the set of domain knowledge used as input for the algorithm.

Next, the developers were asked several questions to understand their experience with and knowledge of the Adyen codebase. These questions were inspired by the interview approach of Schröder et al. [68]. The questions and answers can be found in Appendix C.1.

After each iteration, the algorithm was run with the codebase frozen such that classes in unrelated modules were not moved, thus ensuring that the algorithm only suggests moves for classes with which the developers are familiar. For the initial runs, no domain knowledge was specified. Therefore, the algorithm did not optimise for *Rule penalty* until the second iteration of each case study.

6.2.3 Splitting solutions

While *Change impact* is one of the objective functions the algorithm optimises for, solutions at the end of a run can still consist of dozens of move-class suggestions. Solutions were split into smaller groups of move class suggestions, each group consisting of 10 or fewer suggestions, to ensure that each interview did not take more than 30 minutes. These smaller groups are referred to as *change groups* in the remainder of this chapter. Move-class suggestions were grouped in change groups according to their module destination. The change groups were first filtered on duplicate classes. Finally, those change groups that did not contain cycles and improved at least one of the following objective functions were selected: *IntraMD Cohesion, InterMD Coupling* and *Rule penalty*.

6.2.4 Action research iterations

In the action research iterations following the initial meeting with the developers, various change groups were discussed with the developers to determine domain knowledge to improve the future runs of the algorithm. For each case study, three iterations discussing change groups were performed, in addition to the initial interview. A metric overview for the change groups discussed during the iterations of each case study can be found in Appendix D. For all three case studies, the *Rule penalty* objective function was considered after the first iteration, as that was the iteration after which domain knowledge was specified.

6.3 Case studies results

In this section, we describe the action research iterations for each case study. The metric changes of each change group discussed for all case studies can be found in Appendix D. For each iteration, we list and discuss which change groups and move-class suggestions were accepted, partially accepted and rejected by the developers and list their feedback explaining the choices they made. At the end of each iteration, we summarise the results and use the developer feedback to determine new domain rules and freezes, which are subsequently used by future runs to improve remodularisation.

In general, the modules of the Adyen codebase can be categorised as follows:

- *High-level* - Applications and interfaces.
- *Mid-level* - Business logic functionality.
- *Low-level* - Basic functionality, e.g. string manipulation.

6.3.1 Case study 1

Iteration 1

Table D.1 shows the change groups discussed during the first iteration.

Accepted: No change groups were accepted by the developers.

Partially accepted: No change groups were partially accepted by the developers.

Rejected: Every change group contained move-class suggestions that would introduce unwanted dependencies between several different module types.

Interpretation of the expert developers: As shown in Table D.1, the developers rejected all ten move-class suggestions, even if they contributed positively to both IntraMD Cohesion and InterMD Coupling. The developers indicated that moving these classes to the suggested modules would introduce unwanted dependencies between various modules. From their feedback, multiple domain rules were established for future runs:

- Multiple module-to-module dependencies are not allowed based on their module types, e.g. dependencies between *High-level* modules.

Iteration 2

An overview of the change groups discussed during the second iteration can be found in Table D.2.

Accepted: No change groups were accepted by the developers.

Partially accepted: Change group G1 contained one move-class suggestion that developers accepted. The other two suggestions were rejected because they were determined to be in the correct module, regardless of the code structure quality improvement resulting from the moves.

The first move-class suggestion in change group G4 was rejected because the class was determined to be in the correct module. For the second suggestion, the developers confirmed that the class could be moved, but the suggested module was incorrect, as this would move a class from a *Mid-level* module to a *Low-level* module.

Change group G5 suggested moving three classes. The developers confirmed that the classes could be moved out of their current module. However, the suggested modules were rejected because this would move classes from *High-level* modules to *Low-level* modules, which violated a domain rule specified in the last iteration, as can be seen by the increase in Rule penalty.

For each suggestion in change group G6 the developers confirmed that the classes should be moved out of their current module. However, they rejected the changes because the suggested module was incorrect.

Rejected: All nine suggestions in change group G2 were rejected because the classes were determined to be in the correct modules, regardless of the code structure quality improvement resulting from the moves. For four of these suggestions, the developers stated that the suggestions would move code from *Mid-level* modules to a *High-level* module, which they did not want.

The move-class suggestion in change group G3 was rejected because the class was determined to be in the correct module, regardless of the code structure quality improvement resulting from the move. The suggested module was also explicitly rejected because it is special, requiring a different approach than the other modules owned by the team. It was concluded that this specific module was best left out of consideration for future runs.

Interpretation of the expert developers: Out of the six change groups comprising 21 move-class suggestions, none were outright accepted by the developers. Out of those 21 move-class suggestions, one was accepted, seven classes could be moved to a different module than suggested, and 13 classes were already in the correct module.

It is worth noting that the move-class suggestion in change group G3 was rejected because the class was determined to be in the correct module, which is of special nature and requires a different approach than the other modules owned by the developers. It was concluded that this specific module was best left out of consideration for future runs.

For each of the seven classes that could be moved to a different module, the developers could name modules that they found more suitable as locations. However, the algorithm could not have proposed those modules because it only picks related modules in terms of class dependencies.

Several move-class suggestions in G2 were rejected because this would move classes from *Mid-level* modules to *High-level* modules, which the developers did not want.

In addition to freezing the special module, the following domain rules were established with the developer feedback:

- Classes from *Mid-level* modules should not be moved to *High-level* modules.
- Classes from *High-level* modules should not be moved to *Low-level* modules.

Iteration 3

The results of the last iteration can be seen in Table D.3. An overview of the domain rules resulting from this case study are shown in Table 6.4. At the request of Adyen for confidentiality, the specific module names and types are left out.

Accepted: Change group G3 consisted of a single move-class suggestion that the developers accepted.

Change group G5 consisted of seven move-class suggestions that the developers accepted. Four of the move-class suggestions did not change any rule violation penalty. Two of the move-class suggestions reduced unwanted dependencies between certain module types, while the last move-class suggestion replaced one unwanted dependency between certain module types for another unwanted dependency between different module types, thus resulting in a net-zero change for the penalty. The developers indicated that this last move-class suggestion was still acceptable as a temporary solution before the last class got moved to a different module, which was not yet present in the codebase.

Change group G7 consisted of two move-class suggestions that the developers accepted.

Partially accepted: No change groups were partially accepted by the developers.

Rejected: The move-class suggestions in change groups G1 and G6 were rejected because the developers determined the classes to be in the correct module. The module these classes were located in was also of a special type, similar to the one mentioned in the last iteration. Therefore, the freeze for the specific special module from the last iteration was extended to all similar modules, leaving these modules out of consideration for future runs of the algorithm.

The move-class suggestions in change groups G2, G4 and G8 were rejected because the developers determined the classes to be in the correct module, regardless of the code structure quality improvement resulting from the moves.

Interpretation of the expert developers: Eight change groups were discussed, totalling 21 move-class suggestions. The developers outright accepted three change groups containing ten suggestions. The developers rejected the remaining 11 suggestions for two reasons. First, the six classes in change groups G1 and G6 were rejected because the modules these classes were in are of similar special nature as the module mentioned in the previous iteration. Therefore, the freeze was extended to all modules of such a nature to exclude them from consideration for future runs. The last five classes were rejected because the developers determined these classes to be in the correct modules.

From the feedback of the last iteration, no new domain rules could be established. The freeze for the special module from the last iteration was extended to all similar modules.

6.3.2 Case study 2

Iteration 1

An overview of the change groups discussed in this iteration can be found in Table D.4.

Accepted: No change groups were accepted by the developers.

Partially accepted: No change groups were partially accepted by the developers.

Rejected: The move-class suggestions in change groups G1, G2 and G4 were rejected because they would introduce unwanted dependencies between modules of various types.

The suggestions in change group G3 were rejected because this would move classes from *High-level* modules to a *Mid-level* module, which the developers did not want.

Interpretation of the expert developers: The four change groups totalled 20 move-class suggestions, which were all rejected by the developers, regardless of the code structure quality resulting from the moves. The move-class suggestions in change groups G1, G2 and G4 would introduce unwanted dependencies between *High-level* modules, while those in G3 would move classes from *High-level* modules to a *Mid-level* module, which is also unwanted from the perspective of the developers.

From the feedback on the rejected move-class suggestions, the following domain rules were formulated with the developers:

- Multiple module-to-module dependencies are not allowed based on their module types, e.g. dependencies between *High-level* modules.
- Classes that are originally in specific *High-level* modules may not be moved to specific *Mid-level* modules.

Iteration 2

Table D.5 shows the change groups discussed during the second iteration.

Accepted: All 11 move-class suggestions in change groups G3, G7, G10 and G11 were accepted by the developers.

Interestingly, G10 is the only change group that is accepted even though increased the Rule penalty. This is because moving these classes would decrease the number of dependencies between *High-level* modules while increasing the number of dependencies between *Mid-level* modules. The developers indicated that decreasing the number of dependencies from *High-level* modules to one specific *High-level* module was more important than the increase in dependencies between *Mid-level* modules, which is why they accepted the move-class suggestions.

Partially accepted: The developers confirmed that the class in G1 should be moved but disagreed with the suggested module. Instead, the class should be moved to a module that did not yet exist in the codebase.

Rejected: The move-class suggestions in change groups G2, G5 and G9 were rejected because classes from the modules they were in should not be moved to *Mid-level* modules, according to the developers.

The move-class suggestion in G4 was rejected because it violated more domain rules.

The move-class suggestion in change group G6 were rejected because the classes are in a unique module. The developers indicated that classes in this unique module should be left out of consideration for future runs.

The move-class suggestions in change group G8 were rejected because classes from that module should not be moved to *Low-level* modules.

Interpretation of the expert developers: Change group G1 was partially accepted because while the developers agreed that the class should be moved, the suggested module was wrong. The developers explained that the class should be moved to a module they were planning on creating in the near future. Thus they did not want to move the class until that module existed.

The developers outright accepted the 11 move-class suggestions in change groups G3, G7, G10 and G11. Interesting to note is that the move-class suggestions in G10 actually increased the total Rule penalty. This is because moving these classes would result in a decrease in the number of dependencies between *High-level* modules while increasing the number of dependencies between *Mid-level* modules. The developers indicated that the decreasing the number of dependencies from *High-level* modules to one specific *High-level* module was more important than the increase in dependencies between *Mid-level* modules, which is why they accepted the move-class suggestions despite the overall increase in Rule penalty.

According to the developers, the five move-class suggestions in G2, G5 and G9 were rejected because classes from the modules they were in should not be moved to Mid-level modules.

Change group G4 was rejected because it would violate more domain rules.

The move-class suggestion in change group G6 were rejected because the classes are in a unique module. The developers indicated that classes in this unique module should be left out of consideration for future runs.

The move-class suggestions in change group G8 were rejected because classes from the modules they were in should not be moved to *Low-level* modules.

The unique module was totally frozen for future runs. The developers indicated that they wanted to reduce dependencies from their *High-level* modules to one specific module. In order to prioritise removing these unwanted dependencies, this domain rule was assigned with an arbitrary high penalty of 1000. Additionally, two more domain rules were defined based on the developer feedback on the rejected move-class suggestions:

- Dependencies from *High-level* modules to a specific *High-level* module are not allowed, with a penalty of 1000 per violation.
- Classes that are originally in a specific module should not be moved to *Mid-level* modules.
- Classes that are originally in a specific module should not be moved to *Low-level* modules.

Iteration 3

Table D.6 shows the change groups discussed during this last iteration. An overview of the domain rules resulting from this case study are shown in Table 6.5. At the request of Adyen for confidentiality, the specific module names and types are left out.

Accepted: All six move-class suggestions in change groups G4, G7 and G8 were accepted by the developers.

Partially accepted: For all the classes suggested to be moved in change groups G1, G3, G5, G9 and G10, the developers confirmed that the classes should be moved but disagreed with the suggested modules. For each class, the developers could name a better module in the codebase the classes should be moved to, except the class in G10, for which the module did not yet exist.

For the three classes in change groups G2 and G11, the developers confirmed they should be moved, but they disagreed with the suggested modules, as those were *Mid-level* modules. The developers noted that classes from specific *High-level* modules should never be moved to the suggested *Mid-level* modules.

Rejected: The move-class suggestion in change group G6 was rejected because the class was determined to be in the correct module. Additionally, the suggested module was a *Mid-level* module, and the classes in this *High-level* module should never be moved to this *Mid-level* module.

Interpretation of the expert developers: The developers outright accepted the six move-class suggestions in change groups G4, G7 and G8.

The developers confirmed that the eight classes in G1, G3, G5, G9 and G10 should be moved but disagreed with the suggested modules. For each class, except the one in G10, the developer could name an existing module more suitable than the suggested module. For the class in G10, the developers explained that the class should be moved to a module that they were planning on creating in the near future. Thus they did not want to move the class until that module existed.

Similarly, the developers agreed that the three classes in G2 and G11 should be moved. However, they disagreed with the suggested modules, as the suggestions would move classes from *High-level* modules to *Mid-level* modules, which the developers indicated should not happen.

The move-class suggestion in G6 was rejected because the class was in the correct module. Additionally, the developers explained that the suggestion would move a class from a *Mid-level* module to a *High-level* module, which should not happen.

From the developer feedback on the partially accepted and rejected move-class suggestions, the following domain rule was formulated for several different *High-level* and *Mid-level* modules:

- Classes in specific *High-level* modules may not be moved to specific *Mid-level* modules.

6.3.3 Case study 3

Iteration 1

For the first iteration of this case study, the change groups can be found in Table D.7.

Accepted: No change groups were accepted by the developers.

Partially accepted: Change group G2 suggested moving a class to a *High-level* module. The developers confirmed that the class could be moved out of the current module but rejected the suggestion because the move would introduce dependencies between *High-level* modules.

Change groups G3 and G4 both suggested moving a class from a *Mid-level* module to a *Mid-level* module. The developers confirmed that the classes could be moved out of their current module. However, they rejected the suggestion because moving it to the suggested module would introduce unwanted dependencies between specific *Mid-level* modules.

Rejected: Change group G1 suggested moving a class from a *Mid-level* module to a *High-level* module. The developers rejected this move because classes should not be moved from a *Mid-level* module to *High-level* modules, regardless of the code structure quality changes.

The developers rejected the move-class suggestion in change group G5 because it was located in a sub-module where the developers were confident all classes in that sub-module were at the correct location, stating that none of the classes in that sub-module had to be considered for moving.

Interpretation of the expert developers: For several of the rejected classes, the developers could name a different module that would be more suitable. However, the algorithm could not have picked those modules because it only picks related modules in terms of dependencies of the class. For the four move-class suggestions in change groups G1, G2, G3 and G4, the developers agreed that the classes should be moved, but to different modules than suggested. Moving the classes to their suggested modules would introduce various unwanted dependencies between various module types.

The class in G5 is located in a sub-module, where for all classes, the developers were confident were in the correct place. Therefore, all classes in that sub-module could be frozen for future runs.

In addition to freezing the specific sub-module, various domain rules were formulated with the developers:

- Multiple module-to-module dependencies are not allowed based on their module types, e.g. dependencies between *High-level* modules.
- Classes that are originally in modules of a specific type may not be moved to modules of a certain different type, e.g. classes originally in a specific *Mid-level* module may not be moved to a *High-level* module.

Iteration 2

Table D.8 shows the change groups discussed during the second iteration.

Accepted: Change group G2 contained a single move-class suggestion that the developers accepted.

Partially accepted: Change groups G1 and G4 both contained a single move-class suggestion that the developers rejected because of the suggested module. However, they confirmed that the class itself was suitable to be moved.

Rejected: Change group G3 consisted of two suggestions. The developers rejected the first suggestion, as the class was determined to be in the correct module. The developers were conflicted about the second suggestion. They agreed that the class could belong to the new module. However, they still rejected the suggestion because the module the class was originally in was of a unique nature, which requires a different approach than the other modules owned by the team. It was concluded that this specific module was best left out of consideration for future runs.

Interpretation of the expert developers: The developers accepted the move-class suggestion in G2, while for change groups G1 and G4, the developers confirmed that the classes should be moved, but to a different module than suggested. Unfortunately, the modules that the developers suggested were not among the related modules for those classes. Hence, the algorithm could not have picked these modules.

The first move-class suggestion in G3 was rejected because the developers determined it to be in the correct module, regardless of the decrease in Rule penalty caused by the move. The second class is located in a unique module that does not follow the general domain rules of the other modules the team owns. The developers noted that this module was best left out of consideration for future runs.

Besides freezing the unique module, no additional domain rules were established from the developer feedback in this iteration.

Iteration 3

The change groups discussed in the last iteration can be found in Table D.9. An overview of the domain rules resulting from this case study are shown in Table 6.6. At the request of Adyen for confidentiality, the specific module names and types are left out.

Accepted: All move-class suggestions in change groups G1, G2, G3 and G9 were accepted by the developers. Interestingly, the move-class suggestion in G2 actually violated a domain rule, but the developers still found the move beneficial overall.

Partially accepted: For both classes in change groups G4 and G5, the developers agreed that they were suitable for moving but disagreed on the suggested modules.

Change group G7 totalled three move-class suggestions. The developers accepted one of these move-class suggestions, which accounted for the improvement in IntraMD Cohesion. However, the other two move-class suggestions were rejected because they caused an increase in Rule penalty.

Rejected: Both move-class suggestions in change groups G6 and G8 were rejected because the developers determined the classes to be in the correct module, regardless of the code structure quality improvement resulting from the moves.

Although the move-class suggestion in G8 increased Rule penalty, the developers noted that an exception must be made to allow specific *Mid-level* modules to depend on several *Mid-level* modules.

Iteration	Accepted	Partially accepted	Rejected
1	0 / 4	0 / 4	4 / 4
2	0 / 6	4 / 6	2 / 6
3	3 / 8	0 / 8	5 / 8

Table 6.1: Case study 1 change group acceptance overview

Iteration	Accepted	Partially accepted	Rejected
1	0 / 4	0 / 4	4 / 4
2	4 / 11	1 / 11	6 / 11
3	3 / 11	7 / 11	1 / 11

Table 6.2: Case study 2 change group acceptance overview

Iteration	Accepted	Partially accepted	Rejected
1	0 / 5	3 / 5	2 / 5
2	1 / 4	2 / 4	1 / 4
3	4 / 9	3 / 9	2 / 9

Table 6.3: Case study 3 change group acceptance overview

Interpretation of the expert developers: All four move-class suggestions in change groups G1, G2, G3 and G9 were accepted by the developers. Interestingly, the move-class suggestion in G2 actually violated a domain rule, but the developers still found the move beneficial overall.

The developers deemed the two classes in G4 and G5 suitable to be moved but suggested different modules. However, the algorithm could not have picked those modules because it only picks related modules in terms of dependencies of the class. Additionally, the developers accepted one of the three move-class suggestions in G7, which improved IntraMD Cohesion. The other two suggestions were rejected because they increased Rule penalty.

Both move-class suggestions in change groups G6 and G8 were rejected because the developers determined the classes to be in the correct module, regardless of the code structure quality improvement resulting from the moves. Although the move-class suggestion in G8 increased Rule penalty, the developers noted that an exception must be made to allow *Mid-level* modules to depend on several *Mid-level* modules.

From the developer feedback on the partially accepted and rejected move-class suggestions, an exception to a domain rule was made:

- *Mid-level* modules are allowed to depend on a subset of *Mid-level* modules.

Domain rule	Count
Modules of type ... cannot depend on modules of type ...	11
Classes in module ... cannot be moved to module ...	2

Table 6.4: Case study 1 domain rule overview

Domain rule	Count
Modules of type ... cannot depend on modules of type ...	11
Classes in module ... cannot be moved to modules ...	3
Modules ... cannot depend on module ...	1

Table 6.5: Case study 2 domain rule overview

Domain rule	Count
Modules of type ... cannot depend on modules of type ...	11
Exception: Modules of type ... can depend on specific modules of type ...	1
Classes in modules with type ... cannot be moved to modules with type ...	4

Table 6.6: Case study 3 domain rule overview

6.4 Summary of the observations

Observation 9: Developers wanted to know the fitness of individual move-class suggestions. When change groups consisted of more than one move-class suggestion, developers wanted to know how each suggestion affected the metrics. This was implemented after the first iteration of each case study, allowing developers to understand the effect of individual move-class suggestions in change groups for the second and third iterations.

Observation 10: None of the suggestions in the first iterations were accepted. For all three case studies, the developers either rejected the suggestions entirely because the classes were either in the correct module or moving them to the suggested module violated domain rules or partially accepted the suggestions if they disagreed with the suggested module.

Observation 11: There exists an overlap in the domain rules for the three teams. While each team had specific domain rules that applied only to the modules they owned, several domain rules applied for the modules of all the teams and possibly even for other teams at Adyen. The domain rules resulting from each case study are shown in Tables 6.4 to 6.6. The 11 domain rules *Modules of type ... cannot depend on modules of type ...* were applicable to modules in all three case studies. However, the remaining domain rules in the tables were specific to the modules of their respective case study.

Observation 12: Each team froze one or more modules during their case study. In every case study, developers proposed to leave the special modules out of consideration for future runs, as these modules served a distinct purpose outside the regular domain of the team. The domain rules could have been adjusted to fit these special modules, but the developers chose to freeze them instead. Additionally, a specific sub-module was frozen by the developers of **CS3**, as they were highly confident that all classes in that sub-module were in the correct place, so the classes in that sub-module should be left out of consideration by the algorithm.

Observation 13: Future plans influence how developers assess move-class suggestions. During the last iteration of **CS2**, the developers rejected a move-class suggestion because they were planning to create a module in the near future in which the class would belong. To avoid moving the class twice in a short time, they rejected the move until the module was created. Similarly, during the last iteration of **CS1**, one move-class suggestion was accepted as a temporary solution before that class got moved to a module that was already planned but also not yet present in the codebase.

Observation 14: Developers prioritise different domain rules at different times. During the second iteration of **CS2**, the developers accepted a change group that increased Rule penalty. Moving the classes to the suggested modules would reduce the number of unwanted dependencies between *High-level* modules, while increasing the number of unwanted dependencies between *Mid-level* modules. The developers expressed that their priority at the moment was to reduce unwanted dependencies from certain *High-level* modules to a specific *High-level* module, even if this violated additional other domain rules. By introducing a new domain rule for dependencies from these *High-level* modules to the specific *High-level* module, this priority of the developers was implied by the high penalty associated with violations, allowing the algorithm to prioritise removing these dependencies.

Observation 15: The algorithm cannot always suggest the correct module from the dependencies of a class. In several cases, developers rejected the suggested module for move-class suggestions and explained which modules were more suitable for the classes to be moved to. While in a few instances, these modules did not exist in the codebase, often, these modules were already present. From inspecting the related modules of these classes in terms of the dependencies, it was found that the algorithm often could not have suggested the modules the developers mentioned, as those were not among the related modules of the class.

Observation 16: Improving IntraMD Cohesion, InterMD Coupling and Rule penalty is not sufficient for developer acceptance. Sometimes classes were just in the correct module based on their functionality, regardless of the metric improvements moving that class would bring. The improvements of the metrics are not sufficient for developers to accept the changes. Even for Rule penalty, developers rejected change groups that lowered the number of violations and accepted change groups that increased the number of violations.

This shows that sometimes the classes that violate domain rules or negatively impact code structure quality are still in the right place.

Observation 17: A limited number of domain rules and freezes is sufficient to positively improve the suggestions by the algorithm. For each team, about a dozen domain rules and freezes of various granularity were established during the case studies, which were sufficient to generate move class suggestions that are in line with developer knowledge while improving code structure quality, ignoring irrelevant classes and modules, and decreasing the number of violated domain rules. Tables 6.4 to 6.6 show that 13, 15 and 16 domain rules were established for case studies 1, 2 and 3, respectively.

Observation 18: In general, developers dislike move-class suggestions that increase the number of violated domain rules. With the exception of two change groups, all move-class suggestions that increased Rule penalty were rejected by the developers. For **CS2**, the developers accepted the net increase because they prioritised the decreased violation of one domain rule over the increase of another. For **CS3**, the developers thought the move was beneficial, even though it violated more domain rules. The remaining accepted or partially accepted suggestions decreased or did not affect Rule penalty.

Observation 19: The domain rules positively affect the developer acceptance for the change groups. Tables 6.1 to 6.3 show an overview of the number of accepted, partially accepted and rejected change groups for their respective case study. During each first iteration, none of the change groups were outright accepted and instead only partially accepted or rejected. Introducing domain rules after these first iterations improved the developer acceptance for the change groups.

Chapter 7

Discussion

This chapter discusses the implications of the results for the empirical study and case studies. Additionally, we discuss how the approach can be put to practical use. Lastly, we list the threats to validity of this thesis.

7.1 Implications

This section lists the implications drawn from the results shown in Chapters 5 and 6 of the empirical study and the case studies.

The first implication is that the approach scales to enterprise-level software systems. The algorithm was run for 24 and 72 hours during the empirical study to analyse its results, which showed that the algorithm could significantly improve the codebase in terms of the code quality metrics. However, even the 12 hour runs for the case studies show that the algorithm can suggest remodularisations for large codebases. Interviewing the developers on the algorithm results during the case studies led to capturing their domain knowledge, which could be used for future algorithm runs. Each interview took less than 30 minutes, showing that the developer effort needed to create a knowledge base is limited and can be done iteratively over multiple sessions.

The second implication is that the captured developer knowledge is both scalable and reusable. For each case study, over the course of three iterations, about a dozen domain rules and freezes were established, which can be reused for future runs and proved sufficient in guiding the algorithm to create remodularisations that not only improved the code structure quality but also made sense from the perspective of developers. Furthermore, each case study started without any domain knowledge and with developers' low acceptance of change groups, which improved throughout the case study as more domain knowledge was established and used as input for the algorithm.

The last implication is that it is crucial to validate the algorithm results with developers. As stated by Praditwong et al. [62], software modularisation research often does not validate suggestions from the developers' perspective, only relying on the improvements on paper to determine the validity of the research. While the potential improvements in terms of the code structure quality discussed at the end of the empirical study in Chapter 5 sound promising, the importance of considering the domain knowledge of developers becomes clear from the first iterations for each case study, as shown in Chapter 6, where no suggestions were accepted outright. The results also imply that improvements in any of the metrics, even the domain rule penalty violation metric, are not sufficient for developers to always accept the suggestions. It might partially be attributed to the chosen metrics for cohesion (IntraMD Coupling) and coupling (InterMD Coupling), but further research is required to analyse this.

7.2 Practical use of the approach

Having validated the approach's potential during the case studies in Chapter 6, we now discuss its practical uses from a business perspective. The algorithm could be applied to various parts of a codebase, running as a nightly or weekly job to suggest remodularisations for a select group of modules, similar to the case studies, which would require teams to determine their own freezes and domain rules to guide the algorithm. When a team detects that a move-class suggestion violates existing domain knowledge or new domain knowledge has been established, they can add or adjust a domain rule to their knowledge base. Iteratively building their knowledge base improves future algorithm runs, resulting in better suggestions. This approach may require additional effort to split solutions containing dozens of class move suggestions into smaller, more easily understood groups.

Another application for the algorithm could be as a recommendation tool for shrinking large modules. For example, the objective function *Largest module*, discussed in Section 4.4.6, helps prioritise the algorithm to reduce the size of large modules. By configuring the objective function to focus on specific modules that a team wants to reduce in size, the algorithm can then suggest remodularisations that focus on shrinking those modules.

7.3 Threats to validity

This section acknowledges the threats to internal, construct and external validity. For internal validity, the following threats have been identified:

- For the parameter tuning in Section 5.3, we did not explore every combination of parameter values and did not trying every possible parameter value. This is because exploring all combinations and values would require an immense amount of runs and, therefore, time, making such an optimisation infeasible. Therefore while the chosen combination of parameter values is valid, it cannot be said with certainty that it is optimal, as this would require more extensive parameter tuning, which may improve the algorithm's performance.

- The change groups used during the case studies in Chapter 6 are not entirely representative of the solutions that resulted from the algorithm because of the filtering and the grouping of suggestions by module destination. Filtering and grouping were done to ensure interviews could be kept under 30 minutes. However, as a drawback, the change groups' acceptance rate might differ from that of the original solutions.

- Interviews and the interviewees are prone to bias. For each change group and each individual move-class suggestion, the developers were aware of the effect on the various metrics, which may have caused developers to judge the suggestions positively, as they were aware of the positive effects of the suggestions. However, from the results of the case studies, we found that improvements in the metrics were not sufficient for developers to always accept suggestions.

For construct validity, we have identified the following threats:

- We expressed *Change impact* as the number of class moves in solutions. However, this measure does not consider that some classes are more challenging to move than others. Therefore, a different measure would be more appropriate for estimating the amount of work a solution requires.
- The domain knowledge of developers was captured using freezes and domain rules. While this proved effective when the domain knowledge could be formulated as such, not all domain knowledge could be defined by freezes and rules. For example, modules that did not yet exist in the codebase but were planned for by the developers influence how the developers perceive suggestions. However, these modules could not be accounted for in terms of domain rules.

Lastly, the threat to external validity is that the approach in this thesis was applied to the codebase of one company. Therefore it cannot be said with certainty that the results observed for this codebase generalise to other codebases. To analyse whether the results generalise, more research is required that involves various factors, such as different developers, other programming languages or systems in general.

Chapter 8

Conclusions and Future Work

This chapter starts by stating the goal of the thesis, after which we draw conclusions from the results. Lastly, we discuss potential future work.

In this thesis, we proposed an approach to the software remodularisation problem. This approach to suggest move class refactorings that improve the modularisation of a codebase from a metrics point of view, while also considering the domain knowledge that developers have. While there exist many approaches to the software remodularisation problem, these often do consider the domain knowledge of developers and only account for code quality metrics such as coupling and cohesion. In addition to this, the results of these approaches are often not validated by developers that are familiar with the software system in question. This results in plenty of research that works well in theory but misses the practical validation to inspire confidence that the approach works in a real-world application. The approach in this thesis is not only validated by several development teams of an enterprise-level large-scale codebase, the feedback and domain knowledge of these developers is also used to improve future remodularisations suggested by the NSGA-III algorithm.

This domain knowledge is expressed in the form of freezes and domain rules, varying in granularity from being specific to a single class up to covering multiple modules. This limits the effort required to express the domain knowledge of developers to improve the remodularisation suggestions. The freezes and domain rules are also reusable, allowing developers to iteratively build a knowledge base consisting of a set of freezes and rules. As a result, this knowledge base becomes more in line with their knowledge over time. Additionally, it can be adjusted when needed, for example, after a large-scale refactoring that calls for different domain rules to govern the codebase.

Before performing case studies with different Adyen developer teams, the algorithm was run on the entire codebase for an empirical study to understand how well the algorithm performs from a code quality metric perspective and whether it converges. The results showed that the algorithm converged in 50,000 generations for *IntraMD Cohesion*, while it did not converge for *InterMD Coupling* and *Largest module size*. As the estimated number of circular module dependencies was used as an objective value, there were generations where every solution would result in cycles in the module dependency graph. In terms of *IntraMD Cohesion*, *InterMD Coupling* and *Largest module size*, the algorithm made a significant relative impact of $+8.60\%$, -6.67% and -1.94%, respectively, considering the best solutions that improved all three metrics. Considering all solutions, the average solution improved *InterMD Coupling* and *Largest module size* at the cost of worsening *IntraMD Cohesion*.

After the empirical study, we performed three case studies involving different teams and parts of the codebase. Each case study started by running the algorithm without any defined domain knowledge, only freezing the codebase such that the algorithm considers the modules related to the team. To improve the understandability of the move-class suggestions, solutions were split into smaller groups called change groups, as the original solutions consisted of dozens of suggestions. Each case study consisted of three iterations and accompanying runs of the algorithm to validate the algorithm's suggestions and determine the developers' domain knowledge. For each of the initial iterations, none of the change groups were accepted by the developers, and they all resulted in several domain rules. These domain rules were then used as input for future runs, allowing the algorithm to suggest *move-class* refactorings that do not violate the domain rules of developers or even reduce the number of violations currently in the codebase.

The results of these case studies show that including the domain of the developers positively improves the acceptance rate of the change groups, thus improving the algorithm's performance. The most prevalent reasons for developers to reject the change groups are either because the class is already in the correct module, regardless of the code quality improvement resulting from moving the class, or because moving the class would violate not-yet-specified domain rules. In the latter case, new domain rules can be established to improve future suggestions, while the former case requires some method to detect whether classes are in their correct module already, as metric improvements alone are not always sufficient reason for developers to accept changes.

The contribution of this thesis is an approach for the software remodularisation problem that improves the code structure quality of large-scale codebases in addition to adhering to developer domain knowledge, having validated the results of the approach by the developers of the Adyen codebase.

The results and threats to validity of this thesis show potential for future work. First, while we showed the potential of the approach and the algorithm, one must consider that the results were obtained on a single codebase owned by one company. Therefore, the approach must be applied to various other codebases to understand how well it generalises beyond the codebase of Adyen. For example, one might consider using large-scale open-source software systems to include in their analysis, submitting merge requests according to the move class suggestions of the algorithm. Another interesting subject for future work is investigating the convergence of the algorithm further, similar to the empirical study. For example, the algorithm seemed to converge quickly for the *IntraMD Cohesion* metric, but given the low number of runs performed during that study, this may be attributed to chance. Performing a more thorough analysis would improve understanding of the algorithm's convergence for all metrics.

In terms of the metrics, future research can look into other measures for coupling and cohesion than the ones used in this thesis. While we concluded that improvements of the metrics are not sufficient for developers to always accept the move-class suggestions of the algorithm, there might exist other metrics where improvements would suffice. Therefore we recommend researching a broader selection of code structure quality metrics and their impact on developer acceptance.

With regards to the mutation operator of the algorithm, we make several suggestions for future work. In its current form, the mutation operator moves a single class to a related module. However, with the size of modules in a large-scale codebase, several classes in a module often operate together. Moving such a cluster of classes elsewhere through individual class moves can take a significant number of generations or may not even happen as the intermediate moves are worse in terms of the optimisation variables until the entire cluster is moved. Instead, the mutation operator should be able to move more than one class by determining which classes in a module strongly belong together and should be moved together.

If the algorithm can move larger groups of classes, it is also recommended to allow the algorithm to create new modules. During this thesis, the algorithm could not create new modules in the codebase, as this would often result in moving a single class to a new module, which results in modules that are too granular for large-scale codebases. The last recommendation regarding the mutation operator is to consider removing the *Circular module dependencies* objective function and instead completely prevent the algorithm from introducing cycles in the module dependency graph through mutation. This would require additional effort during the crossover operation, as this step may also introduce cycles. Adjusting these operators would result in every solution of the algorithm not containing circular dependencies between modules and, therefore, could always be implemented without additional work required of the developers.

While the number of classes moved by a solution is used as a minimisation objective function, it is common for solutions to consist of dozens of move-class suggestions. One of the goals of a remodularisation approach is to limit the number of classes to be moved so developers can better understand the suggestions. During the case studies in this thesis, we grouped the classes in the solutions based on their suggested module location. For future work, we recommend looking into other ways to present solutions to developers without splitting them, so the remodularisation is in line with the efforts of the algorithm, rather than an approximation such as the change groups used by the case studies.

During the mutation step of the algorithm, when a class is chosen to be moved, a related module is chosen as its destination. This module either contains a class that depends on the class to be moved, or it contains a class that the class to be moved depends on. During the case studies, developers often disagreed with the suggested module for classes in the partially accepted change groups. In most cases, the developers could name a module in the codebase that would be a more appropriate destination for these classes. However, these modules were rarely among the related modules of the class in terms of its dependencies. Therefore, our recommendation is to improve the step of the algorithm that picks a related module. For example, through semantic analysis of the source code, like the approach of Mahouachi [45], classes and modules that are semantically similar but not related through dependencies can be considered during this step, allowing the algorithm to better suggest to which module a class should be moved.

The last recommendation for future work concerns the domain knowledge of developers. During the case studies, some change groups were partially accepted or rejected because the developers were aware of planned refactorings concerning the classes to be moved, therefore not wanting to move the classes at the current time. For example, in several cases, a new module was planned as the destination for these classes, but this module was not yet present in the codebase. As the domain knowledge of developers is expressed in the form of rules, it is currently not possible to account for such planned refactorings. Therefore, our recommendation is to extend the domain knowledge representation of the algorithm to allow for domain knowledge that is not easily captured using rules.

Bibliography

[1] Fernando Brito Abreu and Rogério Carapuça. Object-oriented software engineering: Measuring and controlling the development process. In *Proceedings of the 4th international conference on software quality*, volume 186, 1994.

[2] Charu C Aggarwal, Alexander Hinneburg, and Daniel A Keim. On the surprising behavior of distance metrics in high dimensional space. In *International conference on database theory*, pages 420–434. Springer, 2001.

[3] Vahid Alizadeh, Marouane Kessentini, Mohamed Wiem Mkaouer, Mel Ocinneide, Ali Ouni, and Yuanfang Cai. An interactive and dynamic search-based approach to software refactoring recommendations. *IEEE Transactions on Software Engineering*, 46(9):932–961, 2018.

[4] Edward B Allen and Taghi M Khoshgoftaar. Measuring coupling and cohesion: An information-theory approach. In *Proceedings Sixth International Software Metrics Symposium (Cat. No. PR00403)*, pages 119–127. IEEE, 1999.

[5] Edward B Allen, Taghi M Khoshgoftaar, and Yan Chen. Measuring coupling and cohesion of software modules: an information-theory approach. In *Proceedings Seventh International Software Metrics Symposium*, pages 124–134. IEEE, 2001.

[6] Periklis Andritsos and Vassilios Tzerpos. Information-theoretic software clustering. *IEEE Transactions on Software Engineering*, 31(2):150–165, 2005.

[7] Nicolas Anquetil and Timothy C Lethbridge. Experiments with clustering as a software remodularization method. In *Sixth Working Conference on Reverse Engineering (Cat. No. PR00303)*, pages 235–255. IEEE, 1999.

[8] Victor R Basili, Lionel C. Briand, and Walcélio L Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on software engineering*, 22(10):751–761, 1996.

[9] Gabriele Bavota, Filomena Carnevale, Andrea De Lucia, Massimiliano Di Penta, and Rocco Oliveto. Putting the developer in-the-loop: an interactive ga for software remodularization. In *International Symposium on Search Based Software Engineering*, pages 75–89. Springer, 2012.

[10] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. Using structural and semantic measures to improve software modularization. *Empirical Software Engineering*, 18(5):901–932, 2013.

[11] Doug Bell, Ian Morrey, and John Pugh. *Software engineering: A programming approach*. Prentice Hall International (UK) Ltd., 1987.

[12] Sabrine Boukharata, Ali Ouni, Marouane Kessentini, Salah Bouktif, and Hanzhang Wang. Improving web service interfaces modularity using multi-objective optimization. *Automated Software Engineering*, 26(2):275–312, 2019.

[13] Lionel Briand, Prem Devanbu, and Walcelio Melo. An investigation into coupling measures for c++. In *Proceedings of the 19th international conference on Software engineering*, pages 412–421, 1997.

[14] Lionel C Briand, John W Daly, and Jürgen Wüst. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering*, 3(1):65–117, 1998.

[15] Ivan Candela, Gabriele Bavota, Barbara Russo, and Rocco Oliveto. Using cohesion and coupling for software remodularization: Is it enough? *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 25(3):1–28, 2016.

[16] Jitender Kumar Chhabra et al. Improving modular structure of software system using structural and lexical dependency. *Information and software Technology*, 82:96–120, 2017.

[17] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[18] Larry L Constantine and Tom O Barnett. *Modular programming: Proceedings of a national symposium*. Information & systems Institute, 1968.

[19] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[20] Steve Counsell, Stephen Swift, Allan Tucker, and Emilia Mendes. Object-oriented cohesion subjectivity amongst experienced and novice developers: an empirical study. *ACM SIGSOFT Software Engineering Notes*, 31(5):1–10, 2006.

[21] Ward Cunningham. The wycash portfolio management system. *ACM SIGPLAN OOPS Messenger*, 4(2):29–30, 1992.

[22] Indraneel Das and John E Dennis. Normal-boundary intersection: A new method for generating the pareto surface in nonlinear multicriteria optimization problems. *SIAM journal on optimization*, 8(3):631–657, 1998.

[23] Kalyanmoy Deb and Himanshu 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, 2013.

[24] Kalyanmoy Deb and J Sundar. Reference point based multi-objective optimization using evolutionary algorithms. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 635–642, 2006.

[25] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[26] Kalyanmoy Deb, D Saxena, et al. Searching for pareto-optimal solutions through dimensionality reduction for certain large-dimensional multi-objective optimization problems. In *Proceedings of the world congress on computational intelligence (WCCI-2006)*, pages 3352–3360, 2006.

[27] Massimiliano Di Penta, Markus Neteler, Giuliano Antoniol, and Ettore Merlo. A language-independent software renovation framework. *Journal of Systems and Software*, 77(3):225–240, 2005.

[28] Len Erlikh. Leveraging legacy system dollars for e-business. *IT professional*, 2(3): 17–23, 2000.

[29] Letha H Etzkorn, Sampson E Gholston, Julie L Fortune, Cara E Stein, Dawn Utley, Phillip A Farrington, and Glenn W Cox. A comparison of cohesion metrics for object-oriented systems. *Information and Software Technology*, 46(10):677–687, 2004.

[30] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[31] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2012.

[32] Dunwei Gong, Xinfang Ji, Jing Sun, and Xiaoyan Sun. Interactive evolutionary algorithms with decision-maker's preferences for solving interval multi-objective optimization problems. *Neurocomputing*, 137:241–251, 2014.

[33] Mathew Hall, Neil Walkinshaw, and Phil McMinn. Supervised software modularisation. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 472–481. IEEE, 2012.

[34] Mathew Hall, Neil Walkinshaw, and Phil McMinn. Effectively incorporating expert knowledge in automated software remodularisation. *IEEE Transactions on Software Engineering*, 44(7):613–630, 2018.

[35] Mark Harman, Robert M Hierons, and Mark Proctor. A new representation and crossover operator for search-based optimization of software modularization. In *GECCO*, volume 2, pages 1351–1358, 2002.

[36] Oğuzhan Hasançebi and Fuat Erbatur. Evaluation of crossover techniques in genetic algorithm based optimum structural design. *Computers & Structures*, 78(1-3):435–448, 2000.

[37] Simon Huband, Luigi Barone, Lyndon While, and Phil Hingston. A scalable multi-objective test problem toolkit. In *International Conference on Evolutionary Multi-Criterion Optimization*, pages 280–295. Springer, 2005.

[38] Simon Huband, Philip Hingston, Luigi Barone, and Lyndon While. A review of multiobjective test problems and a scalable test problem toolkit. *IEEE Transactions on Evolutionary Computation*, 10(5):477–506, 2006.

[39] Amin Ibrahim, Shahryar Rahnamayan, Miguel Vargas Martin, and Kalyanmoy Deb. Elitensga-iii: An improved evolutionary many-objective optimization algorithm. In *2016 IEEE Congress on evolutionary computation (CEC)*, pages 973–982. IEEE, 2016.

[40] Himanshu Jain and Kalyanmoy Deb. An evolutionary many-objective optimization algorithm using reference-point based nondominated sorting approach, part ii: Handling constraints and extending to an adaptive approach. *IEEE Transactions on evolutionary computation*, 18(4):602–622, 2013.

[41] Donald B Johnson. Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing*, 4(1):77–84, 1975.

[42] Vineet Khare, Xin Yao, and Kalyanmoy Deb. Performance scaling of multi-objective evolutionary algorithms. In *International conference on evolutionary multi-criterion optimization*, pages 376–390. Springer, 2003.

[43] Y Lee. Measuring the coupling and cohesion of an object-oriented program based on information flow. In *Proc. Int'l Conf. Software Quality, 1995*, 1995.

[44] Alessandra Levcovitz, Ricardo Terra, and Marco Tulio Valente. Towards a technique for extracting microservices from monolithic enterprise systems. *arXiv preprint arXiv:1605.03175*, 2016.

[45] Rim Mahouachi. Search-based cost-effective software remodularization. *Journal of Computer Science and Technology*, 33(6):1320–1336, 2018.

[46] Spiros Mancoridis, Brian S Mitchell, Chris Rorres, Y Chen, and Emden 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)*, pages 45–52. IEEE, 1998.

[47] Spiros Mancoridis, Brian S Mitchell, Yihfarn Chen, and Emden R Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*, pages 50–59. IEEE, 1999.

[48] Onaiza Maqbool and Haroon Babri. Hierarchical clustering for software architecture recovery. *IEEE Transactions on Software Engineering*, 33(11):759–780, 2007.

[49] Onaiza Maqbool and Haroon Atique Babri. The weighted combined algorithm: A linkage algorithm for software clustering. In *Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.*, pages 15–24. IEEE, 2004.

[50] Andrian Marcus, Denys Poshyvanyk, and Rudolf Ferenc. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Transactions on Software Engineering*, 34(2):287–300, 2008.

[51] Robert C Martin, James Newkirk, and Robert S Koss. *Agile software development: principles, patterns, and practices*, volume 2. Prentice Hall Upper Saddle River, NJ, 2003.

[52] Brad L Miller, David E Goldberg, et al. Genetic algorithms, tournament selection, and the effects of noise. *Complex systems*, 9(3):193–212, 1995.

[53] Mohamed W Mkaouer, Marouane Kessentini, Slim Bechikh, and Daniel R Tauritz. Preference-based multi-objective software modelling. In *2013 1st International Workshop on Combining Modelling and Search-Based Software Engineering (CMSBSE)*, pages 61–66. IEEE, 2013.

[54] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Mel Ó Cinnéide, and Kalyanmoy Deb. On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach. *Empirical Software Engineering*, 21(6):2503–2545, 2016.

[55] Wiem Mkaouer, Marouane Kessentini, Adnan Shaout, Patrice Koligheu, Slim Bechikh, Kalyanmoy Deb, and Ali Ouni. Many-objective software remodularization using nsga-iii. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(3):1–45, 2015.

[56] Rashid Naseem, Onaiza Maqbool, and Siraj Muhammad. Cooperative clustering for software modularization. *Journal of Systems and Software*, 86(8):2045–2062, 2013.

[57] Sam Newman. *Monolith to microservices: evolutionary patterns to transform your monolith*. O'Reilly Media, 2019.

[58] Jevgenija Pantiuchina, Michele Lanza, and Gabriele Bavota. Improving code: The (mis) perception of quality metrics. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 80–91. IEEE, 2018.

[59] Thomas M Pigoski. *Practical software maintenance: best practices for managing your software investment*. Wiley Publishing, 1996.

[60] Denys Poshyvanyk, Andrian Marcus, Rudolf Ferenc, and Tibor Gyimóthy. Using information retrieval based coupling measures for impact analysis. *Empirical software engineering*, 14(1):5–32, 2009.

[61] Babak Pourasghar, Habib Izadkhah, Ayaz Isazadeh, and Shahriar Lotfi. A graph-based clustering algorithm for software systems modularization. *Information and Software Technology*, 133:106469, 2021.

[62] Kata Praditwong, Mark Harman, and Xin Yao. Software module clustering as a multi-objective search problem. *IEEE Transactions on Software Engineering*, 37(2):264–282, 2010.

[63] Amarjeet Prajapati and Jitender Kumar Chhabra. A particle swarm optimization-based heuristic for software module clustering problem. *Arabian Journal for Science and Engineering*, 43(12):7083–7094, 2018.

[64] Roger S Pressman. *Software engineering: a practitioner's approach*. Palgrave macmillan, 2005.

[65] Islem Saidani, Ali Ouni, Mohamed Wiem Mkaouer, and Aymen Saied. Towards automated microservices extraction using muti-objective evolutionary search. In *International Conference on Service-Oriented Computing*, pages 58–63. Springer, 2019.

[66] Santonu Sarkar, Avinash C Kak, and Girish Maskeri Rama. Metrics for measuring the quality of modularization of large-scale object-oriented software. *IEEE Transactions on Software Engineering*, 34(5):700–720, 2008.

[67] Santonu Sarkar, Shubha Ramachandran, G Sathish Kumar, Madhu K Iyengar, K Rangarajan, and Saravanan Sivagnanam. Modularization of a large-scale business application: A case study. *IEEE software*, 26(2):28–35, 2009.

[68] Casper Schröder. Expanding search-based software modularization to enterprise-level projects: A case study at adyen. 2020.

[69] Ian Sommerville. Software documentation. *Software engineering*, 2:143–154, 2001.

[70] Miroslaw Staron. *Action research in software engineering*. Springer, 2020.

[71] Wayne P. Stevens, Glenford J. Myers, and Larry L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.

[72] Mechthild Stoer and Frank Wagner. A simple min-cut algorithm. *Journal of the ACM (JACM)*, 44(4):585–591, 1997.

[73] Robert Tarjan. Enumeration of the elementary circuits of a directed graph. *SIAM Journal on Computing*, 2(3):211–216, 1973.

[74] Navid Teymourian, Habib Izadkhah, and Ayaz Isazadeh. A fast clustering algorithm for modularization of large-scale software systems. *IEEE Transactions on Software Engineering*, 2020.

[75] James C Tiernan. An efficient search algorithm to find the elementary circuits of a graph. *Communications of the ACM*, 13(12):722–726, 1970.

[76] Rui Wang, Robin C Purshouse, and Peter J Fleming. Preference-inspired coevolutionary algorithms for many-objective optimization. *IEEE Transactions on Evolutionary Computation*, 17(4):474–494, 2012.

[77] Zhihua Wen and Vassilios Tzerpos. An effectiveness measure for software clustering algorithms. In *Proceedings. 12th IEEE International Workshop on Program Comprehension, 2004.*, pages 194–203. IEEE, 2004.

[78] Theo A Wiggerts. Using clustering algorithms in legacy systems remodularization. In *Proceedings of the Fourth Working Conference on Reverse Engineering*, pages 33–43. IEEE, 1997.

[79] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. Spea2: Improving the strength pareto evolutionary algorithm. *TIK-report*, 103, 2001.

Appendix A

Glossary

In this chapter we list an overview and description of frequently used terms.

Root class: A class that has no other classes in its module that depend on it.

Leaf class: A class that does not depend on other classes in its module.

Coupling: Abstract measure of how dependent modules are on other modules to function. See Section 4.4.3 for the explanation on the InterMD Coupling measure used in this thesis.

Cohesion: Abstract measure of how dependent classes are on other classes in the same module to function. See Section 4.4.4 for the explanation of the IntraMD Cohesion measure used in this thesis.

Modularisation: The allocation of classes in modules, in a software system.

Dependency graph: A graph that depicts the dependencies of software units. A class dependency graph models classes as nodes and class dependencies as edges. A module dependency graph models modules as nodes and module dependencies as edges.

Codebase: The collection of source code of a software system.

Genetic algorithm: Search-based algorithm based on the theory of natural evolution. Models solutions to a problem as individuals in a population, applying mutation and crossover to advance the population in terms of fitness, over time improving how good solutions are in terms of the problem.

Multi-objective genetic algorithm: A genetic algorithm that optimises for more than one objective function. The fitness of solutions is represented using multiple values.

Solution: An individual in the population of the algorithm, represents a solution to the problem the algorithm is applied to. In this thesis, a solution is a remodularisation of the original codebase.

Population: A group of solutions. Used by the algorithm to create new solutions through crossover and mutation, changing over time to improve.

Pareto front: The set of non-dominated solutions in the population.

Non-dominated: Refers to the domination principle explained in Section 2.3.6. A solution is non-dominated if there is no other solution that is at equal or better in terms of all fitness values, and strictly better in at least one.

Change group: A representation of a solution, made smaller to be more comprehensible. Used during the case studies.

Move class refactoring: The action of moving a class in a codebase from one module to another.

Transitive dependency: An indirect dependency of a class on another class, caused by intermediate class dependencies. If class A depends on class B, and B depends on C, then class A transitively depends on C.

Search space: The set of all possible solutions of the algorithm.

Appendix B

Parameter tuning results

This section discusses which parameters can be tuned for the algorithm and how they were tuned to obtain the best results. To tune each parameter, five runs were performed to obtain the best and average results for that parameter, keeping the other parameters at a default value. For the tuned parameters, these were their default values:

- Number of reference points: 10x the number of objective dimensions
- Population size: 1x the number of reference points
- Mutation chance: 50%
- An elite archive is used
- optimise for the number of cycles
- optimise for the largest module size

These are the values the parameters were tuned for, with the optimal values in bold:

- Number of reference points: {5x, **10x**, 20x} the number of objective dimensions
- Population size: {**1x**, 2x, 3x} the number of reference points
- Mutation chance: {5%, 10%, 20%, **50%**, 75%}
- An elite archive {**is**, is not} used
- The number of cycles {**is**, is not} optimised for
- The size of the largest module {**is**, is not} optimised for

During each run, the objective functions for *InterMD Coupling* and *IntraMD Cohesion* were used. Objective functions *Largest module* and *Circular module dependencies* were used except when they were tuned for. The objective function for *Domain rule violations* was not used, as this objective function is evaluated by the case study outlined in Chapter 6. Tables B.1 to B.6 display the results for the tuning process for each parameter. Per request of Adyen, the tables show relative improvement instead of the absolute values. In the remainder of this chapter, we discuss the results of the parameter tuning:

A low number of reference points means a smaller population, which allows the algorithm to complete each generation faster, while a high number means the algorithm can explore more in a generation. As there is no clear optimal choice for this parameter, a balanced trade-off is used. For population size, the differences in the metrics is very small, but the algorithm is significantly slowed down when increasing the population size. For this reason, a population size equal to the number of reference points is used, in line to the approach by Deb and Jain in the original NSGA-III paper [23]. Table B.5 shows that the algorithm performs better in terms of code structure quality when not optimising for the number of cycles. Unfortunately, many of the resulting solutions did contain cycles, meaning they were not usable as-is. Therefore the choice is made to optimise for the number of cycles. Table B.6 shows that the algorithm can perform better when not optimising for the size of the largest module but this comes at the cost of increasing the size of the largest module. For the purpose of the codebase of Adyen this is unwanted, so the size of the largest module is optimised for.

Reference points	Generations	IntraMD Cohesion	InterMD Coupling	Cycles	Largest module
5x	(**58985**; **55830**)	(4.6%; 3.5%)	(-6.14%; -5.34%)	(**0**; **0**)	(**-2.59%**; **-2.59%**)
10x	(31737; 30467)	(**5.4%**; **4.6%**)	(-6.66%; -6.18%)	(**0**; **0**)	(-1.23%; -1.23%)
20x	(19751; 18886)	(5.4%; 4.3%)	(**-7.17%**; **-6.56%**)	(**0**; **0**)	(-1.23%; -1.23%)

Table B.1: Reference points parameter tuning results

Population size	Generations	IntraMD Cohesion	InterMD Coupling	Cycles	Largest module
1x	(**36007**; **33201**)	(**5.4%**; 3.5%)	(-7.15%; -7.42%)	(**0**; **0**)	(**-2.59%**; **-1.23%**)
2x	(18464; 18093)	(**5.4%**; **4.3%**)	(-7.29%; -6.87%)	(**0**; **0**)	(-1.23%; **-1.23%**)
3x	(12959; 12162)	(4.6%; **4.3%**)	(**-7.62%**; **-7.45%**)	(**0**; **0**)	(-1.23%; **-1.23%**)

Table B.2: Population size parameter tuning results

Mutation chance	Generations	IntraMD Cohesion	InterMD Coupling	Cycles	Largest module
0.05	(32072; 31605)	(5.4%; **3.9%**)	(-6.91%; -6.37%)	(**0**; **0**)	(-1.23%; -1.23%)
0.1	(32797; 31832)	(5.4%; **3.9%**)	(-6.78%; -6.38%)	(**0**; **0**)	(-1.23%; -1.23%)
0.2	(33150; 32090)	(4.6%; **3.9%**)	(-6.71%; -6.29%)	(**0**; **0**)	(**-2.59%**; **-2.59%**)
0.5	(**34566**; **32217**)	(**5.7%**; **3.9%**)	(-7.01%; -6.45%)	(**0**; **0**)	(**-2.59%**; -1.23%)
0.75	(34454; 31188)	(3.9%; 3.5%)	(**-7.12%**; **-6.54%**)	(**0**; **0**)	(-1.23%; -1.23%)

Table B.3: Mutation chance parameter tuning results

Use elite archive	Generations	IntraMD Cohesion	InterMD Coupling	Cycles	Largest module
True	(**33839**; **32872**)	(**5.4%**; **4.3%**)	(**-7.62%**; **-7.42%**)	(**0**; **0**)	(**-1.23%**; **-1.23%**)
False	(27839; 26579)	(3.2%; 2.8%)	(-6.28%; -6.06%)	(**0**; **0**)	(**-1.23%**; **-1.23%**)

Table B.4: Use elite archive parameter tuning results

Optimise for cycles	Generations	IntraMD Cohesion	InterMD Coupling	Cycles	Largest module
True	(35200; 33138)	(4.6%; 3.9%)	(-6.66%; -5.97%)	(**0**; **0**)	(**-2.59%**; **-2.59%**)
False	(**40842**; **38853**)	(**5.7%**; **4.6%**)	(**-7.54%**; **-6.96%**)	(**0**; 71)	(-1.23%; 0%)

Table B.5: Optimise for cycles parameter tuning results

Optimise for largest module	Generations	IntraMD Cohesion	InterMD Coupling	Cycles	Largest module
True	(34165; 31095)	(**5.4%**; **4.3%**)	(-6.71%; -5.81%)	(**0**; **0**)	(**-2.59%**; **-2.59%**)
False	(**43738**; **41119**)	(**5.4%**; 3.5%)	(**-6.99%**; **-6.39%**)	(**0**; **0**)	(0%; +1.97%)

Table B.6: Optimise for largest module size parameter tuning results

Appendix C

Case study initial interviews

C.1 Initial developer interview questions

Each developer was asked the following questions:

- Which case study are you part of?
- How many years have you worked as software developer?
- How many of those years have you worked at Adyen?
- How many of those years have you worked with the modules your team is responsible for?
- On a scale of 1 to 10, how well do you understand the module dependency structure of the entire Adyen codebase?
- On a scale of 1 to 10, how well do you understand the module dependency structure of modules your team is responsible for?

Their answers can be found in Table C.1.

Developer	D1	D2	D3	D4	D5	D6	D7
Case study	1	1	2	2	2	3	3
Years as developer	9	2.5	5	4	4	8	1
Years at Adyen	4	0.5	4	3	1	2	1
Years in team	4	0.5	4	3	1	1	1
Overall structure knowledge	7	5	4	6	5	8	6
Team structure knowledge	8	7	7	8	7	8	7

Table C.1: Initial interview developer answers

Appendix D

Case study iterations tables

Solutions have been split into groups according to the suggested module classes. Therefore the sets of move-class suggestion are not referred to as solutions, but change groups in this chapter.

For each table showing the various change groups, *Change impact* refers to the number of classes suggested to be moved by the change group. *IntraMD Cohesion* refers to the relative change in IntraMD cohesion, for which positive values are better. *InterMD Coupling* refers to the relative change in InterMD coupling, for which negative values are better. The percentages shown for *IntraMD Cohesion* and *InterMD Coupling* were rounded to three decimals. *Rule penalty* refers to the change in rule penalty resulting from violating domain rules, for which negative values are better. Lastly, *Accepted* indicates whether developers accepted all class moves suggested by the change group. For *Accepted*, Y means all class moves were accepted, N means all class moves were rejected, and PA means that the class moves were partially accepted by the developers.

D.1 Case study 1

Group	Change impact	IntraMD Cohesion	InterMD Coupling	Rule penalty	Accepted
G1	2	-0.082%	-0.006%	-	N
G2	5	+0.073%	-0.008%	-	N
G3	1	+0.003%	0%	-	N
G4	2	+0.006%	-0.001%	-	N

Table D.1: Case study 1 iteration 1 change groups overview

Group	Change impact	IntraMD Cohesion	InterMD Coupling	Rule penalty	Accepted
G1	3	+0.057%	0%	-47	PA
G2	9	+0.003%	-0.01%	+9	N
G3	1	+0.001%	+0.002%	-25	N
G4	2	+0.006%	-0.001%	+11	PA
G5	3	+0.002%	0%	+7	PA
G6	3	+0.001%	-0.01%	0	PA

Table D.2: Case study 1 iteration 2 change groups overview

Group	Change impact	IntraMD Cohesion	InterMD Coupling	Rule penalty	Accepted
G1	2	-0.353%	-0.01%	-3	N
G2	2	-0.003%	+0.001%	0	N
G3	1	+0.02%	-0.01%	0	Y
G4	1	+0.001%	0%	+4	N
G5	7	-0.003%	-0.01%	-2	Y
G6	4	-0.368%	-0.108%	0	N
G7	2	+0.051%	-0.039%	0	Y
G8	2	-0.001%	0%	-1	N

Table D.3: Case study 1 iteration 3 change groups overview

D.2 Case study 2

Group	Change impact	IntraMD Cohesion	InterMD Coupling	Rule penalty	Accepted
G1	7	+0.01%	+0.006%	-	N
G2	3	+0.073%	0%	-	N
G3	4	+0.053%	0%	-	N
G4	6	+0.016%	-0.009%	-	N

Table D.4: Case study 2 iteration 1 change groups overview

Group	Change impact	IntraMD Cohesion	InterMD Coupling	Rule penalty	Accepted
G1	1	0%	0%	-1	PA
G2	1	0%	-0.001%	-2	N
G3	1	0%	0%	-4	Y
G4	1	+0.024%	+0.189%	+31	N
G5	1	-0.023%	0%	-2	N
G6	1	0%	0%	-5	N
G7	4	-0.001%	0%	-15	Y
G8	3	-0.005%	0%	-6	N
G9	1	-0.023%	0%	-2	N
G10	3	+0.001%	+0.04%	+9	Y
G11	3	-0.006%	0%	-16	Y

Table D.5: Case study 2 iteration 2 change groups overview

Group	Change impact	IntraMD Cohesion	InterMD Coupling	Rule penalty	Accepted
G1	1	0%	0%	-1001	PA
G2	2	0%	+0.029%	-994	PA
G3	2	0%	+0.01%	-999	PA
G4	1	+0.001%	+0.01%	0	Y
G5	3	0%	+0.029%	-5000	PA
G6	1	+0.051%	0%	+1	N
G7	2	-0.002%	0%	-3003	Y
G8	3	+0.001%	+0.029%	-14	Y
G9	1	+0.005%	+0.147%	+17	PA
G10	1	0%	+0.02%	-1001	PA
G11	1	0%	0%	-1001	PA

Table D.6: Case study 2 iteration 3 change groups overview

D.3 Case study 3

Group	Change impact	IntraMD Cohesion	InterMD Coupling	Rule penalty	Accepted
G1	1	+0.003%	+0.06%	-	N
G2	1	+0.001%	+0.01%	-	PA
G3	1	+0.001%	+0.079%	-	PA
G4	1	+0.001%	+0.149%	-	PA
G5	1	+0.001%	+0.189%	-	N

Table D.7: Case study 3 iteration 1 change groups overview

Group	Change impact	IntraMD Cohesion	InterMD Coupling	Rule penalty	Accepted
G1	1	+0.001%	+0.129%	0	PA
G2	1	0%	+0.02%	-2	Y
G3	2	0%	+0.099%	-5	N
G4	1	+0.001%	+0.109%	-1	PA

Table D.8: Case study 3 iteration 2 change groups overview

Group	Change impact	IntraMD Cohesion	InterMD Coupling	Rule penalty	Accepted
G1	1	-0.03%	0%	-1	Y
G2	1	+0.001%	+0.01%	+1	Y
G3	1	+0.001%	0%	0	Y
G4	1	0%	+0.02%	-1	PA
G5	1	0%	+0.02%	-1	PA
G6	1	0%	+0.128%	-5	N
G7	3	+0.001%	+0.01%	+6	PA
G8	1	+0.005%	+0.088%	+7	N
G9	1	0%	-0.02%	-3	Y

Table D.9: Case study 3 iteration 3 change groups overview