Automated reengineering using evolutionary coupling

Hermans, F.

Award date:
2011

Eindhoven University of Technology
Department of Mathematics and Computer Science
Software Engineering and Technology Group

Master's thesis

Automated Reengineering using Evolutionary Coupling

Frank Hermans

August 2011

Supervisor:
dr. A. Serebrenik

Abstract

As software systems are changed in order to add, remove or modify functionality, the quality of the modular design of the code degrades. In order to improve the code structure again, the source code needs to be reengineered. However, reengineering is labor intensive work, and thus it is desirable to automate this process.

Currently available research on automated reengineering has at least one of three problems. First, a lot of research only focusses on automating a single step of the reengineering process (that is, problem detection, or finding an improved design). Secondly, existing approaches often change the package structure, making it difficult for developers familiar with the original structure to understand the new structure. Finally, the known techniques use metrics which have been shown to have problems in the way they measure coupling and cohesion in another study.

To solve these problems, a new approach to automated reengineering is proposed. Other studies have shown that evolutionary coupling is capable of detecting relations between code entities, based on the change history of the code. Based on evolutionary coupling, we define metrics capable of quantifying the modularization quality of a software system. A case study shows that these metrics are successful in measuring the modularization quality.

Furthermore, an optimization technique is proposed, which uses a genetic algorithm to improve the code structure by optimizing the evolutionary coupling metrics. The genetic algorithm uses multi-objective optimization to simultaneously reduce the number of changes made to the code, ensuring some form of code stability.

Experiments performed for validating the optimization strategy show promising results. The proposed technique is capable of improving the code structure. Furthermore, the results show that multi-objective optimization is preferable over single-objective optimization.

As care needs to be taken when reengineering software, in order to ensure the behavior is not changed, further research into automated reengineering is required. For this purpose, a platform is developed for conducting research into automated reengineering. This platform is also used to execute the experiments of the validation studies performed in this thesis.

Acknowledgements

This thesis is the result of my master project, which concludes my Computer Science and Engineering program at the Eindhoven University of Technology. The project was carried out internally at the Software Engineering and Technology group, department of Mathematics and Computer Science.

I would like to take this opportunity to thank a few people for their support during my project. First of all, I would like to thank my supervisor, Alexander Serebrenik, for his support and guidance during my project, and before the start of the project in defining the project goal.

Furthermore, I would like to thank Toon Calders for his advice on association rule mining, and Serguei Roubtsov for his feedback on the design of the developed tool.

Finally, thanks go out to my friends and family for their support and understanding during my project.

Contents

1. Introduction

In object-oriented programming languages such as JAVA, the source code can be structured into packages. These packages group related classes together, while the classes each implement a set of related methods. A good package structure should realize a high cohesion among the classes in the same package, and a low coupling between two separate packages [60]. Such a good code structure is referred to as a good *modularization* of the code.

A good modularization is important, because it eases a number of issues related to software engineering [12]. For example, when a developer needs to work on code not yet familiar to them, they first have to understand the structure of the code. A good modularization helps the developer in gaining this understanding. Furthermore, maintenance is easier to perform and testing of the code also becomes less problematic.

Unfortunately, as software evolves, and functionality is added, removed or changed, the quality of the modularization degrades [15]. Eventually, due to shifting responsibilities and changing relations between classes, some classes might no longer be placed in an appropriate package [25]. To improve the modularization quality again, reengineering of the code becomes necessary.

The necessity of reengineering also follows from Lehman's laws of software evolution [36]. Lehman formulated eight laws which he stated apply to all evolving software systems. The following two laws are applicable to show the need for software reengineering:

1. Systems must be continually adapted or they become progressively less satisfactory.
2. The complexity of an evolving system increases unless work is done to maintain or reduce it.

The first law states that a software system needs to be modified by adding new features or fixing bugs. If this is not done, the changing environment causes the system to become less satisfactory. For example, if a competing software product does evolve, users might end up preferring the other product, thus decreasing revenue. Alternatively, as computers become more powerful, users expect more from software products. Thus, software systems need to evolve.

According to second law mentioned, an evolving software system becomes increasingly more complex, unless this complexity is actively maintained or reduced. Thus, as new features are added and bugs are fixed, the complexity of the software increases. Therefore, reengineering is important in order to reduce this complexity again.

The main problem with reengineering, is that it is labor intensive. The existing relations between classes and methods need to be identified in order to find problems in the modularization. Next, a new and improved modularization needs to be designed, which ultimately needs to be implemented. This entire process is hard to do manually [8], and therefore automating the reengineering is desirable.

Research into automated reengineering has often focused on only a single step of the process. That is, there are multiple studies on automating the detection of design problems [8, 9, 17, 61]. Furthermore, some research focusses on finding a good, modular design for a software system [40, 44, 45, 48]. Finally, there is research available which studies the actual reengineering step, modifying the source code to improve the code quality [7, 21, 43].

Combining all the steps of the reengineering process into a single tool, capable of automating the complete process, has also been researched in some studies [44, 46]. However, such studies often change the package structure, aiming to find the best modularization for the existing code. With this approach, it can become problematic for programmers who are familiar with the system to understand the new modularization, and map the new situation back to what they know.

Furthermore, existing studies on automated reengineering frequently work with coupling and cohesion metrics based on static analysis of the source code [1, 46]. That is, classes are considered to be related when one class uses the other. However, problems with these metrics have been identified by Anquetil and Laval [4].

There is, however, another technique which can be used to determine which classes are related. *Evolutionary coupling* [61] is based on the change history of source code. Information from the change history is used to determine which classes were often changed together in the past, in which case they are said to *co-evolve*. Co-evolving classes are then considered to be related.

Evolutionary coupling has shown to be successful in detecting relations between classes and other source code entities in a number of studies [61, 63]. Other studies have shown that evolutionary coupling can detect relations between entities not detected using source code analysis [32, 33]. That is, entities might co-evolve, even if no dependencies between the entities is detected from source code analysis. One study even showed that evolutionary coupling performs better than standard coupling measures, such as those based on call or use relationships [28].

In this thesis, we propose a technique for automated reengineering using evolutionary coupling. For this purpose, a platform is developed for conducting empirical research into automated reengineering. This platform is then used to investigate whether evolutionary coupling is indeed suitable for use in automated reengineering. See Chapter 2 for a description of this platform.

Furthermore, metrics are defined, based on evolutionary coupling, in Chapter 3. These metrics are capable of measuring the modularization quality of software. While such metrics are not readily available in the literature, we do use and combine existing and proven techniques to define our metrics. A tool is developed capable of validating these metrics. This tool is discussed in Chapter 4. The metrics are then validated in Chapter 5, before they are used for automated reengineering.

Software modularization is a graph clustering problem [49]. As the graph clustering problem is NP-complete, finding a good modularization using a deterministic algorithm is infeasible [22]. Therefore, we use a genetic algorithm to solve the problem [29], which is presented in Chapter 6. A discussion of the tool developed for automated reengineering is given in Chapter 7. Validation of the proposed reengineering technique is then done by performing experiments, of which the results are presented in Chapter 8.

Note that this thesis is limited to reengineering in the form of improving the modularization. That is, other aspects of code quality, such as code duplication [34], are not considered.

Furthermore, modularization in general is not limited to moving classes into packages. Instead, we can also imagine moving methods into classes, or even (sub)packages into packages. Therefore, we use the term *module* to refer to those elements of the source code which can contain other elements (like packages can contain classes, and classes can contain methods). Note that modules can also contain other modules. Furthermore, we use the term *entity* to refer to code elements which cannot contain other elements, and are thus considered the smallest type of element.

While we do use the general terms to indicate the presented techniques are also applicable for other levels of granularity, we limit ourselves to moving classes into packages. The reason for this is that evolutionary coupling is not suited for fine-grained analysis when the system under analysis is unstable [63].

2. Platform overview

This project is split into two parts. The first part is dedicated to defining metrics based on evolutionary coupling. This also includes a validation study of the metrics to determine which metrics are best suited for measuring the quality of the source code structure. The second part then uses the most promising metrics to investigate if they can be used for automatically improving the source code.

For both parts, a software tool is created which is capable of performing the necessary analyses. That is, one tool is created which can validate the metrics on a case study, and another tool is created for improving the quality of the structure of source code. In this chapter, an overview of both these tools is given, which together form a platform for research into automated reengineering.

The scope of the platform is given in Section 2.1. Next, an overview of the design of the complete platform is given in Section 2.2. Note that the information given in this chapter only provides an overview. The details of the two tools, including specific requirements and an architectural design, are discussed in Chapter 4 (metric calculation) and Chapter 7 (optimization tool).

2.1 Scope

The platform consists of two tools: the *metric validation* tool and the *restructuring* tool. The scope of the platform is depicted in Figure 2.1. Both of the tools take their input from the software repository of the software project under investigation.

The software repository is a version control system such as Subversion (SVN), containing historical data on the software project. From the software repository, the two necessary types of information can be retrieved.

The first is the commit history, which is retrieved as a collection of commits. Each commit contains a collection of entities which were modified in the respective commit. This information is sufficient to determine the evolutionary coupling between entities, and thus to calculate the

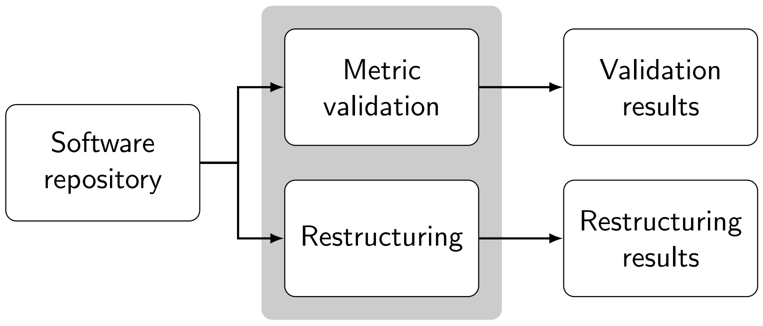

Figure 2.1: *The scope of the platform. The shaded area indicates the platform, containing the two tools. Both tools take their input from software repositories.*

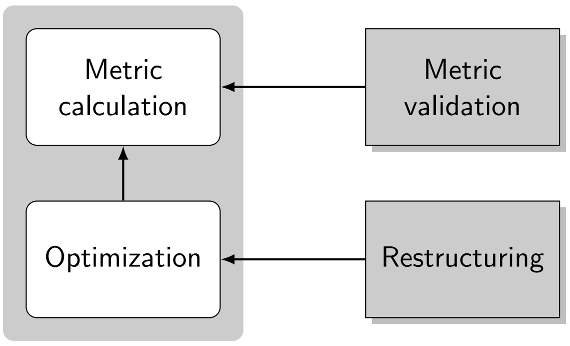

Figure 2.2: *An overview of the design of the two tools: the metric validation tool and the restructuring tool. The metrics calculation package contains all code for calculating the evolutionary coupling metric values. The optimization package contains additional code necessary for the structure optimization process.*

evolutionary coupling metrics defined in Chapter 3.

The second type of information retrieved from the repository, is the current source code structure. That is, a tree structure representing the source code of the software project is constructed from information retrieved from the repository. The tree structure corresponds to the modular structure of the code.

The metric validation tool is used to validate metrics. It does this by retrieving information from a software repository for a software project that has undergone a successful restructuring effort in the past. Metric values for a version of the code before and a version after the restructuring are then calculated, and the results can be used for a validation study. More details on the metric validation tool are given in Chapter 4.

The restructuring tool also uses the information from the software repository to calculate metric values. However, this tool uses an optimization algorithm to find a code structure for which the metric value is optimized. For the purpose of validating that the output is indeed better structured, validation results are output. See Chapter 7 for more details on the optimization tool.

2.2 Design overview

As the two tools contain overlapping functionality, it is important that the tools are not developed independently from each other. That is, both tools need to extract information from a software repository and use that information to calculate metric values for the source code. Therefore, we implement the functionality in a library, which both tools use. The tools then contain only the code necessary to execute their task, using the library.

An overview of the design of the platform is shown in Figure 2.2. Here, we see that the platform contains the two tools (*Metric validation* and *Restructuring*) and a library. The library can be split into two parts. One part, *Metric calculation*, contains all the code needed to calculate metric values, given a software repository. The other part, *Optimization*, contains the additional code needed for the optimization process.

Note that the metric validation tool only depends on the first part of the library. That it, this tool does not depend on the *Optimization* part, as no optimization is performed by this tool. The restructuring tool, however, only directly depends on *Optimization*, while this part in turn depends on *Metric calculation*.

3. Evolutionary coupling

The history of a software product as provided by a version control system such as Subversion can be used to get information on the co-evolution of entities in the code. In general, we can expect that two entities which co-evolved in the past to continue to co-evolve in the future. This expectation is supported by studies in the field of software change prediction [32]. Co-evolving entities are thus in a sense related to each other, and it can be said that they are coupled. This coupling, which is based on the evolution of the code, is called *evolutionary coupling*.

In order to use evolutionary coupling for measuring the quality of the code, we need a metric based on this historical data. In the past, evolutionary coupling has mostly been used in the form of association rules, stating that a change in a certain entity often corresponds with a change in another entity [32, 61]. This method can be used to identify individual problems in the architecture, such as strongly coupled methods which are in different modules. However, such a technique does not allow for quantifying the quality of the complete code base, which is what is needed as a measure to optimize the quality. It is thus necessary to define a metric which provides us with a number that corresponds to the code quality. Zimmermann et al. [61] did define such metrics, the EDI and ECI. However, such existing metrics are not satisfactory for reasons explained in Section 3.4.

A more extensive motivation for the use of historical data for determining coupling is given in Section 3.1. This motivation includes a discussion on problems with other techniques for measuring the coupling and why evolutionary coupling can be expected to give better results. Then, before we can define a metric, some requirements have to be formulated which the metric should adhere to. These requirements are presented in Section 3.2. Next, several definitions for a metric are given and discussed in Section 3.3. In Section 3.4, the related work is discussed. This includes a discussion of existing techniques and why these cannot be used to achieve our goals. Finally, Section 3.5 discusses additional ideas for defining a metric and other future work. A validation of the metrics is postponed to Chapter 5.

3.1 Motivation for evolutionary coupling

It has long been accepted in software development that code with a low coupling between different modules is an indicator for good quality code, while high coupling usually implies that the code is difficult to maintain [60]. When entities are tightly coupled, it means that the two entities cannot be independently changed. That causes problems when maintenance is performed in the code, because making a change in one location can cause code in another location to break. This requires one to also change that code, which can easily be forgotten if the coupling between the two parts is not obvious. Therefore, coupled entities should be placed close together, such as in the same module, minimizing the coupling between modules.

It should be noted that not all relations between entities are bad. Often, it is necessary for entities to be related in order to fulfill the requirements of the system. In fact, when entities are grouped together in the same module, it is even desirable for the entities to be somehow related to each other. Unrelated entities which appear in the same module can distract developers, drawing their attention to an unrelated part of the source code. Therefore, it is desirable for relations between entities, that these entities are both in the same module. To measure these desirable relations, we use the cohesion of a module. A well-structured software system has both high cohesion (of the modules) and low coupling (between the modules), or more importantly, a cohesion which is higher than the coupling.

Anquetil and Laval [4] studied several successful restructuring efforts, measuring several metrics based on coupling and cohesion. The authors studied the evolution of the Eclipse platform between successive versions. In two of the cases (the evolution from version 2.0.1 and 2.1, and from 2.1 to 3.0), the code was purposely restructured in order to solve modularization issues. That is, the developers recognized problems with the architecture of the code and attempted to resolve those problems during restructuring. As a third case, the evolution from version 3.0 to 3.1 is studied in order to investigate whether issues detected after restructuring were fixed (this indicates maturation of the structure). The authors argue that due to the continuing success of the Eclipse platform, the restructurings can be considered effective, thus improving the modularization.

Furthermore, the effect of the restructuring on four metrics, which measure the coupling and cohesion, is determined. This is done for the Bunch cohesion/coupling metrics [40] and for afferent and efferent coupling [41]. For the Bunch metrics it is found that the coupling metric shows a globally slightly decreasing trend, which is to be expected with an improved structure, but the effect was not significant enough to state a clear improvement. Furthermore, it is expected that with improved structure the cohesion increases, but a clearly opposite effect was detected. For afferent and efferent coupling, it is found that both metrics show an increasing trend, while for a better structure, it is expected for the coupling to be reduced.

As the results are contradictory to what was expected for successful restructurings, Anquetil and Laval [4] conclude that the way coupling and cohesion are measured by the studied metrics is flawed. The authors still argue that developers should aim for low coupling and high cohesion, but add that no good metrics exist capable of measuring the kind of coupling and cohesion which developers try to optimize. In the words of the authors: *"One must consider that software engineers work with higher level concepts that cannot be measured by the simple, existing, cohesion/coupling metrics"*. Thus, in order to obtain good metrics, we must find a way to measure such higher level concepts.

The problem with existing coupling/cohesion metrics is that they are based on the assumption that a syntactical dependency between two entities (that is, one entity uses the other) is the same as a coupling between the two entities. However, when some form of API stability is guaranteed, it is possible that an entity depends on another, while the two are still only loosely coupled. That is, it is possible that the two entities can be independently changed, as long as the black-box functionality of the entities remains the same.

Inversely, it is possible that two entities which are not considered related by techniques using static analysis, are still coupled. An example of this is code for reading and writing settings from and to a configuration file. In such a case, it is important that both parts (reading and writing) use the same format for the settings, as stored in the file. Thus, if the format is changed in which the settings are written, the code reading these settings must also be changed in order to correspond with the changed format. While it is possible to detect that the two parts use the same file and can thus be considered to be coupled, this is more complicated than detecting syntactical coupling and is therefore not done in practice.

In order to provide a better technique for measuring the coupling of a software product, we propose to look at the development history of the source code. We can expect that two entities which have co-evolved in the past are coupled and thus expected to continue to co-evolve in the future. This expectation comes from work done in the field of software change prediction, where evolutionary coupling has been successfully used in order to predict which entities should be changed, given an initial change [61, 63]. That is, studies in this field have shown that co-evolution in the past is indeed a good predictor for co-evolution in the future. Therefore, evolutionary coupling actually can be expected to perform better at identifying dependencies in the code which are truly there in practice. That is, it should be successful in detecting relations between entities, where a change in one entity frequently requires a change in the other entity.

We can thus say that evolutionary coupling is based on historical evidence, rather than on a possible dependency. This means that it is expected that evolutionary coupling will perform better in measuring the code quality, and indeed that improved code quality will be reflected by decreased evolutionary coupling.

3.2 Requirements for the metric

Before we can formulate a definition for a metric based on evolutionary coupling, we must first consider what requirements there are on the metric. In general, we want the metric to measure the quality of the code structure. That is, we want to quantify how well the entities are distributed over multiple modules. Therefore, we want the metric to increase as the code structure improves (or, it is also acceptable if the metric is inversely correlated with code structure, thus if it decreases as the structure improves). However, there are some other properties which the metric should have. These requirements for the metric are given in the remainder of this section.

1. **The metric should be normalized**
   To make it possible and meaningful to compare the quality of two different source code trees (two different versions of the same software product), the metric values should be comparable. If the possible range of values is in some way dependent on the source code (for example, if it is at most equal to the total number of modules), it does not make sense to compare the values for different code. Additionally, to make it easier to interpret the metric value as a quality measure, it is desirable for the metric to have a well-defined constant range. This also means the metric values can be compared in a meaningful way, and it is possible to quickly get an idea on how good (or bad) the code quality is.

2. **The metric should not be biased towards a specific module organization**
   It is also important that the metric should not favor a lot of smaller modules over a single large module, or other differences in module organization. For example, although coupling between two entities which are in two different modules is considered bad, the metric should not give an optimal value when all entities are in the same module (and there are thus no inter-module dependencies). In most cases, it is better to have some inter-module dependencies as having all entities in the same module.

3. **The metric should not be dependent on the length of the history**
   Obviously, using a longer history can uncover more (or other) relations than using a shorter history. Also, it is of course important to choose an appropriate history length. When the history is too short, not enough information will be available, while when it is too long, there will be too much noise (old changes which have no impact on the current version of the code). However, the metric should somehow be robust for varying history lengths, which means that when we look back into the past twice as far, we should detect approximately the same number of relations (as opposed to, for example, twice as many).

4. **The metric should work on multiple granularity levels**
   In order to make it possible to apply the presented techniques on multiple granularity levels, it is important that the metric does not depend on a specific granularity level. That is, while we currently consider entities to be files and modules to be directories, the metric definition should not depend on this. It should, for example, be possible to take methods as entities and classes as modules and still use the same metric definition without modification.

5. **The metric should take hierarchical module structures into account**
   In Java, packages are given a name consisting of possibly multiple identifiers separated by dots. Although not necessary, programmers usually use packages in a hierarchical manner. That is, a package named `pkgA.pkgB` is usually considered to reside in package `pkgA`. The metric should take such hierarchical structures into account and consider `pkgA.pkgB` to be a program element in module `pkgA`.

3.3 Quantifying evolutionary coupling by defining metrics

Now the requirements for the metric are known, we can give a definition. Multiple definitions are possible, which all have their advantages and disadvantages. As it is not a priori known whether a given definition will work well in practice, we define several alternative metrics. The reasoning behind these definitions is discussed in this section. In order to provide stronger evidence for the applicability of the different definitions, a validation study is presented in the next chapter. This validation compares how the definitions perform in practice and allows us to discuss whether the metrics are appropriate for quantifying the code quality.

In order to satisfy Requirements 1 and 2, we look at an existing metric which combines coupling and cohesion into a single value. This metric has in the past been applied for standard coupling (that is, non-evolutionary coupling) [13, 40]. The idea is that the coupling between entities in different modules should be small, while the cohesion among the entities in a single module should be large. This technique is discussed first in Section 3.3.1. In the remaining sections some problems with the basic definition of this technique are discussed.

3.3.1 Modularization quality

Doval et al. [13] used a measure called $Modularization\ Quality\ (MQ)$ in order to combine coupling and cohesion in a single metric, measuring the quality of a clustering of modules in subsystems. The MQ is also used in this thesis, adapted to suit our specific needs. MQ consists of two parts: a measure for the cohesion of modules and a measure for the coupling between modules. The MQ then measures the quality of a modularization, such as the modularization shown in Figure 3.1.

The first part of MQ is *inter-connectivity*, which measures the connectivity between two modules, or the coupling between two modules, and a low value means low coupling, which thus indicates good modularization.

Definition 1 (Inter-connectivity). For modules i and j ($i \neq j$), let N_i and N_j denote the number of elements in the modules i and j respectively. Furthermore, let $\varepsilon_{i,j}$ be the number of relations between an element from module i and an element from module j. The inter-connectivity $E_{i,j}$ between the modules i and j is then defined as

$$E_{i,j} = \frac{\varepsilon_{i,j}}{N_i N_j}$$

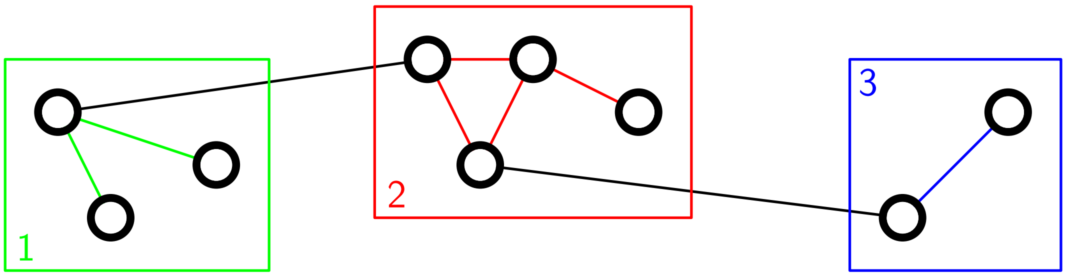

Figure 3.1: *An example modularization of 9 entities, divided over 3 modules. An edge between two nodes denotes a relation between two entities.*

Thus, $E_{i,j}$ is equal to the number of inter-relations ($\varepsilon_{i,j}$), divided by the maximum number of possible inter-relations (N_iN_j). The value of $E_{i,j}$ is equal to 0 when the two modules are completely unrelated, and equal to 1 when all elements from module i are related to all elements from module j.

As an example, consider Figure 3.1. Here, the inter-connectivity between modules 1 and 2 is equal to

$$E_{1,2} = \frac{\varepsilon_{1,2}}{N_1N_2} = \frac{1}{3 \cdot 4} = \frac{1}{12}$$

Similarly, we have $E_{1,3} = 0$ and $E_{2,3} = 1/8$.

The main goal is to minimize the inter-connectivity. However, using only the inter-connectivity is not enough, as then the metric would be optimized whenever all entities are in the same module. To solve this problem, we define *intra-connectivity*, which is a measure of the density of relations between elements in the same module, thus the cohesion of a module. A high value corresponds to high cohesion and therefore is an indicator for a good modularization.

Definition 2 (Intra-connectivity)**.** For a module i, let N_i denote the number of elements in this module and μ_i the number of relations between two elements both in module i. Then, the intra-connectivity A_i of module i is defined as

$$A_i = \frac{2\mu_i}{N_i(N_i - 1)}$$

A_i is thus the number of intra-relations (μ_i), divided by the maximum number of possible intra-relations ($N_i(N_i - 1)/2$). Note that this definition assumes that an entity cannot be related to itself. This is because entities are considered to be related whenever they co-evolve, and it is not possible for an entity to co-evolve with itself (or alternatively, all entities could be considered to co-evolve with itself, in which case the inverse is not possible). The value of A_i ranges from 0, when no relations exist inside the module, to 1, when all elements in the module are connected to each other.

Consider Figure 3.1 for an example. The intra-connectivity of module 1 is equal to

$$A_1 = \frac{2\mu_1}{N_1(N_1 - 1)} = \frac{2 \cdot 2}{3 \cdot 2} = \frac{2}{3}$$

For modules 2 and 3, we get $A_2 = 2/3$ and $A_3 = 1$.

Besides minimizing the inter-connectivity, we now also want to maximize the intra-connectivity. Note, however, that in fact a large number of intra-relations is not necessarily a good thing. In general, it is actually a good thing to have as few relations as possible, including intra-relations. For our purpose, though, we have a given set of relations, and it is better to have these relations contribute to the intra-connectivity than to the inter-connectivity. That is, adding more intra-relations does not improve the quality. However, when a constant set of relations is given and entities are moved, code quality does increase if more relations become intra-relations.

MQ can now be defined by combining the intra-connectivity and the inter-connectivity in a single value. As the intra-connectivity should be high while the inter-connectivity is low, we subtract the latter from the former. For both parts, we use the average over all modules.

Definition 3 (Modularization quality)**.** Let k be the total number of modules. The modularization quality MQ is then defined as

$$MQ = \frac{1}{k}\sum_{i=1}^{k} A_i - \frac{2}{k(k-1)}\sum_{i=1}^{k-1}\sum_{j=i+1}^{k} E_{i,j}$$

MQ thus rewards a high intra-connectivity and punishes a high inter-connectivity. The value is bound to the range $[-1, 1]$, where -1 means no cohesion and maximum coupling, and 1 corresponds to maximum cohesion and no coupling. This fixed range also ensures that Requirement 1 is satisfied.

For an example, consider again Figure 3.1. We know the inter-connectivity of all pairs of modules and the intra-connectivity of each module. Furthermore, with $k = 3$, we get the following value for the MQ:

$$\begin{aligned} MQ &= \frac{1}{3}\sum_{i=1}^{3} A_i - \frac{2}{3(3-1)}\sum_{i=1}^{3-1}\sum_{j=i+1}^{3} E_{i,j} \\ &= \frac{1}{3}\left(\frac{2}{3} + \frac{2}{3} + 1\right) - \frac{1}{3}\left(\frac{1}{12} + 0 + \frac{1}{8}\right) \\ &= \frac{7}{9} - \frac{5}{72} \\ &= \frac{51}{72} \approx 0.71 \end{aligned}$$

As this value is larger than 0, the intra-connectivity is larger than the inter-connectivity, which is a good sign. Furthermore, the value is actually quite near to 1, indicating that the modularization is good.

Note that the definition of MQ used differs from the definition as used by Doval et al. [13]. The original MQ uses directional dependencies, effectively doubling the maximum number of possible relations. However, we use undirected relations, and thus the definitions were adapted accordingly.

The definition for MQ as it is now, is still lacking in several ways. First of all, we need to know the values for all μ_i and $\varepsilon_{i,j}$, which means that we need a way to determine which elements are considered to be related to each other. Furthermore, the current definition considers only a set of entities grouped into modules. That is, the hierarchical structure of modules is not taken into account, thus violating Requirement 5. We need a way to support the case where a module can itself contain (sub)modules.

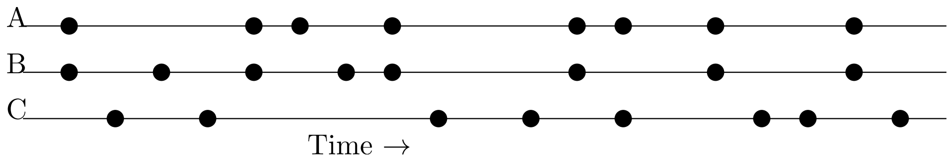

Figure 3.2: *Visualization of the evolution of three entities. A dot represents a change in the entity at the given time. Vertical alignment of two dots means that the two corresponding entities were changed together (in the same commit), and thus co-evolve.*

In Section 3.3.2 we look at techniques for detecting whether two entities are related or not. In Section 3.3.3, this is taken further up in the hierarchy by investigating techniques which can be used to detect relations between modules (or a module and an entity). Finally, we discuss aggregating multiple MQ values into a single metric value in Section 3.3.4.

3.3.2 Detecting relations between entities

In order to count the number of relations, we must first be able to determine whether two entities are related. To do this, we use the history of the source code as stored in a version control system such as Subversion. From this, we can determine which entities co-evolved. Using techniques from association rule mining [2], we determine if evidence of a relation between two entities is strong enough to consider those entities to be related.

The history of source code can be seen as a series of commits, where each commit changes one or more entities. For a version control system which uses file versions as opposed to product versions, such as CVS, preprocessing is needed to group commits together first [62].

A short sample of such a history for three artificial entities is depicted in Figure 3.2. When two entities are changed at the same time (in the same commit), they co-evolve. In general, such co-evolution is an indicator for a relation between the two entities. However, whether the entities should really be considered to be related, depends on the strength of the co-evolution. That is, if they co-evolve almost in every case, evidence for a relation is strong. However, when co-evolution is rare, then it may be considered coincidental and no relation between the entities exists. In Figure 3.2, one would probably consider A and B to be related, because they are often changed together. On the other hand, while there is one occurrence where both A and C are changed, they are more often changed independently. Thus, A and C should not be considered to be related.

There are several measures which can be used in order to quantify the strength of the co-evolution between two entities. The simplest method is to count how often the entities co-evolve and check if this number is above a certain threshold. This value is called the *support*.

Definition 4 (Support)**.** Consider two entities: e_1 and e_2. The support of entity e_1 is denoted by $\mathrm{supp}(e_1)$ and is defined as the number of commits in which e_1 is changed. The support of the entities e_1 and e_2 together, denoted by $\mathrm{supp}(e_1, e_2)$, is defined as the number of commits in which both e_1 and e_2 are changed.

Support is very simple and also very effective at removing potential relations between two entities when those entities only co-evolve infrequently. However, the problem is that when the support is used, an entity which is often changed is much more likely to become related to other entities than one which is almost never changed, even when those co-evolutions are purely coincidental. Therefore, we use the *confidence* of a relation. This confidence value is in principle the same as the confidence of association rules, with the difference that in our case a relation is undirected while an association rule is directed. To make the confidence independent of a direction, the two directed confidences are combined into a single value.

Definition 5 (Confidence)**.** The confidence of a relation between the two entities e_1 and e_2, denoted by $\mathrm{conf}(e_1, e_2)$ is defined as

$$\mathrm{conf}(e_1, e_2) = \mathrm{combine}\left(\frac{\mathrm{supp}(e_1, e_2)}{\mathrm{supp}(e_1)}, \frac{\mathrm{supp}(e_1, e_2)}{\mathrm{supp}(e_2)}\right)$$

This definition is the combination of the confidence of the association rule $e_1 \Rightarrow e_2$ and the rule $e_2 \Rightarrow e_1$, where the confidence of an association rule corresponds to the probability of the right-hand side occurring, given that the left-hand side occurs. Since relations between entities are undirected, the two values are combined using a combination function. Three different combination functions are used: maximum, minimum and average.

The maximum is used in order to consider a relations between two entities to be strong whenever the association rule in one of the directions is strong. This ensures that a relation is detected even if one of the entities is changed much more frequently than the other. When we want both association rules to be strong, we can use the minimum, which ensures that relations which are unidirectional are not considered. This is mostly important in order to prevent a lot of relations from showing up between two layers of a layered architecture. The average is an intermediate form, where a weaker rule can be compensated by a stronger rule.

Now, by giving a minimum support value $\mathrm{minSupp}$ and and minimum confidence value $\mathrm{minConf}$, we can check whether two entities are related. If both the support and the confidence are large enough, the entities are considered to be related. There is however one small problem with giving a minimum support value. If a longer history is used, the support values also become larger. Therefore, we actually use the minimum support parameter as a fraction of the total number of commits, making the relations less dependent on the length of the history, and thus satisfying Requirement 3.

Definition 6 (Confidence-based relation)**.** Given two entities e_1 and e_2, parameters $\mathrm{minSupp}$ and $\mathrm{minConf}$ and the total number of commits n, we consider the entities related if the predicate $R_c(e_1, e_2)$ is true, where $R_c(e_1, e_2)$ is defined as

$$R_c(e_1, e_2) = \mathrm{supp}(e_1, e_2) \geq n \cdot \mathrm{minSupp} \wedge \mathrm{conf}(e_1, e_2) \geq \mathrm{minConf}$$

The confidence-based technique uses the confidence as a way to check the strength of co-evolution. However, it is possible for an entity e_1 to be changed in 95% of the commits, but only in 90% of the commits in which entity e_2 was changed. In this case, the presence of e_2 actually reduces the chance of e_1 being changed in the same commit. But since the confidence of e_1 and e_2 is still very high (0.9), a relation will be detected between the entities, even though they are actually negatively related. To solve this problem, we can use the *lift* [37].

Definition 7 (Lift)**.** Given entities e_1 and e_2, $\mathrm{lift}(e_1, e_2)$ is defined as

$$\mathrm{lift}(e_1, e_2) = \frac{\mathrm{supp}(e_1, e_2)/n}{(\mathrm{supp}(e_1)/n)(\mathrm{supp}(e_2)/n)} = \frac{n \cdot \mathrm{supp}(e_1, e_2)}{\mathrm{supp}(e_1)\mathrm{supp}(e_2)}$$

This value is larger than one whenever the two entities co-evolve more often than what would be expected by chance. On the other hand, when they co-evolve less frequently than expected, the lift is smaller than one.

H_0: $\text{lift}(e_1, e_2) = 1$
H_A: $\text{lift}(e_1, e_2) > 1$

Table 3.1: *Hypothesis test. The null-hypothesis is rejected (and the alternative accepted) when the probability of the observed result, under the assumption that H_0 is true, is smaller than the a priori chosen significance level.*

	e_1	$\neg e_1$	**Totals**
e_2	a	b	n_2
$\neg e_2$	c	d	$n - n_2$
Totals	n_1	$n - n_1$	n

Table 3.2: *Contingency table summarizing the number of commits for each possible combination of presence of the entities e_1 and e_2. Here, n denotes the total number of commits, and $n_i = \text{supp}(e_i)$ for $i = 1, 2$.*

However, there is still a possibility of the lift being larger than one even if the entities are unrelated. In fact, it is unlikely for unrelated entities to have a lift which is exactly equal to one. Therefore, we want to check whether the lift is large enough in order to consider it sufficient evidence that the entities are indeed related. Instead of using a simple — but rather difficult to interpret — minimum lift value, we use a hypothesis test [18] as shown in Table 3.1.

When we look at the definition of lift, we can see that the alternative hypothesis (and in a similar way, the null-hypothesis) can be reformulated as

$$\text{supp}(e_1, e_2)/n > (\text{supp}(e_1)/n)(\text{supp}(e_2)/n)$$

Considering the support of the separate entities as fixed, we are thus interested in whether the support of both entities together is large enough. We can represent this problem as a *contingency table* [56], as shown in Table 3.2. This table summarizes the number of commits according to the presence (or absence) of two entities. In this table, we have $a = \text{supp}(e_1, e_2)$, and we are thus interested in checking whether a is large enough such that the observed values are unlikely in case e_1 and e_2 would be unrelated. Furthermore, we use $n_1 = \text{supp}(e_1)$ and $n_2 = \text{supp}(e_2)$ as abbreviations.

In other words, we reject the null-hypothesis when the probability of a result at least as extreme as the one observed, under the assumption that the null-hypothesis is true, is below a certain threshold (supplied as a parameter). In order to do this, we find the p-value using *Fisher's exact test* [19] and compare this value with the supplied threshold.

Fisher's exact test is capable of finding the exact p-value for contingency tables. It is based on the fact that the listed totals are all constant, while the four numbers inside the table can vary based on correlations and chance. Fisher proved that for the contingency table as shown in Table 3.2, the probability of this specific table occurring by random chance alone (that is, the entities are not related) can be expressed as

$$\frac{n_1!(n - n_1)!n_2!(n - n_2)!}{a!b!c!d!n!}$$

To get the probability of a result at least as extreme as the observed result, Fisher showed that we must sum the probabilities of all tables at least as extreme. That is, since we are looking to verify whether the two entities co-evolved often enough, we need to sum the probabilities of all tables with at least the given number of commits containing both entities. In other words, we need to sum the probabilities of all tables with at least the observed value of a.

Definition 8 (Fisher's exact test)**.** Given an observed contingency table with values as shown in Table 3.2, and the maximum possible value for a equal to $s = \min(\text{supp}(e_1), \text{supp}(e_2))$, the

p-value is equal to

$$p = \sum_{i=0}^{s-a} \frac{n_1!(n-n_1)!n_2!(n-n_2)!}{(a+i)!(b-i)!(c-i)!(d+i)!n!}$$

We can then use Fisher's exact test for the hypothesis test as shown in Table 3.1. To do this, we find the p-value of the test and based on a parameter we decide whether the null-hypothesis is accepted or rejected. If it is rejected, the two entities are considered to be related, as there is good evidence that the entities are indeed related.

Definition 9 (Lift-based relation)**.** Given two entities e_1 and e_2, parameters $\mathrm{minSupp}$ and maxP, the total number of commits n and the p-value of Fisher's exact test p, we consider the entities related if the predicate $R_l(e_1, e_2)$ is true, where $R_l(e_1, e_2)$ is defined as

$$R_l(e_1, e_2) = \mathrm{supp}(e_1, e_2) \geq n \cdot \mathrm{minSupp} \wedge p \leq \mathrm{maxP}$$

Note that the support is still used in this definition. This is done for two reasons. The first is that it can be useful to ignore any relations which do not occur enough, because while it may indeed be true that the relation exists, it is not of sufficient interest to consider the entities coupled. Secondly, it immediately takes care of the problem that for a support of zero, a division by zero occurs in the lift calculation. However, for this purpose, a minimum of support of one is sufficient, so the first reason is most important for the use of a minimum support.

3.3.3 Detecting relations between modules

Using the above described techniques, we can decide whether two entities are related or not. However, while these techniques work on entities, it is also possible for modules to contain other modules. In such cases, we must also be able to determine whether there is a relation between two modules (or a module and an entity). A straightforward way of deciding this, is by considering a module to be related to another element if there is an entity in the module which is related to that element. This method is related to single-link clustering [42], in that a single link between two clusters is sufficient to consider them related.

Definition 10 (Child-based relation)**.** A module M and an element e are related according to a relation R exactly when $R(M, e)$ is true, which is defined as

$$R(M, e) = R(e, M) = \exists_{e' \in M} R(e, e')$$

A likely problem with this approach, however, is that the top-level modules are much more likely to be related to each other than bottom-level entities. Top-level modules contain a large number of entities in their hierarchy, which results in a situation where some entity from those modules is changed in a large number of commits. This means that the support of the relations of course also becomes larger, but the confidence and lift are influenced by other factors as well. It is thus expected that this technique is not very useful.

Another technique is to consider a module changed in a commit whenever an entity in that module was changed in the commit. This way, the modules can then be treated in the same way as entities. This is the technique used for generalized association rules [54], where our module structure is the taxonomy. Besides just considering it changed in a commit, we can even look at how often a module was changed in a commit. That is, we can count a commit two times when two entities from that module are changed in the commit.

Definition 11 (Set-based relation)**.** If $C(e)$ denotes the set of commits in which element e was changed, then the set of commits in which module M was changed is defined as

$$C(M) = \bigcup_{e \in M} C(e)$$

A module M and an element e are then related according to a relation R exactly when $R(M, e)$ is true, which is now defined for modules in the same way as for entities.

Definition 12 (Multiset-based relation)**.** If $MC(e)$ denotes the multiset of commits in which element e was changed, then the multiset of commits in which module M was changed is defined as

$$MC(M) = \biguplus_{e \in M} MC(e)$$

A module M and an element e are then related according to a relation R exactly when $R(M, e)$ is true, which is now defined for modules in the same way as for entities.

These two techniques allow for two modules to be related even if no entities in the modules are related to each other. Inversely, it is possible for two modules to be unrelated even if entities in the modules are related. These techniques thus are independent from any relations in lower levels of the hierarchy, which means that relations in the higher levels of the hierarchy are more meaningful in the sense that it is more likely for modules to be unrelated as is the case when using child-based relations.

The multiset-based technique does have one practical problem. Because it is possible for a module to be "modified" multiple times in the same commit, it is also possible for the support of a module to be larger than the total number of transactions. Since this then also occurs for the support of the relations, the confidence is still sensible. However, there is no natural interpretation anymore, potentially weakening the statistics. Therefore, it is expected that this module-level technique is problematic.

Furthermore, because the lift-based technique uses contingency tables, it does not make sense to combine it with the multiset-based relations, because the number of commits which do not contain the module is then not properly defined. While it is possible to use a different definition for support in the case of multisets, this is awkward and it is uncertain how this definition should best be modified in this case. Therefore, the multiset-based technique is only combined with the confidence-based relations, while the other techniques discussed in this section can be combined with both confidence and lift-based relations.

3.3.4 Aggregating over all modules

Two techniques for detecting relations between entities and three techniques for detecting relations between modules have been defined. These definitions make it possible to calculate MQ for all modules which contain modules. That is, the relations can be determined between all program elements, including modules. This makes it possible to calculate both intra-connectivity of modules, and inter-connectivity between modules. Thus, for a group of modules, we can calculate MQ.

However, while it is the case that we can calculate MQ for the system, which is just a group of modules, this is not enough. Each of these modules can itself contain modules, which means MQ is also defined for those modules. We thus need one final step in order to combine all the

MQ values into a single value. For this purpose, since the MQ itself is already based on averages, we simply take the average value of all calculated MQ values.

Definition 13 (Average MQ)**.** Let S denote the set of all modules for which MQ is defined, and let $MQ(M)$ denote the MQ value for module M. The MQ of the complete system is then defined as

$$MQ = \sum_{M \in S} \frac{MQ(M)}{|S|}$$

Note that other techniques for combining the MQ values into a single value are possible, such as the median or minimum/maximum, or inequality measures [23, 57]. However, the average is the simplest which is useful for our purpose. Minimum and maximum are too local (that is, only one module matters), and techniques for measuring the inequality of the values are more complicated and are not immediately correlated with code quality (which is what we want to measure).

3.3.5 Summary

The metrics defined consist of four basic parts. The first is the MQ, which is the actual function that is used to calculate metric values. The second is one of the entity-level relations detection techniques discussed (confidence-based or lift-based). The third is one of the module-level techniques discussed (child-based, set-based or multiset-based). Finally, the fourth part is the aggregation technique (only the average MQ). This section discusses how these parts together satisfy the requirements identified in Section 3.2.

1. **The metric should be normalized**
   The MQ function always results in a value in the range $[-1, 1]$, where a higher value indicates a better modularization quality.

2. **The metric should not be biased towards a specific module organization**
   The MQ function defines a trade-off between inter-connectivity and intra-connectivity. While inter-connectivity is biased to having all entities in a single module, intra-connectivity is biased to splitting all entities, with only a single entity per module. The MQ, combining both these concepts, is then optimized when a good trade-off is achieved, indicating a good overall modular quality. MQ has been successfully used for the purpose of measuring the modular quality of code without any problems indicating a bias [13].

3. **The metric should not be dependent on the length of the history**
   The history is used for the entity-level techniques (confidence-based and set-based). Both techniques require a minimum support parameter, which indicates the fraction of the commits that should support a relation, as opposed to the actual number of commits. This way, a longer history does not increase this measure (in fact, the fraction can even decrease). Otherwise, a longer history would necessarily increase the support of a relation, thus making the metric detect more relations. However, using the fraction of commits for minimum support ensures the metrics are not dependent on the length of the history.

4. **The metric should work on multiple granularity levels**
   The metrics are defined without making any assumptions on the granularity level. That is, for another granularity level, such as method-level, we only need to redefine what we mean with the words *entity* and *module*. The metric definitions given in this chapter are then applicable on another granularity level.

5. **The metric should take hierarchical module structures into account**
   The MQ is calculated for all modules for which it is defined, and we then aggregate the results by taking the average. Furthermore, we also defined how relations between modules (or a module and an entity) can be detected. These concepts together are based on the hierarchical module structure of the code.

3.4 Related work

Evolutionary coupling is not new, as it has been used in multiple studies in the past. The difference with this study, however, is for what purpose it is used. Most studies calculated association rules from the source code history, and used those rules directly for different purposes. These rules have been used for software change prediction, and for finding architectural problems in the code.

The studies which did define metrics based on these association rules, did so for quantifying the code quality of a single version of the code. That is, the defined metrics are not useful for optimization in order to improve the code structure. The reasons for this are given in this section, along with an overview of other related literature.

3.4.1 Change impact analysis

Most of the research into evolutionary coupling is in the field of change impact analysis or software change prediction [28, 32, 63]. When a change needs to be made to source code (for example to fix a bug or introduce a new feature), this often means that multiple entities need to be changed due to relations between the entities. For this purpose, it is interesting to investigate if it is possible to predict the impact such a change has. For this, evolutionary coupling has been used by returning a list of entities which often co-evolved with a supplied entity containing the initial change.

Kagdi et al. [32] propose a combination of conceptual and evolutionary coupling in order to improve the accuracy for change impact analysis. Conceptual coupling is determined from a single version of the source code, while evolutionary coupling is computed from the history leading up to that version by using item set mining. The lists of entities found using these two techniques are combined into a new list using two techniques: conjunctive (selecting those entities which are in both lists) and disjunctive (selecting those entities which are in at least one list). Both techniques are validated by comparing the results obtained with a test set with the results of the two distinct techniques. It is observed that conceptual coupling and evolutionary coupling indeed find different sets of entities related to an initial entity. Furthermore, the disjunctive combination provides the best performance, which is determined to be statistically significant.

Zimmermann et al. [63] also use evolutionary coupling in order to predict further change locations, given an initial change. A plug-in for ECLIPSE was developed, which would give a programmer real-time advice on which entities should also be considered to change. This provides information in the form "programmers who changed this function, also changed ..." in order to help programmers. Association rules with high confidence are suggested first. If a user tries to commit his changes when there are still rules with high confidence that suggest further changes, the user is warned. The evaluation of the tool shows that it can suggest further change locations in its topmost three suggestions with a likelihood of 64%. Furthermore, it proves to be useful for detecting missing change locations and warning the user about this.

Hassan and Holt [28] compare four different techniques of predicting entities which should be changed. The results show that evolutionary coupling performs better than syntactical coupling (where entities are related based on a *Call*, *Use* or *Define* relation). Syntactical coupling has both smaller recall (that is, misses more correct suggestions) and smaller precision (that is, makes more incorrect suggestions) as evolutionary coupling. A third technique, based on the assumption that a developer only works on a single subsystem and thus suggesting entities the current developer has changed before, also does not perform very well. Finally, the fourth technique suggests all entities defined in the file that was changed. This technique shows the best performance, but of course when restructuring is necessary, the performance of this technique degrades.

This shows that evolutionary coupling indeed proves to be useful in detecting relations between entities in the source code. By using the change history of the software, other relations can be detected which are not found when conceptual coupling is used. Additionally, the relations found by evolutionary coupling perform better in predicting software change locations based on an initial change [28].

In these studies, only association rules consisting of a pair of entities (one antecedent and one consequent) are considered. When a single change location is given, the impact of that change is determined. For evolutionary coupling, this means that all rules with the given entity as the antecedent are selected and sorted on the confidence and support. The consequents then form an ordered set of entities which are expected to be impacted by the initial change. The above studies do not define any numerical metric for quantifying the code quality, by using evolutionary coupling.

3.4.2 Evolution radar

For reverse engineering of source code, it is also important to get an insight in the coupling between entities. D'Ambros and Lanza [9] proposed an approach where evolutionary coupling is calculated and then visualized using an Evolution Radar. This approach makes it possible to look at both the architecture level (modules) to get a global view of the system, and at finer-grained entities (files) to learn which files are responsible for the coupling. The user can interact with the visualization by moving through time (that is, change the time window from which the investigated commits are taken), tracking specific files over multiple time windows, and spawning a new radar based on a set of files to determine which files are responsible for the coupling. The approach is validated on ArgoUML, and several design problems could be identified this way, showing the success of the technique.

While the technique has shown that using evolutionary coupling is useful for detecting design problems, it is only a tool to assist in the process. The detection of design problems is not automated, only a visualization is provided which can be used to find such problems. A software engineer is still required to investigate the results, interpret the visualizations and find problem locations in the source code which should be considered for reengineering. Once these locations are identified, the reengineering has to be performed manually, as well.

The techniques used in the study are again not useable for automatic optimization of the code quality. No metric is defined which can be used for quantifying the code quality. Evolutionary coupling is only used for creating a visualization, which can be used for finding individual problems. Note, however, that some metric is defined in order to determine the distance between two entities in order to draw the visualization. For this, the support is used, but this only measures the distance between two entities and cannot measure the quality of the complete system. The visualization, however, does have an advantage over a metric in some cases. A metric gives no insight on where the problems are, while the Evolution Radar makes this much more easy.

3.4.3 ROSE

Zimmermann et al. [61] used evolutionary coupling extracted from CVS archives in their tool called ROSE. Besides computing association rules and their respective support and confidence values, two numerical metrics were introduced, EDI and ECI, which can quantify the evolutionary coupling for a system. The association rules are considered to be relations between entities when the support and confidence of the rule exceed a certain threshold. The EDI is defined as the number of actual relations, divided by the number of possible relations. The ECI is defined as the ratio between inter-relations and intra-relations. High values of these indices point to design problems, while smaller values are better (with an optimum at 0 for both indices). The evolutionary coupling information is then used to analyze several open source systems, including the GNU Compiler Collection (GCC) and the DDD Debugger. The data is compared with the actual architecture of the software. GCC turns out to be well-structured with relatively few dependencies between entities. On the other hand, DDD is not so well-structured, with relatively more dependencies between entities.

While both metrics make it possible to measure the code quality, neither is appropriate as an objective function for optimization of the code quality. The EDI is only dependent on the number of relations and the number of entities. When entities are moved to other modules, the structure of the code changes, but both the number of relations and the number of entities remain constant. Thus, the EDI also remains constant as the entities are moved. The ECI, however, does change as entities are moved, as it relates inter-relations to intra-relations. But the problem with this metric is the same as when we would only use inter-connectivity: the metric is optimized when all entities are in the same module. In that case, no inter-relations exist and the metric value becomes 0.

The ROSE tool was extended and implemented as an ECLIPSE plug-in by Zimmermann et al. [63]. This tool provides a programmer with information on the evolutionary coupling in the form of "Programmers who changed this function, also changed ...". It is concluded that for stable systems (such as GCC), ROSE works very well with fine-grained analysis (on method level, that is, when entities are methods). But for more rapidly evolving systems (such as KOFFICE), it is more useful to use a file level granularity for suggesting further changes.

ROSE turns out to be a useful tool for suggesting further change locations given an initial change. When in the past changing a method in one class always coincided with changing a method in another class, or maybe even in a completely different package, ROSE will suggest this additional location to the programmer. This can prevent errors resulting from incomplete modifications, because a dependency was missed by the programmer.

By using evolutionary coupling to guide the reengineering process, we can attempt to solve the actual underlying problem of a bad design. In a good design, two methods which are strongly related should be placed in the same class, and ROSE would only suggest change locations in the same class as the initial change. With such a design, a programmer is much less likely to miss a dependency and a tool such as ROSE, although still useful, is not as important.

3.5 Future work

In this chapter, a number of techniques for defining metrics to measure the code quality have been discussed. It should be noted, though, that more techniques for achieving the same goal exist. Additionally, while the presented techniques are evaluated and compared in Chapter 5, other techniques are not evaluated. Therefore, future research can focus on investigating if other techniques are also useful for measuring code quality. This section discusses some of these possibilities.

Alternatives for MQ In this thesis, all defined metrics are based on the Modularization Quality (MQ). This is done because using only coupling has the problem of being optimized when all entities are in the same module, which is not a good code structure. To solve this problem, a combination of inter-connectivity (coupling) and intra-connectivity (cohesion) is used.

The problem with this, however, is that it is actually desirable that even the number of relations between entities in the same module is minimized. Of course, it is better to have many relations within a module and only a few between modules, but the code would be structured even better if there are also few relations within a module.

In order to solve the problem that coupling between the modules alone is not enough, some other metric can be considered. For example, it is possible to use the average number of entities per module, which should be minimized. This metric achieves its maximal value when all entities are in the same module, and is optimal when all entities are in their own module. Therefore, it is also capable of correcting the problem of using only the inter-connectivity.

When this metric is used, though, it should be noted that the range of this metric is not compatible with the range of inter-connectivity. The former ranges over the integers from 1 to N, where N is the number of entities, while the latter ranges over the real numbers from 0 to 1. Therefore, simply using a subtraction is not an option, because then the average number of entities per package dominates the other metric. It might be possible to solve this by dividing the average number of entities by N, but research is required to investigate if this is indeed a good technique for calculating a metric.

Directed relations Relations are considered to be undirected in the metrics defined in this thesis. This makes sense, since we consider that two related entities should be in the same module. It is not possible for two entities to be in the same module in one direction, but in different modules in the other direction. However, it is possible to consider two entities to have a stronger relation between them when the corresponding association rules are strong enough in both directions.

In other words, instead of combining the confidence values of the two association rules into a single value, it is possible to consider the association rules separately. If the rule $e_1 \Rightarrow e_2$ has a sufficiently large confidence (and support), a relation from e_1 to e_2 exists. Similarly, if the rule in the other direction has a large enough confidence, a relation in the corresponding direction exists.

This can also be seen a bit differently. Instead of considering the relations to have actual directions, we can also assign a weight to the relations. For a pair of entities, if neither of the two association rules is strong enough, there is no relation between the entities. If both association rules are strong enough, there does exist a relation between the entities. And if only one of the rules is strong enough, while the other is not, the relation is present with a weight of 0.5, thus as half a relation. This is effectively the same as using directed relations.

Weighted relations We can even take this weighting scheme a bit further. Currently, a relation is considered to be present or absent, based on whether or not the evidence for the relation is strong enough. However, we can also use the strength of the evidence as the weight for the relation. That is, if the association rules for a pair of entities is very strong, the relation is given a large weight. If the rules are weak, though, the relation is given a small weight.

For the weight of relations, we can for example use the confidence of the association rules. A confidence of 1 means that if the antecedent is changed, the consequent is also always changed, so the relation is strong (maximal). A confidence of 0.6 could then be used for a relation with a weight of 0.6. This way, no minimum confidence value needs to be set (which is not an intuitive task), yet the strength of the rules is still taken into consideration.

For lift-based relations, it is less obvious what measure to use as the relation weight. However, it is possible to use lift (and support) in order to select whether a rule is strong enough, and use the confidence for the weight of the relation.

Negative relations Besides assigning a weight to a relation between 0 and 1, it is also possible to use negative weights when there is a negative relation between two entities. The lift of two entities, as defined in Definition 7, is larger than one when there is a positive relation, and smaller than one when there is a negative relation. Currently, a negative relation is considered to be the same as no relation, but it is also possible to assign a negative weight to this relation. That way, entities which co-evolve less frequently than expected should be placed in different modules.

Statistical covariance and correlation In statistics, the covariance and correlation are standard techniques for measuring how much two variables are related. These techniques can also be used for measuring how much two entities are related. However, the covariance and correlation require a number of real-valued observations for each random variable, while we only have a number of boolean-valued observations. That is, instead of checking whether an entity is changed in a given commit, it is necessary to know *how much* that entity changed in a commit.

For this, we need a way to determine how much an entity changes in a certain commit. We can use a simple measure such as the percentage of lines of code in the entity which actually changed. A problem with this is that a small change on a line has the same weight as a complete rewrite of a line. Also, some changes might appear to be small, but in fact completely change the functionality of the code, while other changes seem like a complete rewrite of the code, where the functionality remains completely the same (for example implementing a faster sorting algorithm).

However, if a good measure for the extent of the changes of entities is defined, it is a nice idea to use this in order to calculate the covariance or correlation of the entities. These values can then be used to determine whether two entities are related (or negatively related). The advantage of this, is that small changes are also treated differently as large changes in an entity.

Aging of commits It is also interesting to consider the aging of commits. For the metric definitions used in this thesis, all commits contribute equally towards deciding whether two entities are related. A problem with this approach is that the source code evolved with every commit. The most recent commit resulted in the current version of the source code, but older commits changed older versions of the code. As a consequence, the co-evolution of two entities in an old commit might be based on an equally old version of the code, where the two entities were indeed coupled. This, however, does not mean that the entities are still coupled in the current version. Because of this, more recent commits contain more reliable information, and are thus more important than the older commits.

This problem was partially solved by only considering a limited number of commits instead of the complete history. It is however still the case that the older commits carry less important information as the most recent. Also, this requires a user to set the range of commits to use, which is not easy to decide on. Using more commits gives more information, but this extra information might not be reliable (based on an old version of the code).

A better solution for this problem is to assign a smaller weight to older commits. This way, the more recent commits contribute more than the old commits to deciding whether entities are related. Different aging schemes can be considered. A linear aging scheme, for example, assigns a weight of 1 to the most recent commit and 0 to the oldest commit, and for the intermediate commits the weight changes linearly. However, this scheme is likely to still give too large weights to old commits, so a better aging scheme is likely required.

It is probably better to consider an exponential aging scheme. With such a scheme, the weight of the commit decreases exponentially as the age increases. For example, for commit i of a total of n commits, the weight $1/e^{n-i}$ can be assigned to the commit. This ensures that the most recent commits have a large contribution, while slightly older commits are already considered to be much less important.

The weights of the commits can then be used for determining the support of item sets. That is, instead of counting the number of commits in which a pair of entities is both changed, the weights of these commits are summed. As the other definitions (confidence and lift) are based on the support, these change accordingly.

Relations between modules While the relations between entities are relatively easy to determine, there are more possibilities for deciding whether two modules are related. For two entities, we check how often they co-evolve and how often they evolve independently of each other. If certain requirements are met, the entities are considered to be related. A similar approach is possible for modules, where again a module can change in a commit, or does not change.

However, for modules there are also other possibilities. Definition 12 considered the possibility that modules are changed multiple times in the same commit, based on how many of its containing elements are changed. That is, if in a certain commit three entities in the module were changed, then the module is considered to have changed three times in that commit. Alternatively, Definition 10 considers two modules related whenever children of the modules are related.

Thus, there are different techniques for determining whether two modules are related. Besides the ones mentioned in this thesis, there may be other techniques. Specifically, the multiset-based relations of Definition 12 is not defined in combination with lift-based relations, because the lift cannot be calculated when the support can become larger than the total number of entities. To solve this, it might be possible to modify the definitions of support and lift such that this is possible, and multiset-based relations can be combined with lift-based relations.

Aggregating over modules The structure of source code is often hierarchical, where modules may contain submodules. But because the MQ assumes a flat structure, with entities grouped into modules, there is no single MQ value on the system level. The MQ is calculated for each module on which it is defined, and all these values are then aggregated by taking the average. However, there are other aggregation techniques available besides the average. A simple alternative is, for example, the median.

Especially when the distribution of individual MQ values is skewed, though, more sophisticated aggregation techniques might be required. For example, one could look at the Gini index [23] or the Theil index [57], which are both used to measure economical inequality. While the MQ should be maximized in all modules, it is also desirable for all modules to have an approximately equal MQ value, as opposed to some modules with a very good structure and others with a relatively bad structure. For this, inequality measures such as the Gini and Theil index might indeed prove to be successful alternatives to measuring quality directly. The problem, however, is that some inequality is usually better than bad quality, which should be overcome. A study on which aggregation technique is most useful for our purposes, has not been committed. It is important for future research to investigate this issue, because the average might indeed not be the best choice.

Finally, aggregation is currently performed by taking the average over all modules, regardless of the location of these modules in the hierarchy. Another option is to take the average at each level of the hierarchy separately, and then further aggregating these results. More specifically, the average MQ of all modules which are grouped together in the same (super-)module can be calculated, with the resulting value being used for aggregation of the MQ value one level higher in the hierarchy.

4. Metric calculation tool

The metric validation tool has one purpose: validating the metrics. As stated in Chapter 2, the tool is split into two components. The first is part of the library, which is shared by this tool and the optimization tool. This component contains all classes necessary for retrieving information from a software repository (both the commit history and the source code), and for calculating metric values for a given version of the code based on the commit history. The second component uses these classes for calculating the metric values of different versions of the source code, in order to compare these values of a version before and a version after restructuring. Both components are discussed in the chapter.

This chapter first discusses the requirements of the metric validation tool in Section 4.1. Next, the design is presented in Section 4.2. Finally, some implementation details are discussed in Section 4.3.

4.1 Requirements

Before we discuss the design of the tool, we first formulate the requirements. Obviously, the main requirement is that the tool should use data from a software repository for automatically validating the metrics. To do this, the tool needs to be able to retrieve information from a software repository, and it needs to be able to calculate metric values using this information.

The specific requirements are discussed in the remainder of this section. Note that we only provide an overview of the requirements. That is, detailed requirements of atomic features are not provided. The requirements listed here are used as guidance for the design discussed in the next section.

Flexible design We require the design to be flexible, in the sense that module-level (child-based, set-based or multiset-based) and entity-level (confidence-based or lift-based) techniques can be combined to form metrics. Similarly, metrics can be based on the information from any version control system, without depending on the version control system directly.

1. A module-level technique and an entity-level technique can be combined to form a metric.
2. There should be no direct dependencies between a metric and a repository/version control system.

In the future, it should be easy to add new metrics or new module-level and entity-level techniques, without requiring too many changes to other parts of the code (such as the code which actually performs the validation). The same applies for software repositories: it should be easy to add support for other version control systems.

3. Adding a new metric should be easy, requiring only well-defined changes.
4. Adding support for a new version control system should be easy, requiring only well-defined changes.

With "well-defined changes", we mean that it is clear how to modify the source code to add the new functionality. That is, obviously, the implementation of the new functionality needs to be added. However, the existing code should not have to be changed to accommodate for the new functionality, other than well-defined extension points informing the tool of the newly available functionality.

Subversion support The tool is required to support Subversion repositories. Both the commit history and the source code (in tree representation) need to be retrieved directly from a Subversion server. This way, users are not required to perform a complete checkout of the repository to validate the metrics.

5. Subversion repositories should be supported.
6. The commit history can be retrieved from the repository, based on a revision range.
7. The code structure can be retrieved from the repository, given a revision number.
8. All information can be retrieved directly from the Subversion server over an internet connection. That is, no checkout of the repository is required.

Information cache As retrieving data from a version control system over the internet can be slow, or even fails completely when no internet connection is available, all retrieved data needs to be cached locally. By caching the data locally, new validation experiments are faster (as the information does not need to be downloaded from the internet) and can even be performed without the need to have an internet connection (of course only if the data that is required is present in the cache).

9. All information retrieved from the repository should be cached locally.

The cache should be incremental. That is, all required information in the cache is retrieved from the cache, and only the data not available in the cache is retrieved from the repository. The newly retrieved data is then added to the cache for future use.

10. Data available in the cache is always loaded from the cache.
11. Newly retrieved data is added to the cache.

File-level granularity Currently, only analysis is performed where entities are files. Therefore, the tool only needs to support analysis at this granularity. However, in the design, the possibility of adding support for another granularity needs to be considered.

12. File-level entities need to be supported.
13. Possibility for adding support of another granularity needs to be considered in the design.

For this last requirement, it is important to decouple the entity granularity from other parts of the code. However, since the data that needs to be retrieved from a repository depends on the granularity, some coupling between these two parts of the code cannot be prevented.

Simplicity The metric validation tool is intended as a simple tool which can be used to validate metrics. It is therefore not important to have features such as a Graphical User Interface (GUI) or a configuration system. Specifically, the tool is not required to contain any elaborate features other than the ones listed in this section, but simplicity is preferred.

14. Simplicity is preferred over functionality not otherwise required by the listed requirements.

4.2 Design

A class diagram of the tool is shown in Figure 4.1. For readability reasons, only the classes are shown without any fields or methods. The diagram can roughly be divided into two parts: a part for retrieving information from a software repository and a part for calculating metric values.

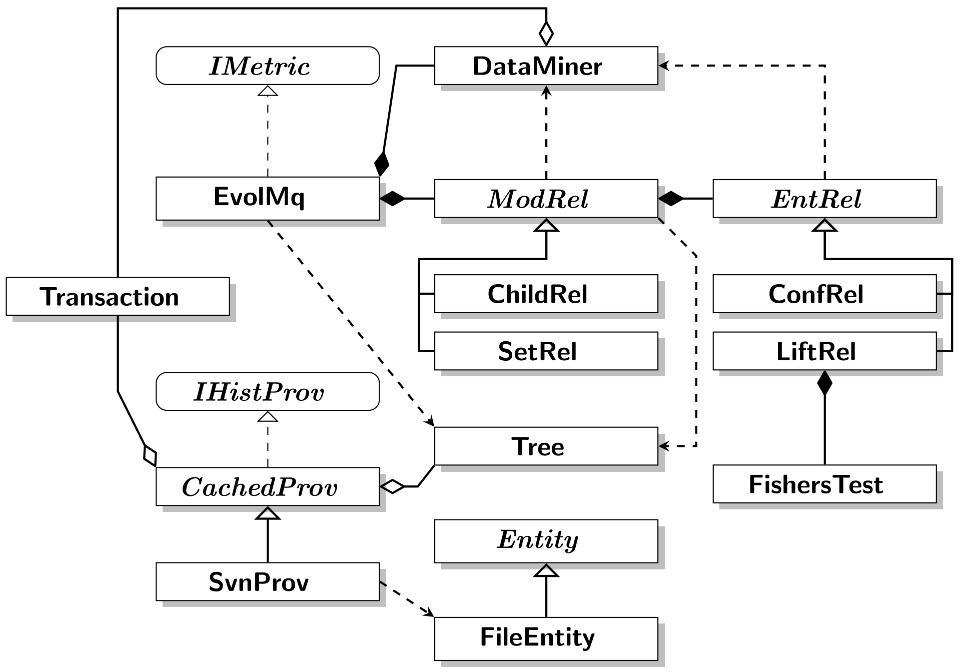

Figure 4.1: *Class diagram of the metric validation tool. For readability reasons, only the classes are shown without any fields or methods.*

Figure 4.1 only shows the classes which are present in a shared library (see Chapter 2). Besides this library, there is also a small and relatively simple executable which uses this library in order to perform metric validation. The design of the library is discussed in Section 4.2.1 and Section 4.2.2. The executable is discussed in Section 4.2.3.

4.2.1 Software repository

The interface `IHistProv` defines the functionality for a history provider. It defines two methods which should be implemented by all classes inheriting from it. To add support for a version control system (Requirement 4), this interface needs to be implemented.

The first method is `getTransactions()`, which should return a collection of `Transactions` to satisfy Requirement 6. A transaction contains information for a single commit in the version control system. Retrieving this information is, especially for a version control system such as CVS (which only tracks file versions instead of product versions), non-trivial. This topic has been studied by Zimmermann and Weissgerber [62] and Vanya et al. [58], and some considerations are discussed in Section 4.3.1.

The second method defined by `IHistProv`, satisfying Requirement 7, is `getTree()`, which returns a `Tree` representing the code structure at a specified revision. This revision is always the last revision in the revision range used to retrieve the commit history, thus returning the structure of the most recent version of the code.

The abstract class `CachedProv` provides the caching of transactions and the tree, and satisfies Requirements 9, 10 and 11. Any provider inheriting from this class automatically contains the caching functionality. `CachedProv`, however, does not have support for retrieving any information from a software repository. It only retrieves data from the cache, or if it is not in the cache, forwards the request to the subclass. The cache is stored in an SQLite database [52]. For more details on the cache, see Section 4.3.2.

The class `SvnProv`, finally, has support for retrieving all the needed information from a Subversion repository (Requirement 5). What data should be retrieved, is passed to this class on construction (that is, the range of commits which should be retrieved). Since this class inherits from `CachedProv`, caching of the data is handled automatically. For communication with the Subversion server (Requirement 8), the open-source library SVNKit [55] is used. See Section 4.3.1 for more information on how the data is retrieved from Subversion.

The class `SvnProv` uses the class `FileEntity` for entities at the file-level granularity (Requirement 12). This class inherits from the abstract class `Entity`. To add support for another granularity level (such as method-level entities), a subclass of `Entity` needs to be created, and `SvnProv` needs to be modified to add the functionality for retrieving data at this granularity (Requirement 13). Note that the code to retrieve the necessary data is dependent on the version control system used. Therefore, support for another granularity needs to be implemented in `SvnProv`, requiring changes to that class.

4.2.2 Metrics

The interface `IMetric` is the core of the metrics calculation part of the code. All metrics need to implement this interface, which defines a single method: `getValue(Tree)`. This method is responsible for calculating the metric value for the given tree. Adding support for a new metric, as required by Requirement 3, is thus a matter of implementing this interface. Any dependencies a metric has, need to be supplied to the constructor.

There is only one class implementing the `IMetric` interface, which is `EvolMq`. This class implements the MQ function as described in Chapter 3. It is composed of two other classes, which help it do this.

The first class contained in `EvolMq` is `DataMiner`. This class contains all the code for performing item set mining on the collection of transactions retrieved from the software repository. The algorithm used for the item set mining is discussed in Section 4.3.3.

The other class contained in `EvolMq` is `ModRel`. This is an abstract class responsible for detecting relations. This class only contains code for detecting relations between modules. For detecting relations between entities, the abstract class `EntRel` is used.

The class `ModRel` has two subclasses, `ChildRel` implementing relation detection using the child-based technique, and `SetRel` implementing relation detection using the set- or multiset-based techniques. These last two techniques are implemented in the same class due to the fact that they are very similar. Which of the two techniques is used, is determined by a boolean parameter supplied to the constructor of `SetRel`.

The classes `ConfRel` and `LiftRel` implement the two entity-level techniques (confidence-based and lift-based respectively). The parameters of the techniques are supplied as arguments to the constructor of the corresponding class. The class `LiftRel` contains the class `FishersTest` for performing the statistical test required for the lift-based technique. The implementation of this statistical test is discussed in Section 4.3.4.

The use of the abstract classes `ModRel` and `EntRel` make it possible to combine a module-level technique and an entity-level technique, to form a single metric. This part of the design thus satisfies Requirement 1.

Note that there is no direct dependency between a history provider and a metric. That is, the metric depends on the transactions and the tree, which are both retrieved by a history provider.

However, there is no dependency of a metric on how or from where this information is retrieved, thus satisfying Requirement 2.

4.2.3 Metric validation

Note that Figure 4.1 does not show any classes which perform the validation of the metrics. These classes are omitted for readability reasons. The validation of the metrics is mostly implemented in a single class, `MetricValidation`. This class requires two instances of `IHistProv`, one containing the tree and transactions from before a successful restructuring effort, and one containing the tree and transactions from after the restructuring.

Furthermore, the class `MetricValidation` needs to be informed of how to create instances of `IMetric`. This is done by supplying the class with an array of factories, capable of creating instances of `IMetric`. These factories are used to create the metrics. Finally, a list of parameters for the metrics is provided, giving `MetricValidation` sufficient information to construct all metrics which should be validated, and performing the validation.

The validation is thus performed in the class `MetricValidation`. All required information is passed to this class (two history providers, an array of metric factories, and metric parameters) upon construction. The configuration settings (used to specify which metrics to test, for which parameters, and on which repository) are given in another class as constants. The configuration is thus not stored in a file, not can it be set through any interface (GUI or command line). Changing configuration settings (such as changing which metrics to validate) thus requires recompilation of the tool (not the library). This results in the code of the tool itself to be as simple as possible (Requirement 14), by not containing an elaborate configuration system.

4.3 Implementation

The metric validation tool is implemented using JAVA. The tool does not have any GUI, nor is it possible to configure it from the command line or configuration files. The metric validation tool is kept as simple as possible, and as a result the configuration (specifying which metrics should be validated and on what project) is therefore handled in the code.

This section continues with discussing the implementation of parts of the tool, for which certain considerations need to be made. The implementation of these parts is discussed here, because decisions need to be made, or the implementation is otherwise interesting to discuss. The Subversion provider is discussed in Section 4.3.1. This section includes a discussion on the preprocessing performed on the SVN log. The cache system is described next, in Section 4.3.2. Section 4.3.3 then discusses the mining algorithm used for item set mining. Finally, Section 4.3.4 discusses how Fisher's exact test is implemented.

4.3.1 Subversion

When retrieving information from an SVN repository, it is worth thinking about any required preprocessing. That is, it is important to realize whether the data is immediately usable, or if it needs to be processed before it can be used.

For version control systems using file versions instead of product versions, such as CVS, preprocessing is required to group commits together [62]. While this is not necessary for SVN (as SVN already uses product versions), it is in some cases still a good idea to consider grouping related commits to accommodate for "micro-committing". Micro-committing is when developers

Transaction
location
revision
transaction

Transaction_cache
location
startRev
endRev

Tree
location
revision
tree

Figure 4.2: *The three database tables used by the cache system.*

make multiple, subsequent commits for a single change, for example committing once for each file changed.

However, no attempt is being made to group commits together in the `SvnProv` class. The reason for this, is that many projects have a "don't break the build" policy, which requires developers to test their changes before they commit them [31]. This implies that, in theory, each commit should contain a complete set of related changes. Therefore, it is, at least for our purpose, considered unnecessary to aggregate subsequent commits.

Another problem which can occur in the retrieved transactions, is commits which only update irrelevant files, such as license files or other non-source code files. As we are only interested in how different parts of the code co-evolve, transactions containing only such irrelevant files can be considered noise. Therefore, all such transactions are filtered out, ensuring that only source code files are included in the analysis.

This preprocessing is implemented by ignoring all files which do not end with `.java`. This is true both for the transactions and the source code tree. Furthermore, any transactions which become empty due to the filtering of non-source code files are ignored. This way, only JAVA source code files are considered in the analysis.

4.3.2 Cache

For caching the information retrieved from a software repository, a simple caching system is used. Both the transactions and the source tree are stored in an SQLITE database [52]. See Figure 4.2 for an overview of the database tables and fields used by the cache system. Note that there are no foreign key relations between the tables.

Each transaction is identified by a location and a revision. The location is a string containing the URL to the software repository and the revision identifies a single commit. The transaction itself is stored by serializing the JAVA object.

The source code tree is also identified by a location and a revision. The tree itself is again stored by serializing the object in memory. Note that while serialization is not an efficient operation, it does make the cache system easier, especially for the tree, since we can now retrieve it with a single, simple SQL query. Also note that storing the tree using serialization does limit the size of the serialized object to 1GB [53]. This limit is not expected to cause a problem in practice, as experiments have not found a tree larger than 1MB. However, if this does become a problem, the caching system needs to be modified.

Due to preprocessing of the SVN log (see Section 4.3.1), it is possible that a range of transactions contains "gaps". That is, it is possible that a transaction for revision 1 and a transaction for revision 3 exist, but no relevant transaction exists for revision 2. In order to check whether a transaction not present in the cache has never been retrieved, or simply is irrelevant, extra meta-data is stored. This meta-data contains information on the ranges of revisions for which it is known that all transactions within that range are present in the cache. That is, any absent transaction with a revision within that range is absent because the transaction is irrelevant.

4.3.3 Item set mining

For mining frequent item sets or association rules, a number of algorithms are available, such as the *Apriori* algorithm [3] and *FP-Growth* [26]. However, these algorithms are much more general than what we need. While these algorithms perform well in finding all frequent item sets of arbitrary length, we only require item sets consisting of a pair of entities.

The generalized algorithms achieve their good performance by pruning the search space. That is, when it is known that the item set containing items a and b is not frequent, than any superset of this item set is also infrequent and thus does not have to be considered. However, if only item sets of size 2 are required, this pruning does not offer much improvement in performance, as all item sets of size 3 or more are already ignored. Therefore, a simpler algorithm is used for frequent item set mining, which is shown in Algorithm 4.1. This algorithm is the same as implemented by the open-course JAVA DATA MINING PACKAGE (JDMP) [30]. JDMP was not used, as the algorithm is simple to implement. This saves us from depending on an external library for a single algorithm. Furthermore, it removes the need for an intermediate data structure required for JDMP, as the algorithm can use our `Transaction` type directly.

Algorithm 4.1 The item set mining algorithm used for calculating metrics based on evolutionary coupling.

```
Input: A list of transactions $T$
Output: A mapping from item set to transactions $S$
  $S(s) = \emptyset$ for all item sets $s$
  for each $t \in T$ do
    for each $e_1 \in t$ do
      $S(\{e_1\}) \leftarrow S(\{e_1\}) \cup t$
      for each $e_2 \in t$ do
        $S(\{e_1, e_2\}) \leftarrow S(\{e_1, e_2\}) \cup t$
      end for
    end for
  end for
  return $S$
```

The used algorithm simply loops over all transactions. For each entity in the transaction, the transaction is added to the set of transactions in which the entity occurs. Also, for each pair of entities in the transaction, the transaction is added to the corresponding set. Note that Algorithm 4.1 uses sets, which means that $\{e_1, e_2\} = \{e_2, e_1\}$, and thus $S(\{e_1, e_2\}) = S(\{e_2, e_1\})$. Also, this means that a transaction cannot occur twice for the same item set. Therefore, although the algorithm might appear to count transactions double for item sets of size 2, it is important to note that this is not the case.

Note that the algorithm results in a mapping for both single entities and pairs of entities. The reason for this is that both are needed in order to calculate the confidence and lift of a pair of entities. Furthermore, the set-based and multiset-based module-level techniques need the set of transactions for each entity.

4.3.4 Fisher's exact test

For lift-based relations, the p-value of a hypothesis test is calculated to determine whether two entities are related or not (see Chapter 3). For this purpose, Fisher's exact test is used. This statistical test requires the calculation of terms of the following form:

$$p = \frac{n_1!(n-n_1)!n_2!(n-n_2)!}{a!b!c!d!n!}$$

The meaning of the variables (n_1, n_2, a, b, c, d and n) are discussed in Chapter 3, and are not important for the discussion here.

The problem with this, is that the factorials in this term quickly become larger than the size of a JAVA integer ($13! > 6 \cdot 10^9$). This would mean that a library for big integers is needed for those calculations. Furthermore, factorials are defined inductively (that is, $n! = n \cdot (n-1)!$), so calculating $n!$ has a time complexity linear in the size of n.

This last problem can be solved by caching all calculated values. The largest factorial we need is $n!$, where n is the number of transactions. We can thus create an cache which never exceeds a size of n, with the value at index i equal to $i!$. This way, each time we need the value of a factorial, we only need to use a cache lookup.

As for the problem of the values getting too large to fit in a 32-bit integer (or a standard floating point data type), we can work with logarithms. Instead of calculating factorials, we can calculate "log-factorials" [59], which are defined as

$$\mathrm{lfac}(n) = \ln(n!) = \ln(n) + \mathrm{lfac}(n-1)$$

Log-factorials grow much slower than factorials, with even $\mathrm{lfac}(10^6)$ easily fitting in a 32-bit integer.

Now, because we have $p = e^{\ln(p)}$, we can first calculate $\ln(p)$ and then get our p back by taking the exponent. Calculating $\ln(p)$ can be done using log-factorials and the equality $\ln(a \cdot b) = \ln(a) + \ln(b)$.

$$\begin{aligned}\ln(p) = {} & \mathrm{lfac}(n_1) + \mathrm{lfac}(n-n_1) + \mathrm{lfac}(n_2) + \mathrm{lfac}(n-n_2) \\ & - \mathrm{lfac}(a) - \mathrm{lfac}(b) - \mathrm{lfac}(c) - \mathrm{lfac}(d) - \mathrm{lfac}(n)\end{aligned}$$

Note, though, that this technique causes rounding errors to accumulate. Therefore, the calculated p-value is not exact. This implies that when the actual p-value is close to the maximal p-value, a relation between two entities might be wrongfully accepted or rejected. However, when evidence for a relation (or a lack of a relation) is strong, it is less likely for this to cause a problem. Due to the accumulation of rounding errors, even for strong evidence the result can be wrong. However, thus use of log-factorials is still considered one of the better solutions [59]. As this problem is non-trivial to solve, the maximal p-value parameter should be taken as a guideline rather than an absolute threshold. Investigating to what extent this becomes a problem in practice, should be considered for future research.

Also note that p is a probability, and thus lies in the range $[0,1]$. This means that after taking the exponent, we do not get the same problem that the value is too large to fit in a 32-bit integer. Also, when p is close to 0, $\ln(p)$ becomes a very small negative number. However, since lfac grows so slowly, $\ln(p)$ does not become so small to cause an underflow in practice, as even a million transactions is not enough to cause this effect. Therefore, the above calculation can be done using only floating point arithmetic and we do not need any special libraries for handling large number. The log-factorials are still cached once calculated for the first time to speed up the calculation.

5. Metric validation

In order to investigate whether the metrics as defined in Chapter 3 are indeed capable of measuring the quality of the software structure, the metrics are validated on a case study. As the case study, we choose a large restructuring effort of the Spring Framework. This restructuring effort is considered to have been successful [51]. The metrics as defined in Chapter 3 are calculated for the code structure both on the code as it was before the restructuring, and on the code after the restructuring. The values are then compared, and are expected to show an increase in metric value, since the code structuring quality has supposedly improved.

The case study on which the validation is performed is first discussed in Section 5.1. Next, the methodology used for the validation is discussed in Section 5.2. This discussion includes an overview on what metrics are validated and for which choices of the metric parameters. Section 5.3 presents the results of the validation. Finally, the conclusion in Section 5.4 includes a discussion on which metrics give the most promising results, and are thus selected for the reengineering process.

5.1 Case

Validation is performed on `spring-webflow`, a subproject of the Spring Framework. During development of version 1.0, the Spring Framework was successfully restructured in order to reduce the component dependencies [51]:

> Restructuring effort successfully reduced project average component dependency from 48 to 27 (45%), largely by isolating and encapsulating the complexity of the Spring Web Flow engine implementation.

The final restructured version is version 1.0 RC4, and was released on October 4th, 2006. The previous version (1.0 RC3) was released on June 23rd, 2006. The validation is performed on the subproject `spring-webflow`, because the change log indicates that most of the improvement of code quality comes from this component.

In order to validate the metrics, we need to calculate the metric values both on a version of the code before the restructuring, and on a version after the restructuring. To do this, we need a part of the history which can be used for analysis. Since between the two version the code structure was changed, we actually need two parts of the history, because the history before the

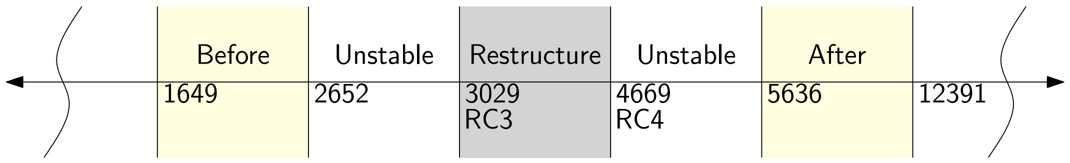

Figure 5.1: *The commit history for the* Spring Framework. *Immediately before and after the restructuring, the source code tree was unstable for a limited time (more restructuring or moving the complete source tree). Therefore, the selected parts of the history could not be chosen immediately adjacent to versions 1.0 RC3 and 1.0 RC4.*

restructuring contains changes to entities which might no longer exist after restructuring, and vice versa. Furthermore, for this same reason, it is important that neither part of the history contains any major restructurings.

For a visualization of the commit history, see Figure 5.1. Some restructuring was also performed before June 23rd, and some after October 4th. Therefore, the parts of the history selected for analysis do not end and start on these dates, ensuring that no major restructuring was performed within the selected ranges. That is, while the version before the restructuring (1.0 RC3) coincides with commit 3029, the code structure was unstable between commit 2652 and 3029. Therefore, the history must end no later than commit 2652. Similarly, the history after the restructuring was also first unstable, so it must start no sooner than commit 5636.

Stability of the commit history depends on the changes made to the code. During the unstable periods, files were moved, or a large number of files were deleted and added, with relatively few files being modified in a single commit. Furthermore, commit message were checked if they indicated large changes. In the stable periods, there were no strong indicators detected for restructuring. That is, the commits in the stable periods contained mostly file modifications (instead of deletions and additions) and the commit messages do not indicate any restructuring either. During the unstable periods, the maximum number of commits without restructuring was at most 100.

As for the length of the selected parts of the history, we again note the stability of the code structure. Before commit 1649, there are a few commits which significantly change the code structure (renaming and moving multiple entities). Therefore, the history before the restructuring is limited to the range 1649–2652. Within this range, there are approximately 500 relevant commits (commits in which the code of the `spring-webflow` project was modified). As a larger history provides us with more information (as long as it does not contain any restructuring), we select this entire range for analysis.

For the part after restructuring, the length of the history is not as constrained. After commit 5636, the code structure remains stable for more than 500 commits. However, since we want to ensure there is no bias towards either version, we choose to take the selected history before and after the restructuring to be the same length. This leads us to using the range 5636–12391, which again contains approximately 500 relevant commits.

For some statistics on the system used for the case study, we note that version 1.0 of `spring-webflow` contains 31 packages, which together contain a total of 242 classes. The depth of the package hierarchy is 3 (that is, the code is divided into packages, which may contain subpackages, and those subpackages can contain subsubpackages, which all contain classes). There are 11 packages at the highest level.

5.2 Methodology

Chapter 3 defines the techniques for quantifying the quality of the code structure. In order to get a metric, we need to combine an entity-level technique (confidence-based or lift-based), and a module-level technique (child-based, set-based or multiset-based). For the confidence-based technique, we also need to select a confidence combination technique (maximum, minimum or average). Finally, the entity-level techniques require parameters as input (that is, the minimum support, and minimum confidence or maximum p-value).

In order to keep the validation compact, not all combinations of a module-level technique and confidence combination technique, for confidence-based relations, are validated. Instead, the validation is split into three series of experiments. The first series investigates which module-level

#	Entity	Combiner	Module	Support	Confidence
1	Confidence	Maximum	Child	$\{0.002, 0.005,$ $0.01, 0.02, 0.05\}$	$\{0, 0.25,$ $0.5, 0.75\}$
2			Set		
3			Multiset		
4	Confidence	Minimum	Set	$\{0.002, 0.005,$ $0.01, 0.02, 0.05\}$	$\{0, 0.25,$ $0.5, 0.75\}$
5		Average			
6		Maximum			
#	**Entity**		**Module**	**Support**	**p-value**
7	Lift		Child	$\{0.002, 0.005,$ $0.01, 0.02, 0.05\}$	$\{0.01, 0.02,$ $0.05, 0.1\}$
8			Set		

Table 5.1: *An overview of the different metrics which are validated, along with the sets of parameters used for the metrics.*

technique gives the best results for confidence-based relations, for a range of minimum support and minimum confidence values. In this part, only the maximum is used as the confidence combination technique. The next series compares the three different confidence combination techniques, and only the set-based module-level technique is used. Finally, a comparison is made between the different module-level techniques for the lift-based relations. The results of all experiments are compared and discussed in the conclusion in Section 5.4. An overview of all metrics used in the validation, along with parameter values, is shown in Table 5.1.

The best value for the parameter values (especially minimum support and confidence) is not a priori known [2]. Depending on the data set (commit history), different values for these parameters might be more appropriate. Therefore, a set of different parameter values is used and compared.

The parameter values listed in Table 5.1 are chosen in such a way that they are neither too small nor too large. That is, the minimum support should correspond to at least one commit. For a history length of 500 commits, this implies that the minimum support should be at least 0.002 (as $0.002 \cdot 500 = 1$). The other values are chosen to be the same as used by Agrawal et al. [2], excluding 0.001, as this is too small for our history length. Also, experiments showed that hardly any relations satisfy the constraint that the support should be at least 0.05 (results are discussed in Section 5.3), implying that this value is sufficiently large for the validation. Therefore, we do not need to consider any larger values for the minimum support.

The minimum confidence should always be in the range $[0, 1]$. Again, the values are chosen such that we can detect how the MQ value is affected by the confidence. A minimum confidence of 1, however, is not used, because this means association rules need to have a confidence of 1 for a relation to be detected. But association rules this strong are expected to be rare, and a minimum confidence of 1 is thus considered to be too large. In fact, the results in Section 5.3 show that even a minimum confidence of 0.75 is so large, that not many relations are detected. Therefore, the largest minimum confidence used in the analysis is 0.75. The smallest value used is 0, which means that confidence is ignored.

Finally, the maximum p-values are chosen based on popular values as used in other literature. Often, a significance of 90%, 95% or 99% is used in order to determine whether a hypothesis is accepted or rejected [5, 6].

Note that the parameter values used are not the only valid choice. It is impossible to know a priori which parameter values are best. Therefore, a selection of different values is chosen with the goal that the validation study gives us more information on which ranges for the parameters are acceptable in the sense that these values give meaningful results. It is thus important to note that the validation study does not give us the perfect parameter values, but rather a range of acceptable values.

5.2.1 Confidence-based module-level comparison

The first part of the validation uses the confidence-based relations as defined in Definition 6. In order to combine the two confidence values of the association rules, we take the maximum of both values. The module-level technique is varied in this part of the validation, which gives us three different metrics: using child-based, set-based and multiset-based relations. These three metrics are shown in Table 5.1 as metrics 1–3.

For confidence-based relations, we need to supply two parameters: the minimum support and the minimum confidence. For each of the three metrics, we use the same set of parameter values. For the minimum support, we need the smallest value to correspond to at least one commit, as this ensures that the confidence value is always properly defined. For the minimum confidence there are no such strict requirements. Therefore, we opt for values evenly spread over the range $[0, 1)$. The parameter values are shown in Table 5.1.

5.2.2 Confidence-based combination comparison

The second series of experiments also focusses on confidence-based relations, but this time the confidence combination technique is varied and the module-level technique is always the set-based relations. This again gives us three metrics, shown in Table 5.1 with the numbers 4–6. The same parameters for minimum support and minimum confidence are used as for metrics 1–3.

5.2.3 Lift-based module-level comparison

Finally, the metrics using the lift-based technique, as defined in Definition 9, for detecting relations on the entity-level are investigated. For these metrics, there is no need to combine two values. Therefore, we only have the module-level technique which we can vary. Additionally, note that the multiset-based relations do not make sense in combination with the lift-based technique, because multisets produce a support value larger than the total number of commits. While this is not so much a problem for the confidence, the lift is not properly defined in that case. Hence, only the child-based and set-based techniques are compared.

For the lift-based technique, we need to supply two parameters. Again, we need to provide the minimum support, which is mostly important to ensure no division by zero occurs when calculating the lift. For the minimum support, we use the same values as for the validation of the confidence-based technique. The second parameter is the maximal allowed value for the p-value of the hypothesis test. Four popular significance levels for hypothesis testing are chosen. The metrics 7–8 in Table 5.1 show the parameter values used for the metrics.

5.3 Results

The results of the validation experiment are presented in three sections. First, the results for the confidence-based relations are given, where the module-level relations are determined using the three different techniques. Second, the module-level technique is kept constant and the confidence combination technique is varied. Finally, the results for the lift-based relations are presented.

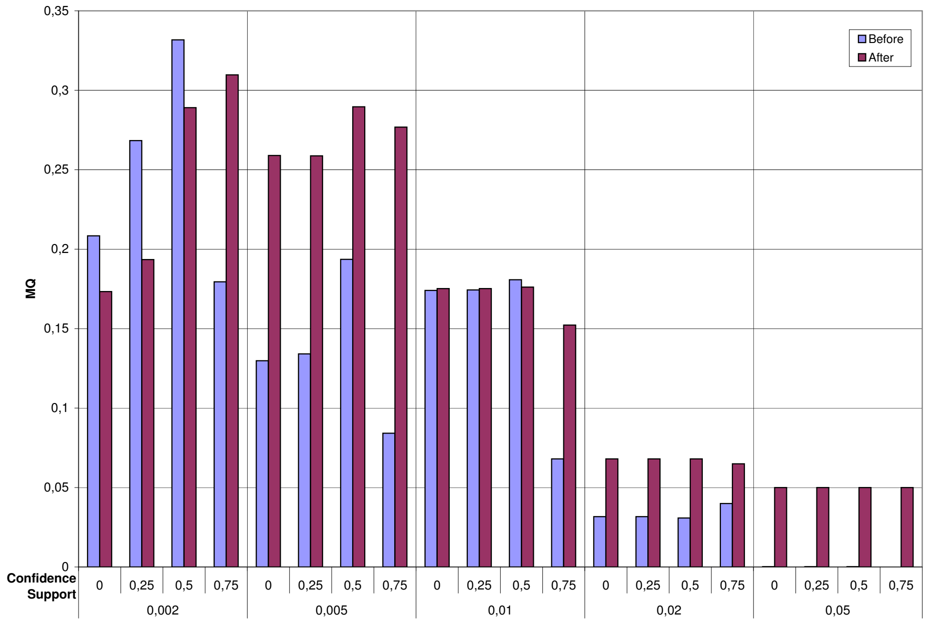

Figure 5.2: *Metric 1: A combination of low support (0.002) and low confidence (at most 0.5) shows a decrease in MQ value. For a high support (0.05), hardly any relations are detected. A support of 0.01 gives unexpected results.*

Note that for each set of results, there is a single figure for each technique. For each combination of parameter values (support and confidence, or support and p-value), a pair of bars is shown: one bar for the metric value before restructuring and one for the value after restructuring. The height of the bars indicates the metric value.

A discussion of all results and a comparison on which metrics are most promising for use in the reengineering process is given in Section 5.4. This section is only intended for presenting the results and making some basic observations.

5.3.1 Confidence-based module-level comparison

The first set of results is for the confidence-based technique. Here, we compare the three different module-level techniques with each other. Thus, we always combine the confidence values using the same technique, specifically by taking the maximum. The results for the child-based, set-based and multiset-based techniques are shown in Figures 5.2, 5.3 and 5.4, respectively.

The results show that the MQ value after the restructuring increases for some parameter values, while it actually decreases for other parameter values. Such a decrease appears for low support and confidence values. In other words, this happens when too many files are considered to be related to each other. Indeed, on closer inspection, we see that in these cases both the coupling and cohesion increase, where the coupling increases more than the cohesion. This effect can in these cases be explained by the fact that so many pairs of entities are considered to be related, that the cohesion is already near its maximum value of 1. Indeed, in the cases where the MQ value decreases, the cohesion shows a typical change from 0.72 to 0.80, while the coupling is increased from 0.38 to 0.51 (example values taken from Metric 1 for support 0.002 and confidence 0.5).

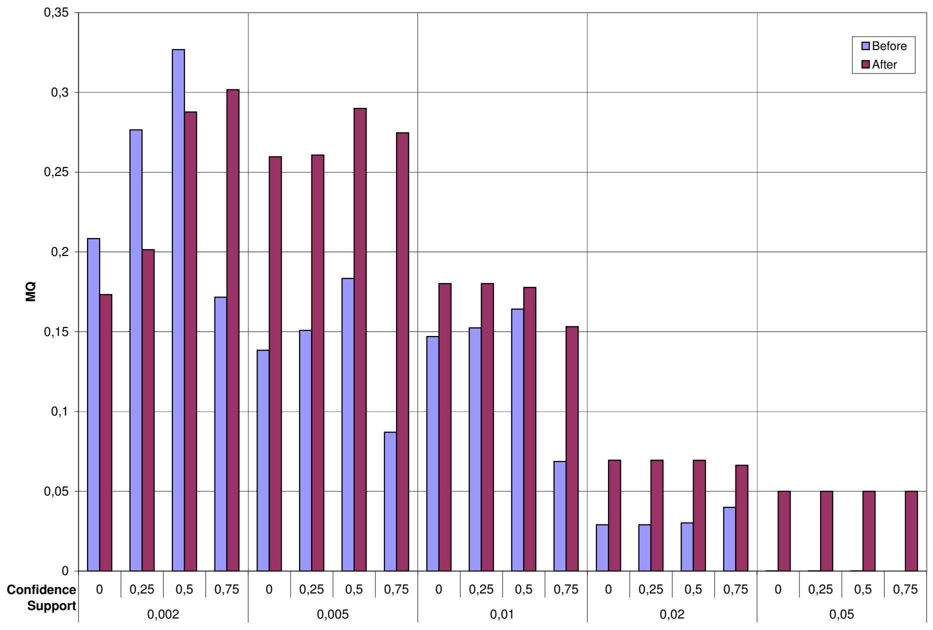

Figure 5.3: *Metric 2: A combination of low support (0.002) and low confidence (at most 0.5) shows a decrease in MQ value. For a high support (0.05), hardly any relations are detected.*

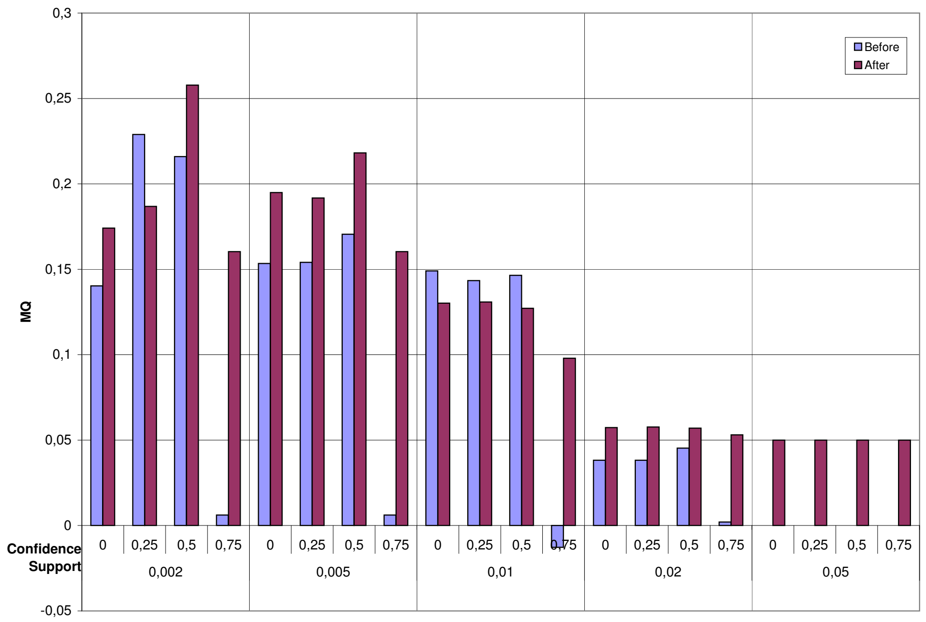

Figure 5.4: *Metric 3: A combination of low support (0.002) and a confidence of 0.25 shows a decrease in MQ value. For a high support (0.05), hardly any relations are detected. A support of 0.01 gives unexpected results.*

Furthermore, we see that for a minimum support of 0.05, the resulting values are (almost) the same for all minimum confidence values and all metrics. This is because hardly any relations are detected in these cases, which is especially true for the coupling. This means that a minimum support value of 0.05 (which for the tested 500 commits corresponds to $0.05 \cdot 500 = 25$ commits) is too large to give meaningful results: hardly any files were modified that often. Even for a support value of 0.02 (10 commits), the results show that not a lot of relations will be detected due to the high support value. It is important that at least some relations are still detected, because without any relations, the MQ value is always equal to zero. That is, we cannot compare the quality of two code structures if there are no relations. Even with a small number of relations, the difference in quality becomes hard to measure.

In general, we see that a high confidence value corresponds to the more significant increases in MQ value. Obviously, when a too high minimum confidence value is chosen, results will go to 0 as no relations will be able to meet the requirement. In particular, this is why the MQ value with the multiset-based technique before restructuring is close to zero for a minimum confidence of 0.75.

As the minimum confidence is increases from 0 to 0.5, we see that the MQ value also increases. A larger minimum confidence means that fewer relations are detected, and in this case more inter-package relations are omitted than intra-package relations. However, for a minimum confidence of 0.75, the MQ value is actually decreased. This is most apparent for the values before restructuring. In this case, it turns out that the coupling is already approaching its minimum value and the cohesion thus decreases more than the coupling. This suggests that a minimum confidence of 0.75 is likely too high (too few relations are detected).

When we compare the results of the three experiments (the three module-level techniques), we see that the results of using the child-based technique and those for using the set-based technique are fairly similar. The largest difference is when a minimum support of 0.01 (5 commits) is used. For small confidences, the child-based technique does not show any significant changes in the MQ value. On the other hand, the set-based technique does show some improvement in MQ value, although it is not a large difference.

The multiset-based technique, however, does show significant differences. For a high confidence value (0.75), a large increase in MQ value is shown, while for a smaller minimum confidence, there is a much smaller increase or even a decrease in MQ. Because of this, we might be tempted to think that multisets in combination with a high minimum confidence is a good metric. However, the metric value before restructuring is in this case close to zero, with both the coupling and cohesion component smaller than 0.1. This indicates that before the restructuring, there are not many relations detected, so a minimum confidence of 0.75 is actually too large to detect sufficient relations. Indeed, the metric values after restructuring decrease more when the minimum confidence is increased from 0.5 to 0.75 than is the case for the other two metrics.

A particular interesting and unexpected part of the results occurs for a minimum support of 0.01. Here, especially for the child-based and multiset-based metrics, we see that the restructuring did not result in a large increase in MQ (or even a decrease). This does not correspond with what one would expect from the results for a support of 0.005 and 0.02, where this decrease is not seen. The exact reason for these strange results is unknown, but it is possible that the minimum support is already getting too large here, starting to cause problems.

Thus, the choice of parameters and module-level technique is important. A minimum support which is too small gives unstable results, as too many relations are detected. That is, it becomes much more important to choose a correct value for the minimum confidence in order to limit the number of relations. By selecting a larger minimum confidence value, this can thus be compensated somewhat. A support which is too large is equally bad, because this eliminates too many relations. A minimum support of 0.005 seems optimal for this case. Of course this might be different depending on the project, but it should be larger than 1 commit, and smaller than 0.02.

As for the confidence, again it should be large enough, but not too large. Here, a value of 0.5 seems acceptable. Smaller than 0.5 means too many relations are detected, increasing the coupling while the cohesion is already almost maximal. Larger than 0.5 appears to reduce the number of relations too much.

Finally, with the correct choice of minimum support and confidence, all three module-level techniques do a good job. However, for the multiset-based and child-based techniques, it is more important to make the right choices for the parameter values than for the set-based technique. The set-based technique only shows a decrease in MQ for a minimum support of 0.002, and other problems (too few relations detected) only occur with a minimum support larger than 0.02. For the other two techniques, there are more unexpected or problematic results. Therefore, the set-based approach shows to be the most promising module-level technique, especially when combined with a minimum support of approximately 0.005 and a minimum confidence of 0.5.

5.3.2 Confidence-based combination comparison

Secondly, we look at the effect of choosing a different confidence combination technique. Here, the module-level technique is kept constant, specifically set-based relations are used. The results for using the minimum, average and maximum for combining the two confidence values are shown in Figures 5.5, 5.6 and 5.7, respectively. Note that Figure 5.7 is the same as Figure 5.3, which is repeated here to provide a more complete picture on the effect of using a different confidence combination technique.

The first thing we note is that for the results for minimum and average, the values of coupling and cohesion are both decreased, as obviously there are fewer relations for which the condition is true. It is, however, more interesting to investigate how the confidence combining technique influences the difference between coupling and cohesion, so how it influences the MQ value.

For minimum and average, we can see that a large value for the minimum confidence gives a negative value of the MQ before the restructuring. This means that the coupling is larger than the cohesion. With these alternative combination techniques for such high confidence values, a relation is only considered to exist when the evidence for such a relation is very strong.

Furthermore, we see the same trend in all three figures, in that a decrease in MQ value is detected for small support and confidence. However, for a support of 0.002, a minimum confidence of 0.5 is sufficient to show an increase for minimum and average, while for maximum a minimum confidence of 0.75 is required for that. It is still the case, though, that for all three, a larger confidence value causes the MQ value before restructuring to drop faster than the MQ value after restructuring.

Regardless of the confidence combination technique, it is still useful to select a large value for the minimum confidence. It is still a good idea to use a minimum confidence of 0.5. Any smaller does not give the cohesion enough room to grow, and any larger does not allow the coupling to decrease.

As to which of the three techniques is the best, this is not immediately clear from these results. The fact is that they show a similar trend between the code quality before and after the restructuring. The only major difference is that using the minimum or average combination technique makes it possible to use a small minimum support. That is, a minimum support of

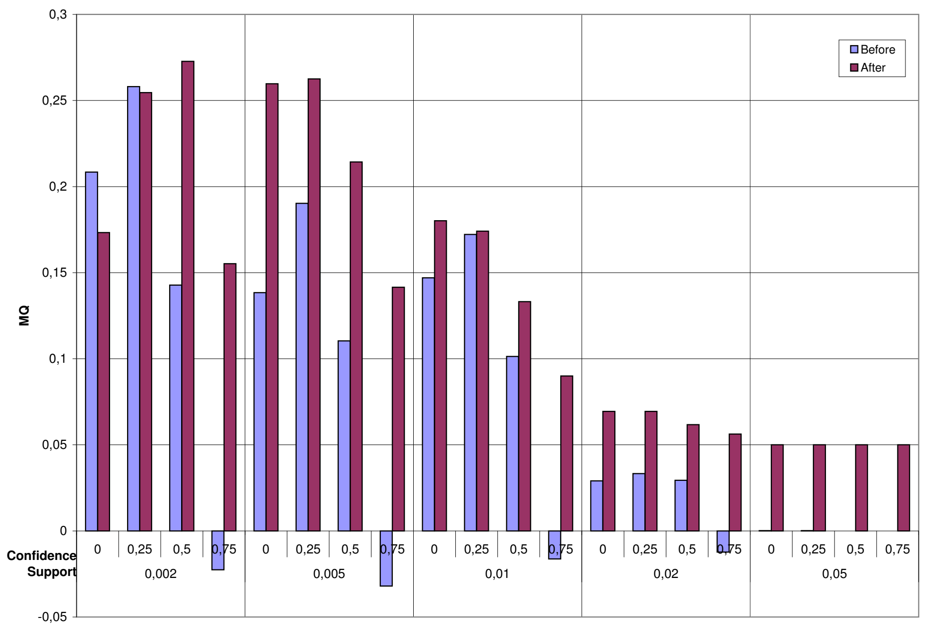

Figure 5.5: *Metric 4: A combination of low support (0.002) and low confidence (at most 0.25) shows a decrease in MQ value. For a high support (0.05), hardly any relations are detected.*

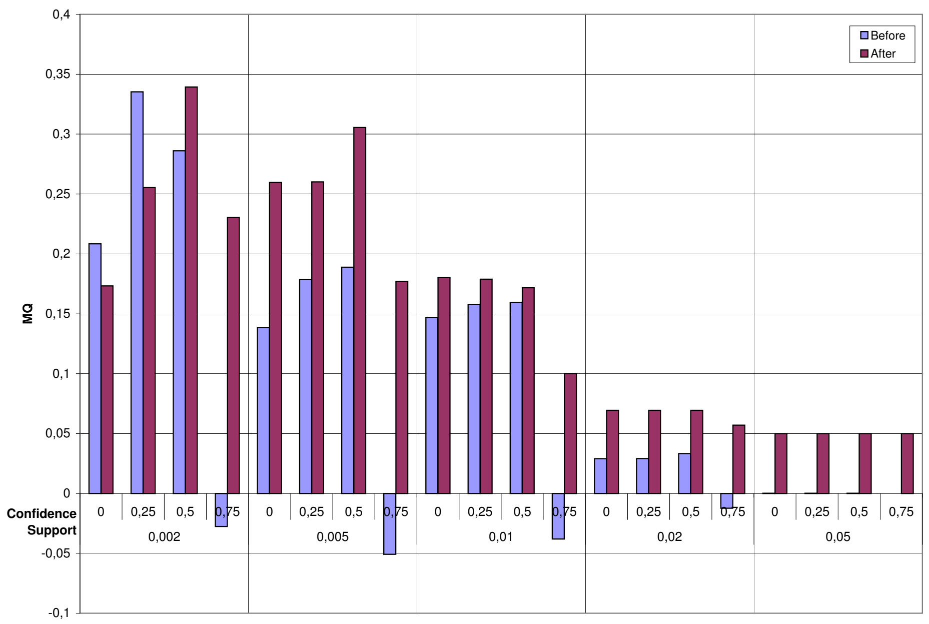

Figure 5.6: *Metric 5: A combination of low support (0.002) and low confidence (at most 0.25) shows a decrease in MQ value. For a high support (0.05), hardly any relations are detected.*

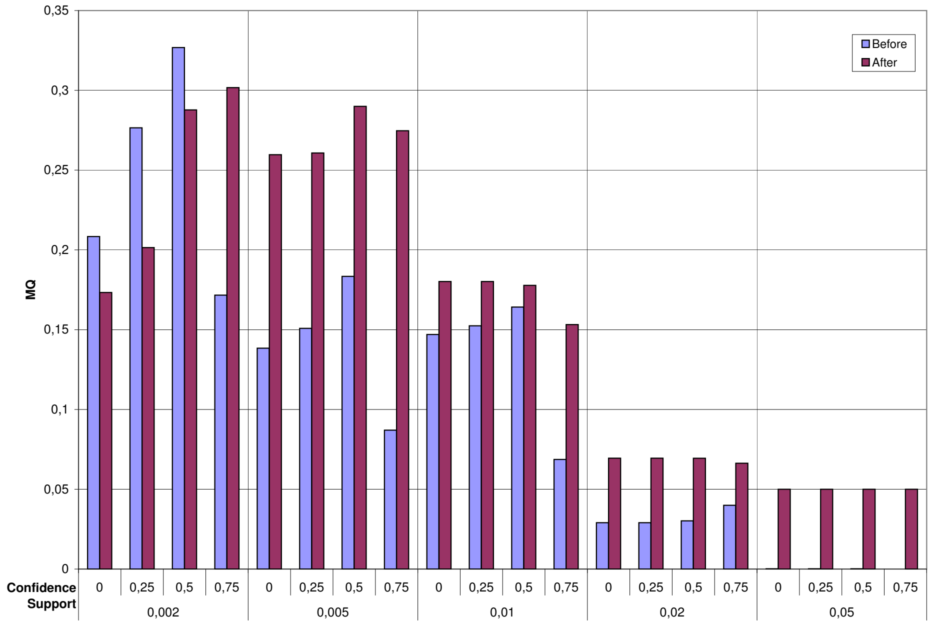

Figure 5.7: *Metric 6: A combination of low support (0.002) and low confidence (at most 0.5) shows a decrease in MQ value. For a high support (0.05), hardly any relations are detected.*

0.002 (1 commit) is acceptable with a minimum confidence of 0.5. This is because when taking the minimum or average, the confidence is capable of filtering out sufficient potential relations, while when the maximum is taken, too many relations are found.

5.3.3 Lift-based module-level comparison

Finally, we present the results for using the lift-based technique for detecting relations on the entity level. For lift-based relations, there are only two applicable module-level techniques. The results for using child-based relations are shown in Figure 5.8, and the results for set-based relations are shown in Figure 5.9.

One thing we can see from these results, is that for a minimum support of 0.002 (1 commit), the MQ value before restructuring clearly increases as the the maximum p-value increases, while the metric value after restructuring shows a decreasing trend. For a large maximum p-value, more relations are detected in total. This combined with a too small minimum support again means that too many relations are detected for the metric to give a sensible value: almost all entities are related to each other.

For different values for the minimum support, a different p-value does not seem to have much effect. With only the p-value changing, the MQ value remains more or less constant, as the same entities are considered to be related. Therefore, as long as a proper value for minimum support is chosen (again, approximately 0.005 seems a good choice), the choice of p-value is less important, with anything in the range 0.01–0.1 being acceptable. However, a small p-value can somewhat compensate for a too small minimum support.

We see unexpected results for the child-based metric with a minimum support of 0.01, which also occurred with the confidence-based metrics. Again, it is unknown what causes this, but it does

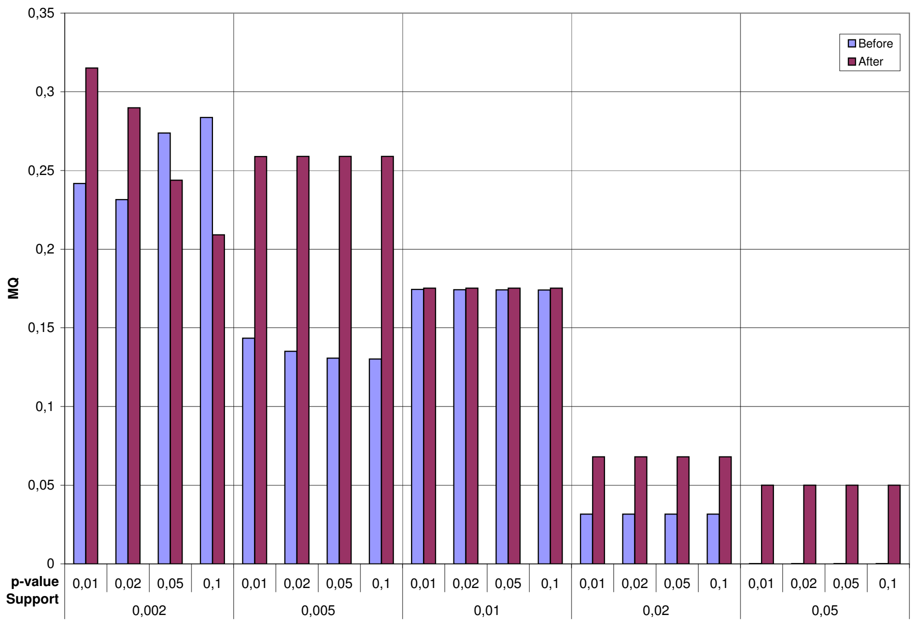

Figure 5.8: *Metric 7: A combination of low support (0.002) and high p-value (at least 0.05) shows a decrease in MQ value. For a high support (0.05), hardly any relations are detected. Results for a support of 0.01 are unexpected.*

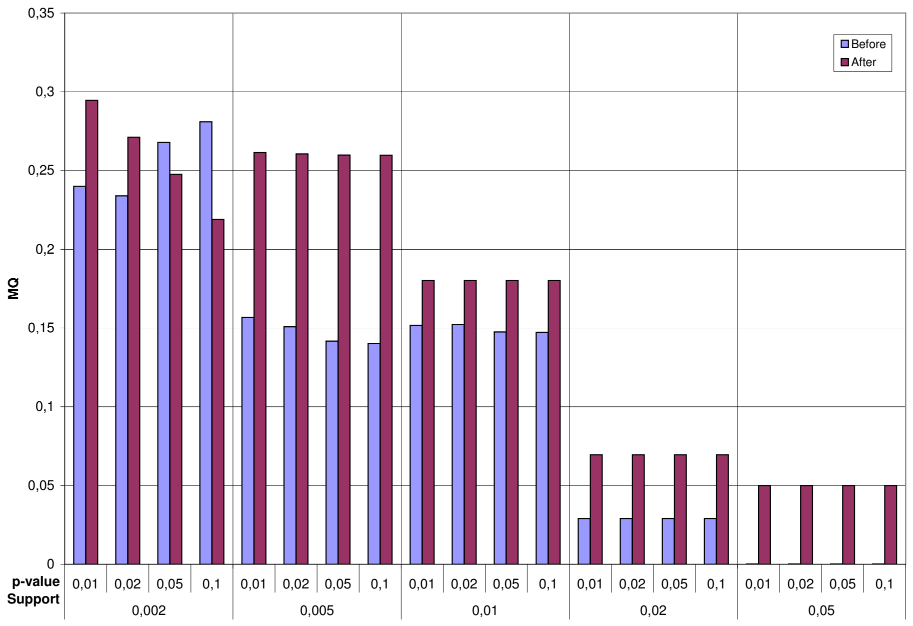

Figure 5.9: *Metric 8: A combination of low support (0.002) and high p-value (at least 0.05) shows a decrease in MQ value. For a high support (0.05), hardly any relations are detected.*

suggest that a minimum confidence of 0.01 might already be too large. However, this problem does not occur to such a large extent for the set-based metrics, indicating that this module-level technique is indeed the best choice.

Comparing the child-based technique with the set-based technique, we see something similar as for the confidence-based relations. Both work well when a correct minimum support value is chosen, but the set-based technique makes the choice for the minimum support less important. Therefore, also considering the strange results for a support of 0.01, the set-based technique is likely a better choice.

5.4 Conclusion

Considering all the results, we can conclude that the set-based technique shows to be the most promising. While its results are not all that different from the other module-level techniques, it does have the advantage that the choice of the parameter values seems to be less important. More precisely, more minimum support values are acceptable when the set-based approach is used than with another technique.

Besides more promising results, the set-based approach also has another advantage. The multiset-based technique has the problem of the support values of modules potentially getting larger than the total number of commits, which means it is not an intuitive technique. This is also the reason it cannot be used in a lift-based metric. Therefore, the restructuring of the code will only be done using the set-based technique.

The results for the different techniques for combining the two confidences of an association rule into a single value, do not indicate a clear preference for a specific technique. The main difference is again that using the minimum or average makes the requirements on the minimum support less strict as when using the maximum. However, the most important reason for comparing the different techniques, is to prevent unidirectional relations (relations which are strong in only one direction and weak in the other direction) from causing too many entities to be moved. Therefore, it is meaningful to compare these techniques when performing restructuring experiments.

When comparing the confidence-based metrics with the lift-based metrics, we see that there is not a very clear difference indicating which is better. However, the confidence does have theoretical problems as discussed in Chapter 3. While this problem did not present itself in the results of the validation, it is possible that it becomes more apparent on other cases.

Also, the confidence-based metrics require several parameters to be precisely chosen. Besides choosing a minimum support, which is also required for the lift-based metric, we also need to select a minimum confidence and a confidence combination technique. The meaning of these parameters is not very intuitive, and this makes it difficult to choose the correct values. In fact, the optimal values for these parameters likely depend on the specific project on which the metric values will be calculated.

On the other hand, for the lift-based metric, we only need to select a p-value. This p-value is easier to understand intuitively and thus makes it easier for a user to make a reasonable choice. The p-value can be considered roughly equivalent to the probability of a false relation being detected. That is, the probability a relation is detected even though it is not a true relation, can be limited by the p-value.

The restructuring experiments in Chapter 8 are performed with the most promising metrics. Based on the results presented in this chapter, this means all metrics use set-based relations. Furthermore, the lift-based technique and the confidence-based technique are both compared, where for the confidence-based approach all three confidence combination techniques are used. A minimum support of 0.005 showed to give the most promising results, as did a minimum confidence of 0.5 and a maximum p-value of 0.01.

5.4.1 Threats to validity

While the validation of the different metrics is performed such that the results are correct and meaningful, the validity of the results can be threatened by a number of issues. The main problem is that only a single case study was considered. It is thus possible that on another system, the results will be different. For example, another case study might expose the problem that confidence has, where the confidence is high only because the support is high.

As the confidence can be high only because the support is high, even when the association rule is not significant, it is possible that the confidence-based metrics are unreliable. That is, it is possible that relations are detected between entities which should not be related. In order to prevent this from becoming an actual problem, the results were compared with the results of the lift-based metrics. As the lift does not have this theoretical weakness, and results are similar for all metrics, we can thus conclude that this problem likely did not have a significant impact on the results.

Another problem is that there is no guarantee that the structure of the code indeed improved. While the developers who performed the restructuring of the Spring Framework do claim that the structure is improved [51], this could not be independently verified. It is also not feasible to verify this, because no metric can be fully trusted and manual inspection is labor intensive, difficult and error prone. However, the developers do report decreased dependencies between components. Therefore, it is reasonable to assume that the restructuring effort was indeed successful.

Finally, there is a risk of "over-validation". A lot of metrics with parameter values are validated, all on the same case. Because of this large number of different metrics, but only a single case, it is possible that for some values the metric shows promising results by chance. However, since several fundamentally different techniques were used and they all showed a similar trend, this is unlikely to have occurred here. That is, if the positive results are the result of random chance, then it would not be expected that the results would show a similar trend for all the different techniques and parameters used.

6. Search-based software engineering

Many problems in the domain of software engineering are related to optimizing potentially conflicting goals. A good example of such a problem, is finding a good architectural design. This problem consists of trade-offs between conflicting goals [46]. Finding a perfect solution to such a problem is often infeasible, as defining an algorithm to solve the problem can be problematic [27]. *Search-based Software Engineering (SBSE)* attempts to solve these problems by applying meta-heuristic search-based optimization techniques, such as a genetic algorithm. These techniques attempt to optimize specified objectives in order to find near-optimal solutions.

The problem of automated reengineering studied in this thesis is an optimization problem. Here, we try to improve the structure or modularization of our source code by optimizing one of the metrics defined in Chapter 3. However, rearranging entities in modules also carries the undesirable effect of all knowledge about the structure of source code by the developers becoming obsolete.

Attempting to reduce the impact of this problem, Abdeen et al. [1] suggest not changing the module structure, but only moving entities into other, existing modules. This way, the module structure of the code remains constant, and the developers are not confronted with a completely changed module structure.

However, moving a large number of entities still causes developers to have trouble finding those entities later. Therefore, we in fact want to improve the structure of the source code without moving too many entities. That is, while maximizing the MQ metric, we also want to minimize the changes made. This chapter discusses the use of a genetic algorithm for optimizing these two conflicting objectives.

A more extensive motivation for the use of a genetic algorithm is given in Section 6.1. Next, we discuss our solution to the problem in more detail in Section 6.2, specifically by presenting how the problem of automated reengineering is solved using a genetic algorithm. Here, we limit ourselves to optimizing only the structure of the code. In Section 6.3, we investigate how we can optimize multiple objectives simultaneously, thus minimizing the number of changes made. Section 6.4 then discusses some related work, which includes a discussion on existing approaches for automated reengineering. Finally, Section 6.5 discusses the open problems and other ideas which are interesting for future research.

6.1 Motivation for a genetic algorithm

Using a genetic algorithm for solving an optimization problem is not an immediately obvious choice. After all, using a genetic algorithm offers no guarantee that the optimal solution is found, or even a solution which is guaranteed to not be far from optimal [29]. While a genetic algorithm finds increasingly better solutions the longer it runs, it is impossible to say whether it is already close to an optimal solution, or the optimum is still a long way to go.

Other optimization techniques might indeed be better in some cases. If an algorithm can be devised which can find the optimum in a reasonable time, then indeed a genetic algorithm is a bad choice. Furthermore, approximation algorithms or greedy algorithms might be able to find better solutions in less time, and give guarantees on how optimal the found solution is. However, for the case of automated reengineering, or more specifically, software restructuring, we can argue that a genetic algorithm is in fact a reasonable choice.

Source code restructuring is concerned about grouping related entities into modules. This problem is basically a graph clustering problem, where vertices in a graph are grouped together in clusters, such that vertices in the same cluster are similar or connected [49]. The graph clustering problem has been shown to be NP-complete for a number of objective functions [50], implying that it cannot be solved in polynomial-time, unless P=NP [22].

It should be noted that the NP-completeness of graph clustering does depend on the objective function used for optimization. That is, the use of a different objective function changes the problem, potentially making it easier to find an optimal solution. Šíma and Schaeffer [50] showed for a number of objective functions that the problem is indeed NP-complete, including when optimizing only the intra-connectivity of our MQ (thus ignoring the inter-connectivity). NP-completeness of the clustering problem with the use of MQ as the objective function, has not been proven. However, while a formal proof remains an open problem, it is expected that the general clustering problem is indeed NP-complete [44].

It is uncertain whether source code restructuring using MQ is NP-complete, or not. However, it is generally accepted that it is indeed likely to be an NP-complete problem [45]. It is thus unlikely that a polynomial-time algorithm exists which exactly solves our problem.

Furthermore, in our problem, we are not looking to optimize the source code structure on MQ alone. By changing the structure of the code, the modularization might indeed improve, but it does also have a downside. When a developer has worked a long time with the source code of a system, he eventually knows how the code is structured. By changing this structure, the developer might get into trouble finding the piece of code he is looking for: it is not where he remembered it to be.

To solve this problem, we want to minimize the number of changes made to the code structure. At the same time, we also want to optimize the structure of the code. More precisely, we want to achieve a good modularization with the fewest changes possible. Clearly these two objectives are conflicting, as improving the structure of the code necessarily means changing the structure of the code.

Optimizing multiple conflicting objectives can be difficult to achieve, but a genetic algorithm can be used for optimizing multiple, conflicting objectives at the same time [10]. This way, we can find a code structure with better modularization, such that as few changes as possible to the code are made to achieve this structure. The details of multi-objective optimization are discussed in Section 6.3.

Thus, as it is unlikely for an efficient algorithm to exist which can solve our problem, the use of a genetic algorithm is reasonable. Furthermore, a genetic algorithm has the added advantage that multiple conflicting objectives can be optimized simultaneously. For other techniques, the optimization of multiple objectives is problematic [10].

6.2 Genetic algorithm

A genetic algorithm [29] is an iterative algorithm. That is, the algorithm executes a predefined number of iterations. Each iteration starts with a *population* of *individuals*. An individual is a candidate solution for the problem (that is, a valid solution, but not necessarily the optimal solution). This population of individuals is evolved into a new population in a number of steps to

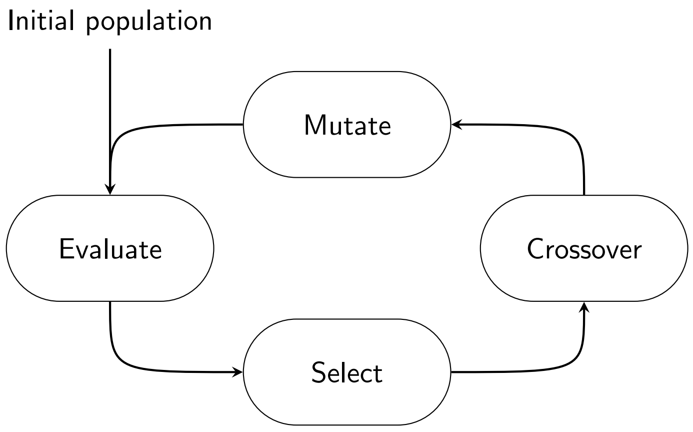

Figure 6.1: *An overview of the steps in a genetic algorithm.*

complete a single iteration. This process is then repeated until some stopping criterion is reached. A schematic overview of the steps in a genetic algorithm is shown in Figure 6.1.

When the genetic algorithm starts executing, it first generates an initial population. This population is often generated randomly. When the initial population is generated, the first iteration starts. In each iteration, we first evaluate all individuals in the population by calculating their objective values. Next, the best individuals are selected for the crossover operation. This operation takes two "parent"-individuals and recombines them to generate two "children". These children are then randomly mutated, resulting in a new population for the next iteration. The algorithm stops when some stopping criterion is reached, often when a predefined maximum running time or number of iterations is exceeded.

The details of a genetic algorithm are not discussed here. The implementation of the steps for our problem of restructuring is discussed in the remainder of this section.

In our case, the population is initialized with the current tree structure of the source code. That is, the individuals are not randomly generated. This is done because we want to change the existing code structure. If we would initialize the population with randomly generated individuals, we might be able to find a better structure due to the larger solution space explored. However, this would come with a cost of individuals which require more changes to the existing structure.

Since we want to minimize the number of changes required, it makes sense to start with the existing structure, as parts of the solution space unreachable from this initial population contain only individuals which require a large number of changes, and are thus undesirable solutions. By using only the existing structure in the initial population, we in fact evolve more individuals in the desirable part of the solution space (that is, in the part of the solution space containing the individuals which require few changes to the existing structure).

Evaluation of the individuals is done by calculating the objectives. That is, we calculate the metric value for each individual. Then, we can compare two individuals to determine which one is better. The best solutions are then selected to become parents in the crossover step.

The crossover recombines two individuals, the parents, to generate two new individuals, the children. This is done by taking part of the first parent and part of the second parent, which together form a child. These parts are generally parts of a list, combining a sublist of the first parent with a sublist of the second parent. Figure 6.2 shows how crossover works for lists.

The main problem here is that individuals in our case are trees instead of lists. We thus need a way to define the crossover operation on our tree. Since the only changes made to the tree in our

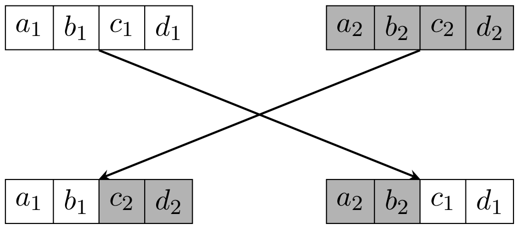

Figure 6.2: *The crossover operation on lists. The first part of the first parent and the second part of the second parent are recombined to form one child. The other child consists of the first part of the second parent and the second part of the first parent.*

case is moving entities to other modules (as suggested by Abdeen et al. [1] to reduce the problem of developers not understanding the new structure), we can represent each individual as a list. In this list, each entity corresponds to an index in the list, and the value at that index corresponds to the module this entity is in. That is, the entity e_i in module M means that the value in our list at index i corresponds to module M.

The final step in an iteration of a genetic algorithm is mutation. In this step, the children resulting from the crossover operation are mutated. That is, the list is randomly changed in some positions. In our case, that means some entities are randomly moved to another module.

6.3 Multiple objectives

With metrics defined to quantify the modularization quality of a code structure, and an optimization problem optimizing such a metric, we have all the ingredients we need for optimizing the code structure. However, this code structure can be completely different from the original structure, with every entity moved to another module. This is undesirable, because it can be difficult for developers to understand this new structure, if they were already familiar with the old one. Therefore, we need a way to minimize the changes we make to the code structure.

In this section, we investigate how we can optimize the code structure, but at the same time minimize the changes made to the code in the process. First, we investigate how we can quantify change in Section 6.3.1. Here, we define two metrics capable of measuring how much the original code structure was changed to get the new one. In Section 6.3.2, we discuss multi-objective optimization and how we can use it to optimize our two objectives. Finally, Section 6.3.3 discusses how we select the best solution based on the trade-off between the two objectives.

6.3.1 Quantifying change

To measure how much one structure needs to be changed to get another, we can use several techniques. Abdeen et al. [1] count the number of entities they move, and use the resulting value to quantify change. However, the problem with this simple technique for our case, is that this does not take the hierarchy of the module structure into account.

For example, consider a code structure as shown in Figure 6.3. Now, consider we move entity e_1 to another module. Since e_1 is in module M_{11}, which is part of module M_1, we can say that e_1 is also (indirectly) part of module M_1. Therefore, moving e_1 to M_2 can be considered to have more impact on the structure than moving it to M_1 or M_{12}.

Taking this hierarchical structure into account, we define two metrics which measure the extent of a set of changes. But first, we need to define a function which returns all modules on the path

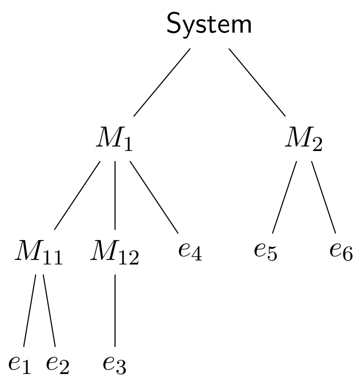

Figure 6.3: *Tree representation of a code structure consisting of modules M_1, M_{11}, M_{12} and M_2 and entities e_i for $1 \leq i \leq 6$.*

from the root to a given module. That is, a function returning all ancestor modules, including the given module.

Definition 14 (Ancestor modules)**.** Let M be a module and let $\mathrm{parent}(M)$ denote the parent of M (that is, the module directly containing M). The function anc is then recursively defined as follows.

$$\mathrm{anc}(M) = \begin{cases} \{M\} & \text{if } M \text{ is the root} \\ \{M\} \cup \mathrm{anc}(\mathrm{parent}(M)) & \text{otherwise} \end{cases}$$

Using this definition, we can define the first metric for quantifying change. This first metric counts the number of edges in the tree each entity has to traverse to reach its new position. That is, for each entity, the metric is defined as follows.

Definition 15 (Move penalty)**.** Let e be an entity which was moved from module M to module M'. Furthermore, let $A = \mathrm{anc}(M) \cap \mathrm{anc}(M')$ be the common ancestors of M and M'. The penalty for entity e is then defined as

$$\mathrm{penalty}(e) = |\mathrm{anc}(M)| + |\mathrm{anc}(M')| - 2|A|$$

In the example tree structure in Figure 6.3, this means we have a penalty of 1 when e_1 is moved to M_1. When e_1 is moved to M_{12}, the penalty is 2. Finally, we have a penalty of 3 when it is moved to M_2.

The second metric for measuring the extent of a change assigns a weight to each edge according to an exponential scheme. That is, edges from the root are given a weight of $1/2$, edges one level further down in the tree a weight of $1/4$, and so on. In general, the weight of the edges on level i of the tree, where $i = 1$ for edges coming from the root, have weight $1/2^i$. Note that instead of $1/2$, we can take any value smaller than 1.

Definition 16 (Exponential move penalty)**.** Let e be an entity which was moved from module M to module M'. Furthermore, let $A = \mathrm{anc}(M) \cap \mathrm{anc}(M')$ be the common ancestors of M and

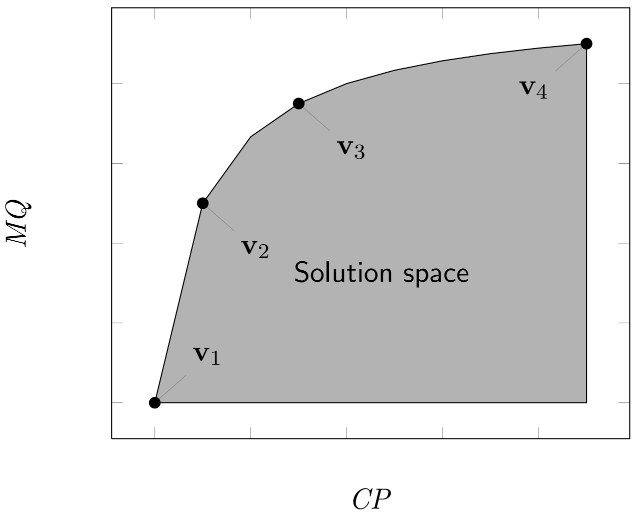

Figure 6.4: *A plot of the solution space containing the feasible solutions for optimizing MQ and CP simultaneously. Four Pareto-optimal solutions are marked.*

M'. The penalty for entity e, with $0 < r < 1$, is then defined as

$$\begin{aligned}\text{penalty}(e) &= \sum_{i=1}^{|\text{anc}(M)|-1} r^i - \sum_{i=1}^{|A|-1} r^i + \sum_{i=1}^{|\text{anc}(M')|-1} r^i - \sum_{i=1}^{|A|-1} r^i \\ &= \frac{2r^{|A|} - r^{|\text{anc}(M)|} - r^{|\text{anc}(M')|}}{1-r}\end{aligned}$$

For the example tree in Figure 6.3 and $r = 1/2$, we have a penalty of $1/4$ when e_1 is moved to M_1. Moving e_1 to M_{12} results in a penalty of $1/2$, and moving it to M_2 gives us a penalty of $5/4$.

The actual metric for measuring how much a series of changes affects the code structure, is referred to as the *Change Penalties (CP)*. This value is obtained by adding the penalties for each entity together. That is, the metric value for a system containing the entities E is equal to

$$CP = \sum_{e \in E} \text{penalty}(e)$$

Here, $\text{penalty}(e)$ is the function from Definition 15 or from Definition 16.

6.3.2 Multi-objective optimization

We now have two objectives we want to optimize simultaneously. The first is one of the metrics defined in Chapter 3, and the second is one of the metrics defined in Section 6.3.1. As these metrics are conflicting, we want to find a solution with a good trade-off between the two metrics.

A genetic algorithm has support for multi-objective optimization [10]. With multi-objective optimization, we place all the objectives in a vector, and we then optimize the value of this vector. That is, our objective becomes optimizing a vector $\mathbf{v} = (v_1, \ldots, v_k)$, where $v_1, \ldots, v_k$ are the values of the k objectives we want to optimize.

For this, consider our case where we want to optimize two objectives. The solutions found can be plotted in a graph, where the x-axis contains the CP value and the y-axis the MQ value of the solution. See Figure 6.4 for an example of such a plot.

When optimizing two objectives simultaneously, there are multiple solutions with a trade-off between the objectives. Some solutions are better than others. When all objectives in the vector $\mathbf{v}$ are more optimal than the ones in vector $\mathbf{v}'$, then $\mathbf{v}$ is better in total. We than say that $\mathbf{v}$ $dominates$ $\mathbf{v}'$. However, when one of the objectives is better for $\mathbf{v}$ and the other is better for $\mathbf{v}'$, then neither vector is strictly more optimal than the other, and neither vector dominates the other.

Because of this, multi-objective optimization returns multiple optimal solutions. Each such solution is said to be $Pareto\text{-}optimal$. A Pareto-optimal solution is optimal in the sense that, although it might still be possible to improve an objective, this necessarily comes with the cost of degrading another objective. Multi-objective optimization thus returns multiple solutions, each with a different (optimal) trade-off between the objectives. The best solution can then be selected from a limited set of Pareto-optimal solutions.

6.3.3 Selecting the best solution

A way to select the best solution, is taking a look at the plot of all Pareto-optimal solutions. These solutions generally form a curve as shown in Figure 6.4 [10]. In this plot, we can see that the first few changes contribute more to the increase in MQ, while the later changes are increasingly less helpful. Note, however, that the curve can also take other forms, as long as they are monotonically increasing [10].

A curve as shown in Figure 6.4 suggests that it might be worthwhile to make some changes to improve the MQ, but not opt for the best MQ. That is, the solution marked with $\mathbf{v}_4$ is likely not the best choice, because although it has a higher MQ than $\mathbf{v}_3$, this comes at a relatively high cost in CP. Similarly, $\mathbf{v}_1$ has a smaller CP, but it comes at a cost of a relatively low MQ. The solutions marked $\mathbf{v}_2$ and $\mathbf{v}_3$ show a better trade-off between MQ and CP.

Deb and Sundar [11] proposed an approach to selecting one (or more) solutions from the set of Pareto-optimal solutions. To do this, they calculate the normalized distance of each solution to a reference point. The distance d of a solution x to the reference point z is calculated as

$$d = \sqrt{\sum_{i=1}^{k} \left(\frac{f_i(x) - z_i}{f_i^{max} - f_i^{min}} \right)^2}$$

Here, $f_i(x)$ denotes the value of objective i for solution x, and f^{max} and f^{min} denote the maximal and minimal value objective i takes, respectively. The reference point is a vector containing objective values based on a posteriori knowledge. Originally, the reference point is selected by a human. However, we can also use the optimal values of all objectives (that is, f^{min} for minimized objectives and f^{max} for maximized objectives) [16].

Now, we select the solution which is closest to the reference point. Using this selection technique makes it possible to completely automate the restructuring process, even when multiple objectives are used. Note that the selection process uses a posteriori knowledge on the shape of the solution space. That is, it is not possible to use the distance d as an objective in the optimization process, as z, f^{max} and f^{min} are not known at that time.

6.4 Related work

The use of genetic algorithms or other meta-heuristic search techniques for improving the code structure is not new. Several other studies have been performed in this area. This section discusses these studies and compares them to the current work. The two main differences between this thesis and other research, are the use of evolutionary coupling and the use of multi-objective optimization.

6.4.1 Clustering software systems

Large and complex software systems are often difficult to understand. In order to help developers understand the structure of a software system, clustering algorithms have been used to automatically detect subsystems. Studies which used clustering algorithms to identify subsystems in software are discussed in this section.

Doval et al. [13] note that module dependency graphs (which show the dependencies between modules as edges between nodes) are often large and complex. Therefore, it is not always easy to understand such graphs. Partitioning the graph in clusters, or subsystems, of highly related modules can, however, help in understanding the graph. To automatically generate a partitioning of a module dependency graph into clusters, they use a genetic algorithm. For this, they use the MQ as their objective function. Results of a case study show that their technique is successful in finding a good clustering.

No code is actually restructured in the study by Doval et al. [13]. They only partition an existing module dependency graph into clusters in order to aid in the understanding of the structure of code. Furthermore, no relation detection techniques are required, because they start with a module dependency graph containing all the relations between the modules. Therefore, their approach is similar to other graph clustering studies [49, 50], with the exception that an objective function is used which measures software modularization quality.

Mitchell [44] also looks at the problem of clustering software systems to identify subsystems. The main difference is that now the module dependencies are not known beforehand. These dependencies are automatically determined by analyzing the source code. A modified version of MQ is used as the objective function.

Praditwong et al. [48] again use a modified version of MQ for automatically clustering software systems. However, they also use other objectives which are optimized simultaneously using multi-objective optimization. Other metrics used are the number of inter-edges and intra-edges, the number of clusters and metrics related to the size of clusters. They show that single-objective optimization of MQ outperforms multi-objective optimization using MQ in combination with the other metrics. However, when the concepts of coupling and cohesion, which together make up MQ, are considered separately, the multi-objective approach gives better results. That is, the multi-objective approach finds solutions with higher cohesion than the single-objective approach, and solutions with lower coupling. Finally, it is noted that while multi-objective optimization is able to produce better solutions, this comes at a cost of increased computational cost.

Research on clustering software systems often uses meta-heuristic search techniques, such as genetic algorithms, for identifying subsystems. The choice for these search techniques is attributed to the NP-complexity of the problem, making these algorithms a good choice. The main difference with research in this area as compared to this thesis, is that no code structure is actually modified. The algorithms are intended to discover code structure.

In this thesis, however, we already have a code structure, and the goal is to improve this structure. This comes with the additional problem that while we do want to improve the structure, we also want to keep the changes to a minimum. This is because developers are already familiar with the existing structure, and learning the new structures takes valuable time. Therefore, we have to take measures to prevent this from becoming an actual problem.

6.4.2 Search-based software maintenance

An approach to automated reengineering which is guided by optimizing the values of software quality metrics is proposed by O'Keeffe and Cinnéide [46]. Three quality functions (flexibility, reusability and understandability) are defined as a linear combination of eleven metrics, which all measure the quality of some aspect of the source code. By using multiple metrics, the authors argue that the approach is not limited to a single aspect, but can cope with the trade-offs in object-oriented design. The design of the source code is then optimized using search-based software engineering, where the three quality functions are used as an objective for the optimization. To determine which refactorings can legally be applied without changing program behavior, pre-condition checking was used. The refactorings are performed by modifying the abstract syntax tree of the Java code, thus effectively modifying the code. Results show that understandability, and to a lesser extent flexibility, are suited as quality functions for this purpose.

O'Keeffe and Cinnéide [46] do not consider the fact that changing the code can be problematic for developers who are familiar with the existing code structure. Furthermore, multiple objectives are optimized in the sense that multiple metrics are used, but these metrics are combined into three functions (flexibility, reusability and understandability) using a weighted sum. That is, the optimization process is a single-objective optimization, using one of the three functions as objective. Finally, the three functions used measure a different aspect of code quality than the metrics used in this thesis. That is, they do not look at the modularization of the code, but only consider the class hierarchies. Methods and fields are moved only to superclasses or subclasses, and are not allowed to move to any class in the system. Classes are not grouped into packages.

6.5 Future work

In this chapter, we discussed an optimization technique which can be used for automatically restructuring source code. However, other techniques for solving this problem are also available. This section discusses some possible future work in the area of automated software reengineering. Furthermore, open problems are identified and discussed.

Meta-heuristic search Because of the difficulty of the problem, we concluded that using a meta-heuristic search technique is a good approach for solving the restructuring problem. For this, we used a genetic algorithm. There are, however, also other meta-heuristic search techniques available. Examples are hill-climbing [47] and simulated annealing [35].

An especially interesting possibility for future research, is to implement multiple search techniques and run experiments with all of them. This way, the results can be compared, and statements can be made on which technique is most appropriate for solving the problem.

Multi-objective optimization We already used multi-objective optimization for the purpose of achieving the highest increase in MQ at the lowest cost in CP. However, other objectives can be identified which might be worth using in the optimization process. Most significantly, it is interesting to investigate whether splitting inter-connectivity and intra-connectivity (which together make up MQ) into two competing objectives gives better results than when they are combined in the MQ.

Furthermore, we can identify other objectives which we might want to optimize. For example, we can use metrics which are determined from source code analysis, such as the number of classes per package. It is thus interesting to check whether these additional metrics help in getting better results.

Solution selection In this chapter, we discussed a technique which is used to get a single solution from a set of Pareto-optimal solutions. However, this is not the only existing technique for selecting a solution [16]. Therefore, future research can try using other techniques and comparing the results. This way, it could be possible to make claims on which technique for selecting a single solution performs best for the specific problem of software restructuring.

NP-completeness As noted in Section 6.1, the problem of restructuring is NP-complete at least for some objective functions. However, NP-completeness has not been shown in general (that is, unrelated to the objective function). In specific, it has not been shown to be NP-complete when the MQ is used as objective function. However, it is expected that this problem is also NP-complete.

Therefore, future research can focus on finding an NP-completeness proof for the specific problem of restructuring using MQ, or even for the general problem of restructuring using any objective function. Obviously, alternatively it is also possible to show that the problem can, in fact, be solved in polynomial time. However, this result is not expected.

7. Optimization tool

For the restructuring of source code, the restructuring tool is developed. This tool implements an optimization algorithm which modifies the structure of source code such that the value of a metric is optimized. Thus, this tool needs to be able to calculate the values of the evolutionary coupling metrics discussed in Chapter 3. For this task, the same classes are used as the metric validation tool uses, which are discussed in Chapter 4. This is achieved by means of using a shared library, as discussed in Chapter 2.

The restructuring tool is implemented using the JAVA optimization framework OPT4J [39]. This framework already takes care of a large part of the functionality. That is, it already provides us with optimization algorithms and a GUI. Therefore, we only need to implement a library which is loaded by OPT4J, providing the classes specific to the problem of restructuring. As OPT4J uses GOOGLE GUICE [24] to handle dependency injection, our classes also use GOOGLE GUICE for the same purpose.

This chapter discusses how the library is extended for the added functionality required by the restructuring tool, and how the library is used by the restructuring tool. The requirements of this tool are first discussed in Section 7.1. Then, we give an overview of the design of OPT4J and how the required functionality is implemented in the library in Section 7.2. Finally, we discuss the use of dependency injection and GOOGLE GUICE in Section 7.3.

7.1 Requirements

The main purpose of the restructuring tool is that it optimizes the structure of source code, based on evolutionary coupling metrics. For our purposes, the tool is only used for research into automated restructuring, so it should provide us with validation results instead of just an improved structure. These results are used in Chapter 8 for investigating whether the proposed evolutionary coupling metrics are well-suited for automated restructuring.

The specific requirements are discussed in the remainder of this section. Note that we only provide an overview of the requirements, used as guidance for the design discussed in the next section. Furthermore, all requirements as discussed in Chapter 4 also apply for the optimization tool. Here, only the additional requirements are listed.

Flexible GUI The restructuring tool is required to have a graphical user interface. This interface should be flexible, in the sense that the user can select which metrics to use, and which history provider to use (currently, only a provider for SVN repositories is supported). Furthermore, the GUI should allow the user to change the parameters of the metrics and other configuration settings. Finally, the GUI should provide means to track the progress of the optimization process with graphs showing the current optimal values of the metrics.

1. The restructuring tool should have a GUI.
2. The GUI should allow to select which components to use (metrics and history provider).
3. The GUI should allow the user to change metric parameters and other configuration settings.

4. The GUI should provide graphs showing the current optimal metric value to track the progress of the optimization process.

Multi-objective optimization The tool should provide support for multi-objective optimization. That is, it should be possible for the user to select multiple metrics, which are then optimized simultaneously using true multi-objective optimization.

5. Multi-objective optimization should be supported.

Multi-objective optimization results in multiple Pareto-optimal solutions (see Chapter 6), of which the best one is selected when optimization finishes. While currently only one technique for selecting the best solution is implemented (see Chapter 6), it should be possible to add other techniques later.

6. Of multiple Pareto-optimal solutions, one solution should be selected.
7. Adding a new solution selection technique should be easy, requiring only well-defined changes.

With "well-defined changes", we mean that it is clear how to modify the source code to add the new functionality. That is, obviously, the implementation of the new functionality needs to be added. However, the existing code should not have to be changed to accommodate for the new functionality, other than well-defined extension points informing the tool of the newly available functionality.

Validation Support for validating the optimization process should be present. For validation, it is important that multiple runs of the optimization process can be executed automatically, removing the need for a user to manually start each run. Furthermore, the results of all runs should be accumulated and written to disk at the end of the validation process.

8. A specified number of runs of the algorithm (using the same settings) can be executed automatically.
9. The results of all runs are written to a file on disk after all runs are completed.

Finally, validation should include independent analysis of the results, by also calculating the metric values afterwards using an independent part of the history. For more details on the validation methodology, see Chapter 8.

10. An independent part of the history is used to calculate MQ values for both the original and improved structure, after the optimization process is finished.

Genetic algorithm The tool should use a genetic algorithm for the optimization of the code structure. However, it should also be considered that other optimization algorithms can be added in the future. Therefore, the user should be able to select the optimization algorithm which should be used from the GUI.

11. The user can select which optimization algorithm to use, in the GUI.
12. A genetic algorithm should be available to use as optimization algorithm.

7.2 Opt4J

For the restructuring tool, the optimization framework Opt4J [39] is used. This framework is used, because it already takes care of most of the requirements, namely requirements 1, 2, 3, 4, 5, 11 and 12. Only one other framework, jMetal [14], could be found which comes with a complete GUI and support for multi-objective optimization, but for this framework, satisfying Requirement 2 requires more work.

As discussed in Chapter 2, the restructuring tool is split into an executable and a library. The executable is actually completely provided by the used optimization framework OPT4J. The library is shared between the restructuring tool and the metric validation tool. As Chapter 4 already discussed the design of the part of the library used by the metric validation tool, this is not repeated here. Instead, only the extensions made for the restructuring tool (that is, code required for the optimization) are discussed here.

The OPT4J framework defines a number of interfaces which can be implemented. Which implementation of an interface is used, is defined in `Module`s. All classes only depend on interfaces, and the dependency injection framework GOOGLE GUICE is used to inject the correct implementations (as defined in the `Module`s). GOOGLE GUICE and the `Module`s are discussed in Section 7.3. This section only discusses the interfaces implemented in the restructuring tool, for a complete overview of the architecture of OPT4J, see the official OPT4J documentation [38].

It is important not to confuse a GOOGLE GUICE `Module` (defining type bindings) with a module in the code structure (containing entities). To make this distinction clear, the former will always be capitalized, while the latter is not.

In order for the optimization process to work, we need to implement at least five interfaces: `Phenotype`, `Genotype`, `Creator`, `Decoder` and `Evaluator`. Furthermore, for the validation of the results, we need to implement the `OptimizerStateListener` interface.

Phenotype and Genotype A solution in OPT4J is represented in two ways, both as a phenotype and as a genotype. A genotype is the genetic representation of a solution, and all operators of the optimization process are applied on it. That is, for a genetic algorithm, the crossover and mutate operators are applied on the genotype. A phenotype is the decoded representation of the genotype. That is, it is the direct representation of a solution.

The phenotype in our case is a tree representing the structure of source code. More specifically, it is the directory tree, indicating in which directory each file resides. It represents the hierarchical structure of directories containing (sub)directories and files. Thus, the class `Tree` presented in Chapter 4 implements the interface `Phenotype`. This interface in fact does not define any methods, it is only a marker interface.

The genotype is the genetic representation of a tree. It should be simple in the sense that crossover and mutate operations can be performed on it efficiently. As these operations are already defined by OPT4J for the predefined genotype classes, we extend the genotype `SelectMap-Genotype` with the class `TreeGenotype`. This genotype defines a mapping, in our case a mapping from entities to modules. Note that the structure of the modules is kept constant, only entities can be moved to other modules. For more details on the genotype, see Chapter 6.

Creator, Decoder and Evaluator The `Creator`, `Decoder` and `Evaluator` classes define how new solutions (genotypes) are created, how genotypes can be decoded into phenotypes and how to calculate the objective values of phenotypes, respectively.

Creating new genotypes is a process which is only performed at the start of the optimization, and it is intended to get an initial population of solutions. In our case, we want to start with the current tree structure. Therefore, we implement the interface `Creator` with the class `Data-Loader`. This class contains the initial tree structure and encodes this structure into a genotype representation in its only method `create()`.

Decoding a genotype is the process of transforming a genotype into a phenotype representing the same solution. Thus, the `Decoder` interface is implemented by a class `TreeDecoder`,

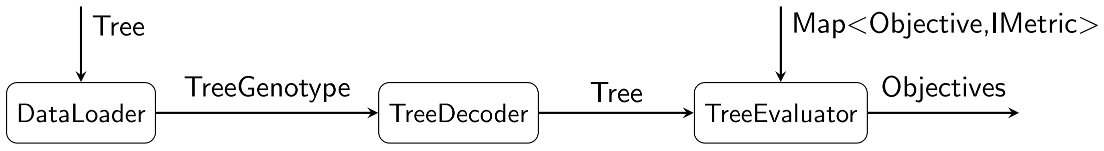

Figure 7.1: *Overview of the implementations of the main interfaces defined by* Opt4J*. Constructor arguments are drawn above a class. Methods arguments are drawn as horizontal incoming arrows, and return values are horizontal outgoing arrows.*

which decodes the `TreeGenotype` into a `Tree`. This is done in the method `decode()`.

Finally, we need to implement the interface `Evaluator` for evaluating a solution. This is done with the class `TreeEvaluator`, which contains a set of metrics (classes implementing `IMetric`, see Chapter 4). When this class needs to evaluate a tree, it calculates the metric value for the tree, for all required metrics. These metric values are then returned as a vector by the method `evaluate()`.

An overview of the process from creation to evaluation is shown in Figure 7.1. This shows that the creator contains a tree and encodes it in a genotype whenever a new genotype is required. Each time a genotype needs to be decoded into a phenotype, the decoder performs its task. Finally, the evaluator contains a set of metrics and calculates and returns the metric values of a tree when a tree needs to be evaluated.

Note that Figure 7.1 contains two types which were not yet discussed. `Objective` describes a single objective which is to be optimized. This includes a name of the objective and a direction (maximize or minimize). The type `Objectives` contains all the objectives used in the optimization, and assigns a value to each one. Finally, the set of metrics contained in the evaluator is represented as a map from `Objective` to `IMetric`. This is done such that the metric values can be calculated and correctly assigned to objectives in the type `Objectives`.

Optimizer state listener With the five interfaces mentioned above all implemented, the optimization process can be executed. However, this does not yet include any functionality to validate the restructuring process. For this, we need to implement two interfaces: `OptimizerState-Listener` and `OptimizerIterationListener`. In fact, both interfaces are implemented by the same class: `ValidationListener`.

The validation listener contains two classes which actually implement the tasks that need to performed for validation: selecting the best solution in case multi-objective optimization returns multiple solutions (implemented by `SolutionSelector`, satisfying Requirement 6) and performing the actual validation (`SolutionValidator`, Requirement 10). Furthermore, this class resets the optimization process when a single run is completed, so the next run is performed automatically (Requirement 8). The results are written at the end of the validation process, as required by Requirement 9.

Creating a new solution selection technique is a matter of creating a subclass of `Solution-Selector` and implementing the new technique. All that needs to be done to change the solution selector used, is providing the validation listener with the newly added class, thus satisfying Requirement 7.

The state listener of `ValidationListener` gets notified just before optimization starts and right after optimization stops. At the start of the optimization process, we initialize `Solution-Validator`, opening the file in which the results will be written. When optimization stops, this file is closed.

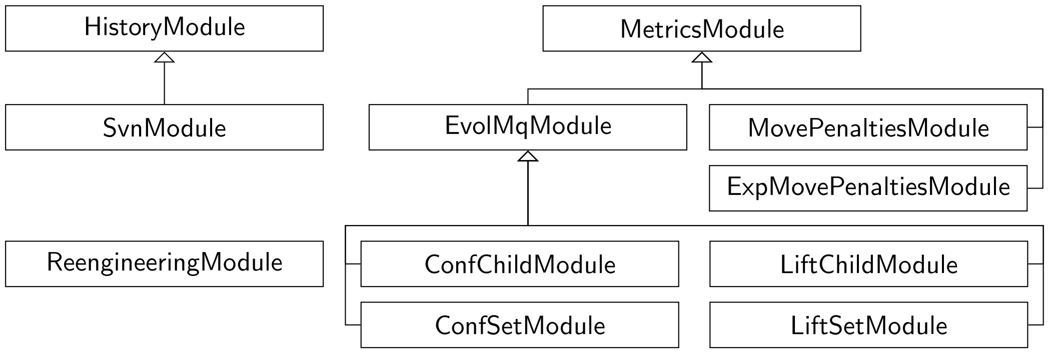

Figure 7.2: *An overview of all* `Modules` *as used for dependency injection by* GOOGLE GUICE.

The iteration listener actually contains all the code for validation. If the current run is completed (that is, only after the last iteration of the run validation is performed), we first get the (best) solution from `SolutionSelector`. This solution is then passed to `SolutionValidator` which performs the validation. For more details on the validation process, see Chapter 8.

Additional metrics Finally, it should be noted that two additional metrics are defined in Chapter 6. These two metrics are not intended to measure the quality of the code structure, but rather for penalizing each move of an entity to another module. These metrics are just like the evolutionary coupling metrics in the sense that they both implement the interface `IMetric`, and can thus be added to the optimization process just like the other metrics.

7.3 Google Guice

Dependency injection [20] is a programming technique where dependencies of a class are passed to that class, instead of the class creating instances of its dependencies itself. More specifically, when a class C_1 depends on another class C_2 to do its job, an instance of C_2 is given to (injected into) C_1, instead of having C_1 request an instance of C_2. Often, dependencies are injected into the constructor upon creation of an object. This way, it is not necessary for a class to depend on any specific implementation of its dependency. Any class implementing the functionality C_1 depends on, can be injected.

OPT4J uses the dependency injection framework GOOGLE GUICE [24] for resolving object dependencies. Therefore, the library is designed such that GOOGLE GUICE is also used for the dependencies internal in the library. That is, all classes are implemented such that GOOGLE GUICE can resolve the dependencies automatically and inject them into the constructor. Because of this, there are two requirements on the code.

1. The constructor of each class GOOGLE GUICE should use, needs to be annotated with `@Inject`.
2. `Module` classes need to be added, which tell GOOGLE GUICE how to resolve dependencies by binding an abstract type to a concrete type.

Note that GOOGLE GUICE requires all `Module`s to inherit from the class `AbstractModule`. OPT4J extends this class with a class OPT4JMODULE. Figure 7.2 shows all defined `Module`s for the restructuring tool (part of the library), which are used to bind types. The `Module` classes

Module	**Abstract type**	**Concrete type**
ReengineeringModule	`Creator`	`DataLoader`
	`Decoder`	`TreeDecoder`
	`Evaluator`	`TreeEvaluator`
	`OptimizerStateListener`	`ValidationListener`
	`OptimizerIterationListener`	`ValidationListener`
HistoryModule	*See discussion Section 7.3*	
SvnModule	`IHistProv`	`SvnProv`
MetricsModule	*See discussion Section 7.3*	
EvolMqModule	`Map<Objective,IMetric>`	`EvolMq`
	`IMetric`	`EvolMq`
ConfChildModule	`ModRel`	`ChildRel`
	`EntRel`	`ConfRel`
ConfSetModule	`ModRel`	`SetRel`
	`EntRel`	`ConfRel`
LiftChildModule	`ModRel`	`ChildRel`
	`EntRel`	`LiftRel`
LiftSetModule	`ModRel`	`SetRel`
	`EntRel`	`LiftRel`
MovePenaltiesModule	`Map<Objective,IMetric>`	`MovePenalties`
ExpMovePenaltiesModule	`Map<Objective,IMetric>`	`ExpMovePenalties`

Table 7.1: *The modules, along with the type bindings they define.*

`HistoryModule`, `MetricsModule` and `ReengineeringModule` in Figure 7.2 inherit from OPT4JMODULE, which is not shown for readability reasons.

Table 7.1 shows a list of all the `Module`s and the type bindings they define. In general, each implementation class of an interface also has a corresponding `Module` class. This `Module` then binds the interface to the implementation class. Furthermore, the `Module` can contain a number of parameters which are required by the class. For example, the metrics require a minimum support value.

A special note needs to be made for a few `Module`s, though. First of all, Table 7.1 suggests that metric `Module`s bind a `Map` to a subtype of `IMetric`. This, however, is misleading, because in fact it *adds* a binding to the map. That is, each module loaded adds a binding, which results in a map containing multiple metrics. This then makes it possible to load multiple metrics for the purpose of multi-objective optimization.

Secondly, `HistoryModule` and `MetricsModule` are abstract classes, serving as a category to group the other `Module`s. Additionally, they define methods `bindHistory()` and `bind-Metric()` respectively, easing the binding for sub-`Module`s (that is, sub-`Module`s only need to supply the concrete class, and for metrics also the name and optimization direction).

Finally, `EvolMqModule` defines two bindings for `EvolMq`. The first binding adds the metric to the map of all metrics which should be used for optimization. The second binding binds `EvolMq` directly to `IMetric`. This binding is for the validation. It binds a different, independent instance of the metric which is used for the validation process. See Chapter 8 for details on the validation.

The `Module`s present in the library are automatically discovered by OPT4J, as long as they inherit from `Opt4JModule`. `Module`s can then be loaded from the GUI, effectively binding interfaces to classes. For each `Module`, the parameter values can be modified in the GUI as well.

There is one `Module` which should always be loaded, `ReengineeringModule`. This `Module` binds all types required by OPT4J. These types are discussed in Section 7.2. This `Module` also attaches the validation listener, enabling the validation.

The GOOGLE GUICE injector is constructed by supplying a list of `Module`s. Creating an object is then done by requesting an instance of a class from the injector. The appropriate constructor (annotated with `@Inject`) is then called, with all dependencies automatically resolved. Resolving the dependencies is based on the bindings in the loaded `Module`s. Adding a dependency is then only a matter of accepting the dependency in the constructor, and the rest is handled automatically.

OPT4J creates the GOOGLE GUICE injector, loading those `Module`s the user selected in the GUI. Furthermore, only the OPT4J framework ever uses the injector to create objects. Our library only lets GOOGLE GUICE inject dependencies in the constructors whenever an instance of the class is required by OPT4J or as a dependency to another class.

8. Experimental results

Several experiments are performed using the optimization technique discussed in Chapter 6. These experiments are performed on the same case study as used in Chapter 5, Spring Framework. However, this time we are not investigating a large restructuring effort, but instead we are restructuring the source code using the techniques discussed in this thesis.

The case study is first briefly discussed in Section 8.1. Then, the methodology of the experiments is presented in Section 8.2. This includes a list of the metrics used and an overview of the experiments performed. Section 8.3 then presents the results of the experiments and a conclusion is finally given in Section 8.4.

8.1 Case

Again, the `spring-webflow` subproject of the Spring Framework is selected as a case study for the experiments. However, this time we are not interested in investigating a restructuring effort from the past. Instead, we now take the most recent version of the source code of this project and attempt to restructure the code using the optimization technique from Chapter 6.

The most recent commit to the project at the time of the experiments is commit 14496. We need a commit history which does not contain any major restructurings, because otherwise commits might include changes to entities which no longer exist. As this is undesirable, we also note the oldest commit which can safely be used, which is commit 5636. In the range 5636–14496, there are almost 600 usable commits.

8.2 Methodology

In Chapter 5, we validated multiple different metrics, each with a set of possible parameter values. From the results of this validation study, we selected four metrics which are used in the experiments in the current chapter. These metrics are all based on MQ with the set-based technique for module-level relations (see Chapter 3 for the metric definitions). Furthermore, three metrics use the confidence-based technique for entity-level relations, while the fourth metric uses the lift-based technique. For the confidence-based metrics, the three different techniques for combining confidence values are used (maximum, average and minimum). An overview of the metrics used for the experiments in this chapter, including the parameter values, is given in Table 8.1.

Furthermore, Chapter 6 defined two additional metrics, the normal CP (Definition 15) and the exponential CP (Definition 16). These two metrics are intended to be combined with an MQ metric for multi-objective optimization. Therefore, for each MQ metric, we perform three experiments:

1. Single-objective, using only the MQ metric as an objective.

2. Multi-objective, optimizing both MQ and the normal CP metrics.

3. Multi-objective, optimizing both MQ and the exponential CP metrics.

Entity-level	**Confidence combiner**	**Support**	**Confidence**
Confidence	Maximum	0.005	0.5
	Average		
	Minimum		
Entity-level		**Support**	**p-value**
Lift		0.005	0.01

Table 8.1: *An overview of the four metrics used in the experiments. All metrics use the set-based module-level technique. The choice of these metrics is based on the results of the metric validation study in Chapter 5.*

Because our optimization technique, the genetic algorithm, is non-deterministic, we repeat all experiments multiple times. From these runs, we can calculate the average and a 95% confidence interval. This allows us to make claims on the typical results, instead of just on a single value. All experiments are therefore performed by executing 10 runs of the algorithm.

For validating the success of the algorithm, we can opt for consulting an expert to check whether the returned structure is indeed better than the original structure. However, using an expert to validate results is not very precise, and the use of a more formal validation technique is preferable [44]. Experts can only give their "opinion" about how well the two structures match their mental model. Furthermore, experts might be biased (for example, developers on the source code are expected to prefer the original structure, as that is what they are familiar with).

Note that statistical techniques can only be used for conclusions on values measured by our metrics, and cannot ensure improvement of the modular quality based on aspects not measured by the metrics. A validation study based on expert opinions, however, needs to be executed with great care. That is, to make the results more objective (instead of a subjective opinion), multiple experts are needed. Furthermore, these experts should be free of any bias, and thus cannot be familiar with the original code structure. The experts need time to investigate both structures (original and improved) to form their opinion. As all of this takes time (both to execute the experiment, and time for the experts to form their opinions), it is a good idea to first validate the results using statistical techniques. If there are theoretical problems with the approach, there is no need to spend time on finding practical problems as well. Therefore, we now use statistical techniques to validate the results, and a validation study based on expert opinion is left for future research.

In order to measure the performance of the genetic algorithm, the commit history is split into two parts: a *training set* and a *test set*. The training set is used by the MQ metric which is optimized. The test set contains other commits than the training set and is used at the end of the optimization process to calculate the metric value on the improved structure. The commits in the test set are thus not considered during the optimization process, but are only used for validation.

In other words, the commit history is split into a training set and a test set. The training set is used for metric calculation during the optimization process. Therefore, the metric value based on this set is expected to increase significantly. However, this increase in metric value might be the result of a structure based on relations supported by the training set, while these relations are not supported by the test set. The test set is thus used to validate that the metric value even increases when a different set of commits is considered. For the optimization process to be successful, we thus also expect a significant increase in metric value based on the test set.

To minimize the risk that the split into training and test set is not fair, in the sense that the

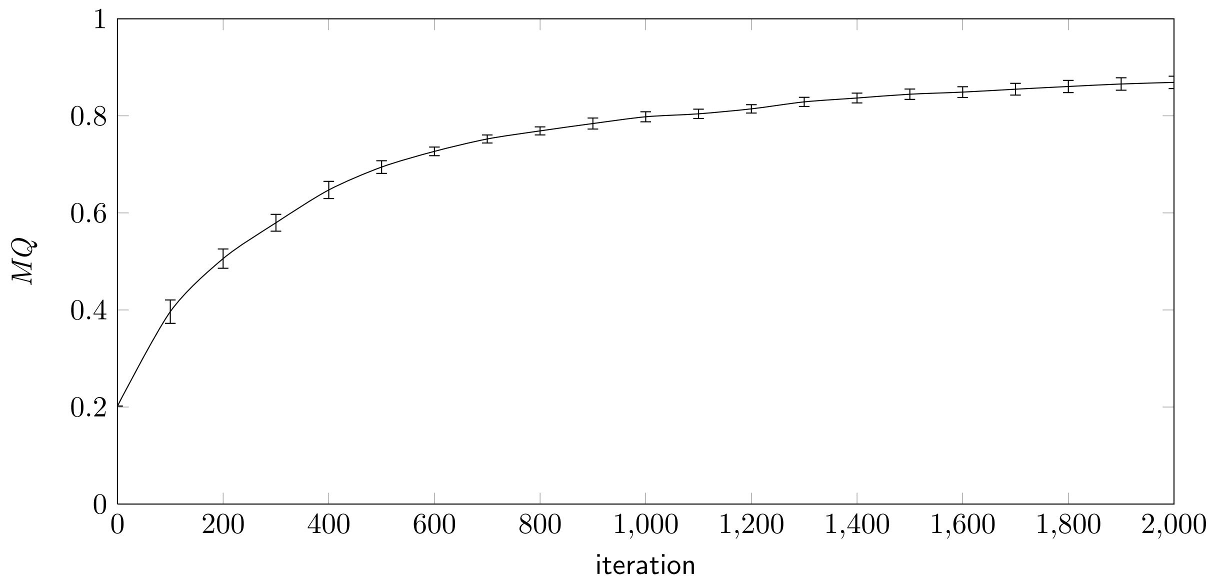

Figure 8.1: *The evolution of the MQ value during the execution of the genetic algorithm. A total of 2000 iterations with the corresponding maximum metric value at that iteration are shown, together with 95% confidence intervals after every 50 iterations. We see that the metric value is already fairly stable after 1000 iteration.*

metric values are influenced by the properties of the two sets, we choose to make the training set and the test set the same size. This way, we eliminate the risk that the metrics are sensitive to the size of the history.

Furthermore, the commit history is not split on a given time, with all commits before that time in the training set and the commits after that time in the test set. The problem with splitting this way, is that older commits might lead to the discovery of relations which were later removed, or miss the relations which were later added. Therefore, both the training set and the test set span the complete, analyzed history. That is, if all commits are numbered sequentially, the even numbered commits are used in the training set and the odd numbered commits in the test set. Note that this numbering does not correspond to the revision numbers, because some revisions might be missing.

By using the above described technique of splitting the commit history in a training set and a test set, we can investigate whether an increase in MQ value really corresponds to an improvement in code structure quality. That is, while it is obvious the optimization process will find structures with a higher MQ value on the training set, we can validate whether this increase also occurs when the metric is calculated on the independent test set.

Additionally, we investigate multi-objective optimization in a bit more detail. For the two CP metrics, we plot the found solutions of all runs in order to compare if approximately the same set of Pareto-optimal solutions is found. Furthermore, we check which of those solutions are selected to be returned by the algorithm.

One more parameter needs to be selected for all experiments, and this is the number of iterations the genetic algorithm will perform during each run. For this, we choose to use 1000 iterations. The reason for this is shown in Figure 8.1. This figure shows how the MQ value increases during the execution of the genetic algorithm. Initially, a relatively large increase per iteration is detected, but after about 1000 iterations, the MQ value hardly increases anymore.

Furthermore, the 95% confidence intervals shown after every 50 iterations also show that the metric value varies more during the earlier iterations, and becomes more stable at about 1000 iterations. That is, the confidence intervals becomes smaller. Therefore, we conclude that running the genetic algorithm for more than 1000 iterations neither helps much to get higher MQ values, nor does it give more stable results over multiple runs. Therefore, we decide to use 1000 iterations for all experiments.

8.3 Results

This section presents the results of the experiments. First, the results for single-objective optimization are discussed in Section 8.3.1. Next, the results for multi-objective optimization (for both CP metrics) are discussed in Section 8.3.2. Furthermore, we investigate the shape of the Pareto-front in Section 8.3.3. Finally, the performance in terms of execution time of the techniques is compared in Section 8.3.4.

The results discussed in this section all use a 95% confidence interval. Furthermore, the statistical significance of observations of the results is tested using a significance level of 5% (that is, a p-value of 0.05).

8.3.1 Single-objective optimization

The results of the experiments with single-objective optimization are shown in Figure 8.2. Here, we can see a large increase in MQ for all four metrics, for the training set. An increase is of course expected, since the best solution found is returned, which is always at least as good as the original structure.

However, an increase from 0.2 to 0.8 is detected, which means the objective function could be significantly improved. As the maximal value the MQ can take is 1, a solution quite close to the optimum is found. Note that even the best structure might not have an MQ value of 1, so it is indeed a good result.

When we look at the results of the test set, however, there is a much smaller increase in MQ. It is indeed to be expected that for the test set the improvement is smaller, because the commits in the test set are not considered during optimization. That means that any relations apparent in the test set, but not in the training set, are not detected during optimization, and vice versa.

Of course, this independence of the test set with respect to the training set is exactly what we want. Now, we can see that the improved structure indeed also shows an improvement when other commits are considered. Thus, entities which co-evolve in the training set must also have co-evolved in the test set, causing similar relations to be detected. The fact that the relations detected from the training set and the test set are similar, explains why the metric value was also increased on the test set. This already suggests that the presented automated reengineering technique is indeed capable of improving the code structure.

Comparing the four MQ metrics, we notice that, for the training set, a difference in MQ value before restructuring corresponds with a similar difference after restructuring. That is, the MQ value for the confidence-based metric with minimum confidence combiner is smaller than the MQ value for the other metrics, but this is true both before and after restructuring.

On the test set, however, this is not always true. For example, the MQ value before restructuring is lower when using the average as confidence combination technique when compared to using

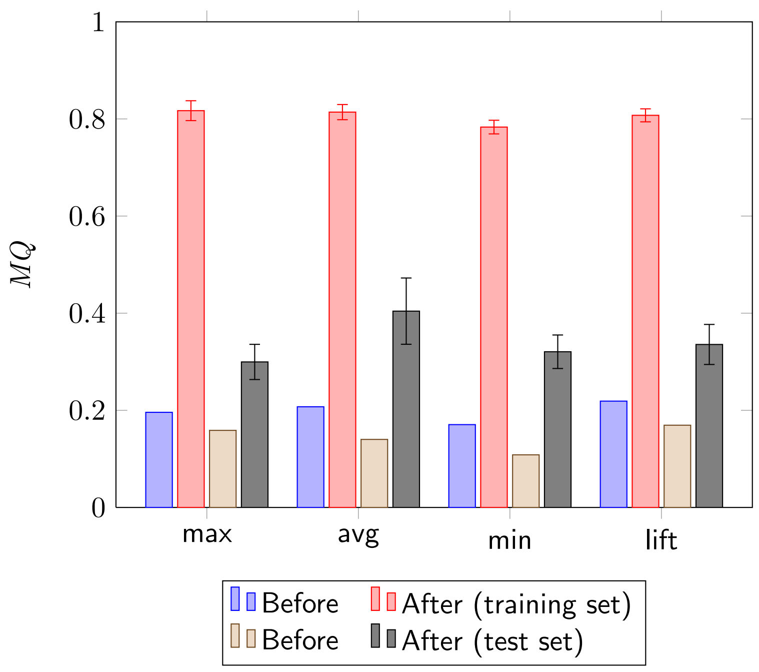

Figure 8.2: *Results of the experiments for single-objective optimization, for 10 runs. 95% confidence intervals are shown for the metric values of the code structure after restructuring. Both the results for the training set and the test set are shown. On the training set, there is a large increase in MQ shown for all metrics, but even on the test set a statistically significant increase in MQ is detected for all metrics.*

the maximum. But after restructuring, the MQ value is higher for average. A similar observation is made between using the maximum and the minimum for the confidence-based metrics.

This suggests that using the maximum to combine two confidence values is not the best option, as the average and the minimum appear to perform better. However, the difference in MQ value after restructuring is not statistically significant and the difference in MQ value before restructuring is based on only a single measurement. Therefore, we cannot make a strong claim on this difference, but it does appear to show that the average and minimum are better techniques for combining two confidence values, than taking the maximum.

8.3.2 Multi-objective optimization

Figure 8.3 shows the results for the experiments with multi-objective optimization for the four MQ metrics, combined with the normal CP. We can see similar results as with the single-objective optimization. That is, a large increase in MQ is found for the training set, and a smaller but still statistically significant increase on the test set.

One important difference we should note, is that the measured MQ values after the restructuring are significantly lower for the multi-objective optimization than for the single-objective optimization. That is, on the training set, the MQ value of the best solution found is only about 0.35, while for single-objective optimization values of 0.8 were reached. It should be noted, however, that the shown MQ values are not the maximal MQ values found, but the MQ values of the selected, best solution (according to the selection technique as described in Chapter 6). The individual solutions (and thus also the one with highest MQ) are discussed in Section 8.3.3.

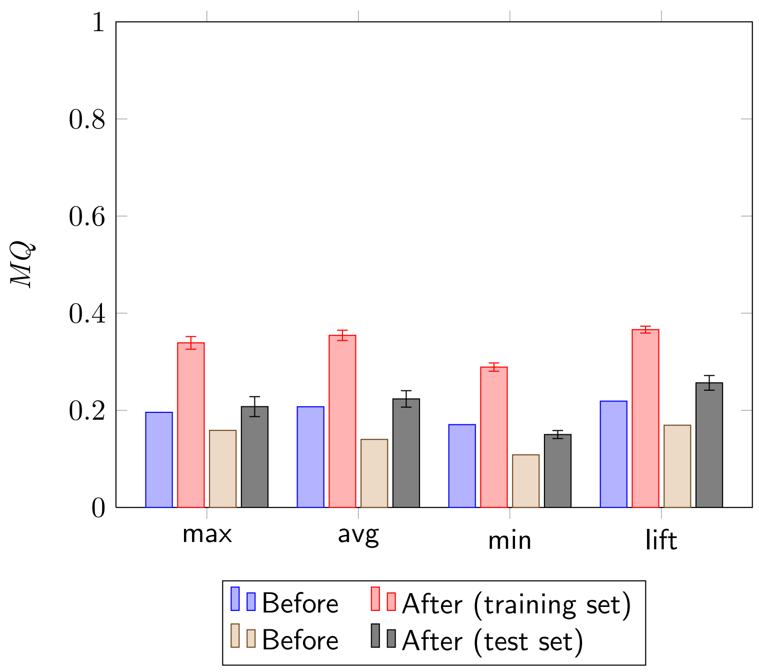

Figure 8.3: *Results of the experiments for multi-objective optimization using the normal CP. 95% confidence intervals are shown for the metric values of the code structure after restructuring. For both the training set and the test set, an increase in MQ is detected for all metrics. The increase for the test set, however, is smaller than for the training set.*

When we look at the results on the test set, we again note that the measured MQ values are smaller than for single-objective optimization. However, in this case, the difference is not as large as on the training set. Where single-objective optimization showed MQ values around 0.3, the multi-objective approach shows values around 0.2.

In other words, using multi-objective optimization (as opposed to single-objective optimization) results in MQ values which are significantly smaller on the training set, but only slightly smaller on the test set. This increased stability between training and test set suggests that the restructurings performed by the multi-objective genetic algorithm are more limited to those restructurings which truly contribute to a better code quality. That is, single-objective optimization made more changes, but many of them do not contribute much to an improved structure.

Comparing the four MQ metrics, we again make the same observation as for single-objective optimization. On the training set, a larger MQ before restructuring corresponds with a larger MQ after restructuring. For the test set, this is however not true when comparing the maximum and average as techniques for combining confidence values. This time, minimum does behave the same on the test set as on the training set.

This again suggests that taking the average is better than taking the maximum of two confidence values. However, again the differences after restructuring are not statistically significant. Therefore, we cannot make any strong claims, and can only suggest that the average might perform slightly better than the maximum.

Figure 8.4 shows the results for the experiments using the exponential CP as the second objective. These results look remarkably similar to the results for multi-objective optimization with the normal CP. Indeed, there are no statistically significant differences between the results.

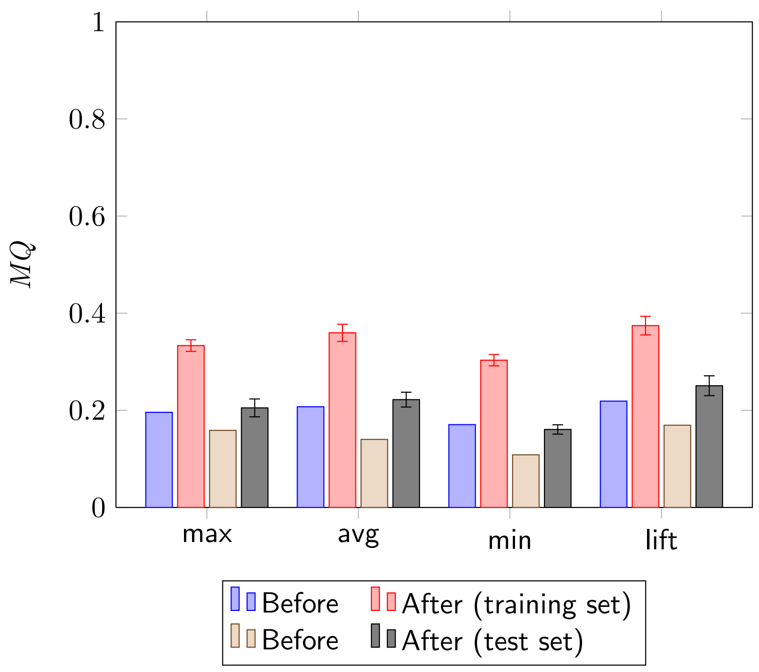

Figure 8.4: *Results of the experiments for multi-objective optimization using the exponential CP. For the code structure after restructuring, 95% confidence intervals are shown. For both training and test set, an increase in MQ is detected for all metrics. The increase for the test set, however, is smaller than for the training set.*

Thus, there does not seem to be any real difference between the two CP metrics, when comparing the quality of the returned code structure.

The change penalty values for both CP metrics are shown in Figure 8.5. Here, we immediately notice the difference for the confidence-based MQ with the minimum confidence combiner. For the normal CP, the measured values are clearly the smallest of all metrics, while for the exponential CP the largest change is measured.

It is, however, an interesting difference which we observe. The results for CP and for MQ come from the same set of 10 runs. This is interesting, because although the results for MQ are extremely similar for the two CP metrics, the results for CP are observably different.

However, one should not be fooled by the observed differences in the results for CP. The confidence intervals in these results are also fairly large, and indeed, no statistically significant difference between the CP values is detected. Therefore, we cannot make any strong claims on the differences in CP between the metrics. As a result, we must conclude that no clear difference is detected between the normal CP and the exponential CP.

8.3.3 Pareto-front

A plot of all the Pareto-optimal solutions found (all 10 runs aggregated) by the multi-objective optimization using the normal CP is shown in Figure 8.6. Note that each run only returns those solutions which are Pareto-optimal with respect to all solutions found during that run. However, it is possible that a run of the optimization algorithm finds solutions which dominate the Pareto-optimal solutions of another run.

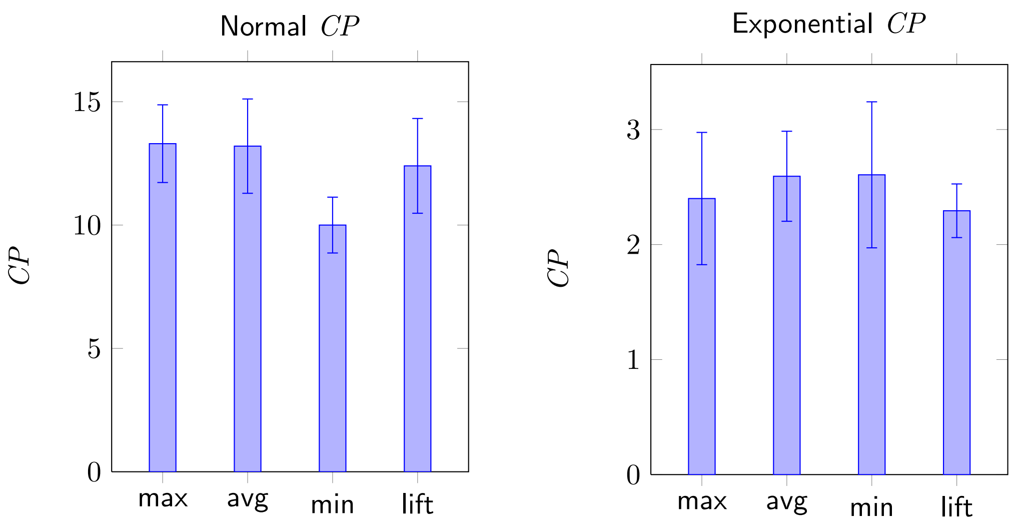

Figure 8.5: *The CP values as measured during the experiments for multi-objective optimization, including 95% confidence intervals. One immediately obvious difference is that the confidence-based technique with the minimum confidence combiner shows the least number of changes for the normal CP, but the largest value for the exponential CP.*

Figure 8.6 thus distinguishes between solutions which are Pareto-optimal with respect to all solutions found by all 10 runs together, and those solutions which are Pareto-optimal with respect to the solutions found in the same run, but not for all solutions found. Furthermore, the solutions selected as the best solution in each run is also marked.

The first thing we note is that the Pareto-front indeed shows a shape similar to the one discussed in Chapter 6. That is, the first few changes result in a relatively large increase in MQ, while later changes result in a smaller increase in MQ. Also, the shape of the Pareto-front for the individual runs are fairly consistent, in the sense that they approach the same values.

However, a large number of solutions turn out to be no longer Pareto-optimal when the solutions of all 10 runs are aggregated. This is to be expected, because only a limited number of Pareto-optimal solutions can exist. Since the normal CP is always an integer value, and the CP value of the found solutions is always smaller than 60, there can be no more than 60 Pareto-optimal solutions. Since the 10 runs combined have more than 200 solutions, it is obvious that the majority of those solutions is no longer Pareto-optimal.

The next thing we notice is that the selected solutions are all located near the same part of the Pareto-front. This again shows some consistency between the runs, with a similar result returned each time. Also, we note that these solutions are located there where the increase in MQ becomes relatively smaller compared to the increase in CP. That is, the selected solutions are exactly where we expect them to be.

Finally, we note that the solutions are closer together at a low CP and MQ, while they are further apart for higher CP and MQ. This is explained by the fact that each run finds the same minimum MQ and CP, namely those corresponding to the original structure of the code. As more changes are made to the code structure, the runs start deviating more. This is especially true for those solutions with a higher CP than the selected solution.

Comparing the results for the four MQ metrics, there are not many differences. All have a similarly shaped Pareto-front and even the differences in maximal and minimal MQ and CP values

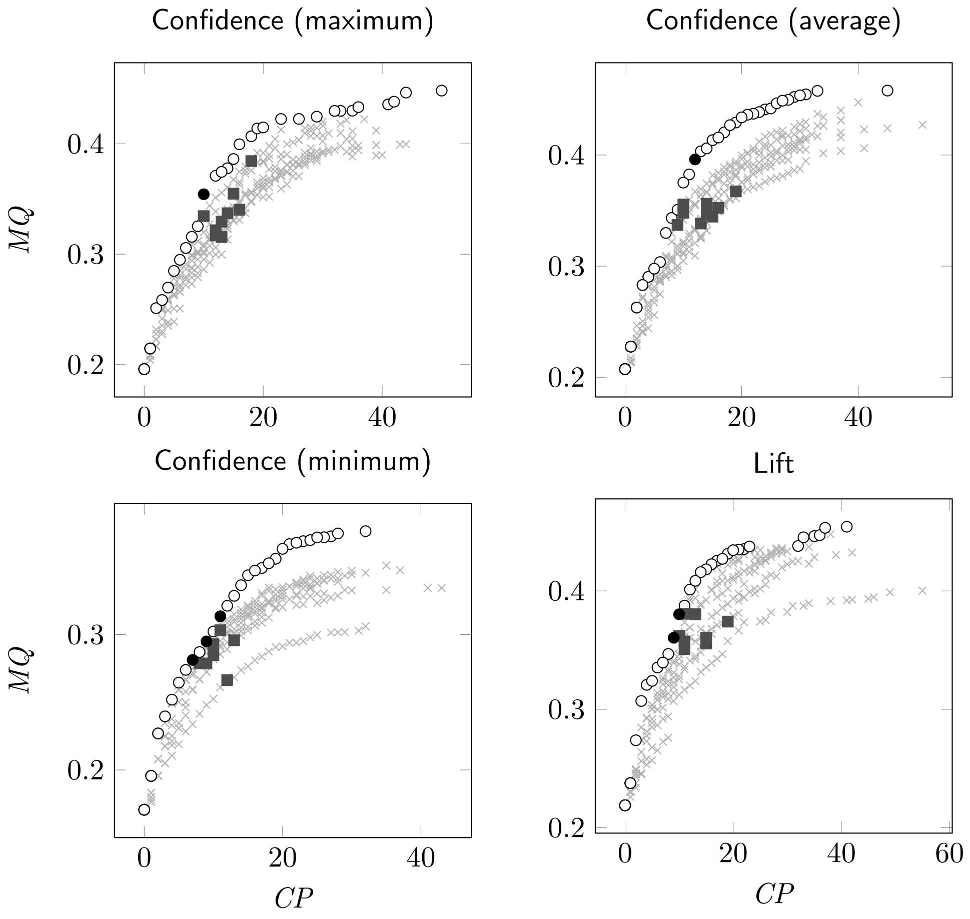

Figure 8.6: *Plots of the solutions found during all 10 runs using normal CP. Pareto-optimal solutions which were selected are denoted with •, and those not selected with ○. Solutions which are dominated by solutions from another run are denoted with ■ when they were selected and with × when they were not selected. The Pareto-front has a similar shape for all runs, with the selected solution always at approximately the same part of the Pareto-front. However, the selected solutions are frequently dominated by solutions from another run.*

found is similar, with only the same differences as detected in the average values for MQ.

Figure 8.7 shows a plot of all Pareto-optimal solutions of the 10 runs aggregated, for the exponential CP. The results here are similar to the results for the normal CP, with a few notable differences.

The first difference is that the Pareto-front shows a much steeper increase in MQ at the start, and a smaller increase at the end, for the exponential CP. This is most likely because of the exponential nature of this CP. That is, the penalty of a single move is exponential with respect to the distance over which it is moved, while this penalty is linear for the normal CP. Therefore, for small CP (and thus few changes), this exponential nature causes a more rapid increase in MQ, while for large CP (and thus many changes), the summation of all the penalties suppresses the effect of the exponentiations, causing a slower increase in MQ.

The second notable difference is the larger number of solutions. The exponential CP returns a total of more than 400 solutions, which is twice as many as the normal CP. This difference

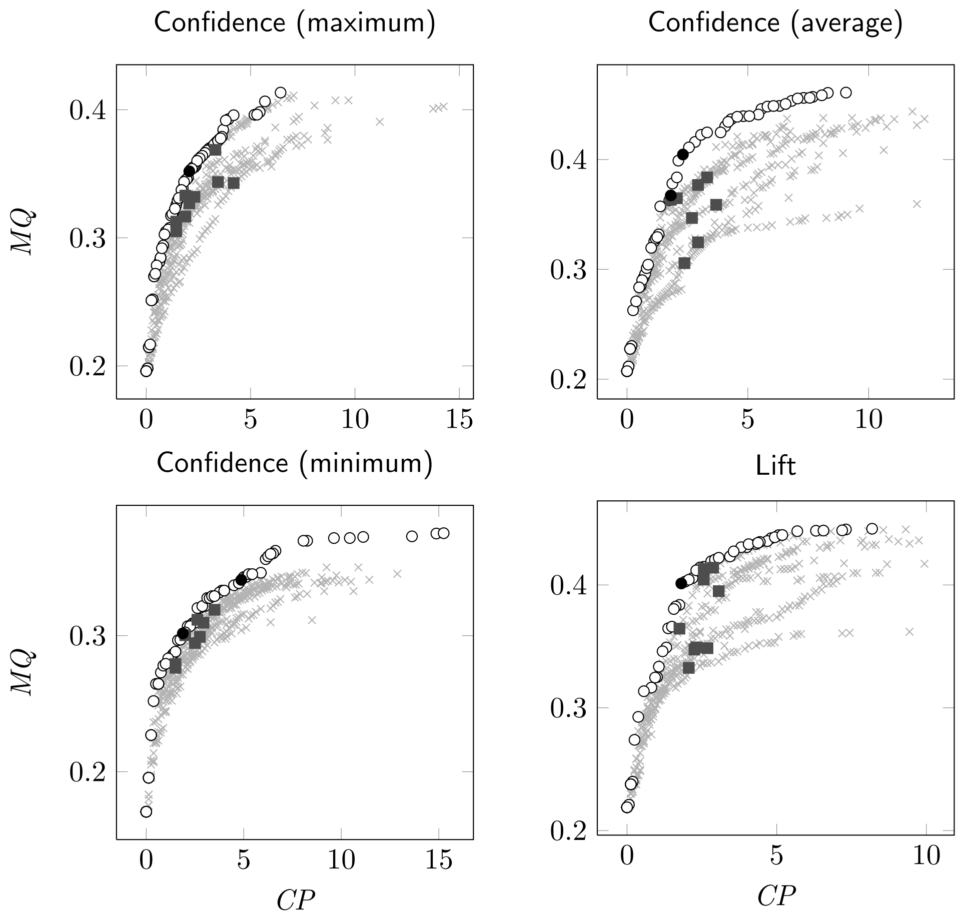

Figure 8.7: *Plots of the solutions found during all 10 runs using exponential CP. Pareto-optimal solutions which were selected are denoted with ●, and those not selected with ○. Solutions which are dominated by solutions from another run are denoted with ■ when they were selected and with × when they were not selected. The Pareto-front has a similar shape for all runs, with the selected solution always at approximately the same part of the Pareto-front. However, the selected solutions are frequently dominated by solutions from another run.*

comes from the fact that the exponential CP can theoretically grow in arbitrarily small steps. In practice, for code with a module depth of 3 (which is true for the SPRING FRAMEWORK), the exponential CP can grow in steps of $1/2^3 = 1/8$. So for the measured range for the CP of 15, this means $15 \cdot 8 = 120$ possible CP values. This is twice that for normal CP, so having twice as many solutions found is to be expected.

While there are differences between the two CP metrics, the differences are not of such a nature that we can say one is better than the other. We must thus again conclude that the two metrics are similar, and perform equally well in measuring the changes.

8.3.4 Execution time

The execution times as measured during the experiments are shown in Figure 8.8. The execution times are measured per run (1000 iterations), and the results of the 10 runs are then aggregated

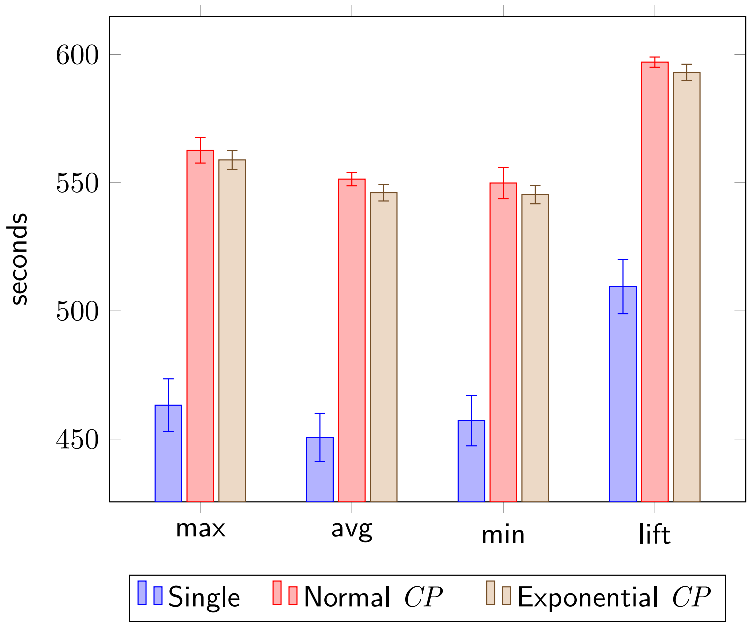

Figure 8.8: *Average execution times per run (1000 iterations) of the genetic algorithm. Single-objective optimization is the fastest, while multi-objective shows similar performance for both CP metrics. The confidence-based metrics show better performance than the lift-based metric.*

to get an average and 95% confidence interval.

The first thing we note is that single-objective optimization is the fastest for each MQ metric. This is to be expected, because the multi-objective optimization requires the evaluation of two metrics each time the fitness of a solution needs to be calculated. This causes a 20% increase in execution time for multi-objective optimization, as compared to using only a single objective.

Furthermore, for multi-objective optimization, no significant difference between the performance of the normal CP and the exponential CP is detected. It could be expected that the exponential CP would be slower than the normal CP, because the normal CP only requires integer addition to calculate, while the exponential CP requires the exponentiation of floating point number. However, the results show that exponential CP is even slightly faster, but this difference is not statistically significant.

Finally, the confidence-based metrics are faster than the lift-based metric, both for single-objective and multi-objective optimization. This is to be expected, because calculating the confidence is computationally inexpensive. For the lift-based metric, a statistical test needs to be performed, which is a more expensive operation. The execution time for the lift-based metric is therefore approximately 10% longer.

No clear difference can be detected between the execution times of the three confidence-based metrics. While the average execution time is lowest when the minimum is used, and highest when the maximum is used to combine two confidence values, this result is not statistically significant.

8.4 Conclusion

The experiments performed in this chapter compared four MQ metrics and three optimization techniques for the problem of automated restructuring. The results for all experiments indicate that a significant increase in MQ value can be achieved using these techniques. Therefore, we can conclude that the proposed approach is indeed capable of improving the structure of source code.

While single-objective optimization results in a larger increase in MQ value, this comes at a cost of large changes to the code structure. As it is desirable to keep the code structure stable, multi-objective optimization is a good approach to minimize the number of changes required to achieve an increase in MQ.

Besides limiting the number of changes to the code structure, multi-objective optimization also showed more stable results on the test set. That is, while the MQ value after restructuring on the training set was significantly smaller for multi-objective optimization than for a single objective, the MQ value on the test set was more similar to the results for single-objective optimization. This stability for multiple objectives indicates that single-objective optimization causes too many changes to the code structure, which then only result in a further degradation of MQ on another set of commits.

Nonetheless, even for single-objective optimization, the results are promising. An increase in MQ value is detected, even on the test set. This indicates that the proposed technique does not restructure the code only to match what is learned from the given set of commits. Instead, the technique restructures the code by grouping entities together which also co-evolve in the commits not considered during the optimization process. In other words, it groups entities together which are in fact related to each other.

Multi-objective optimization does come with a downside, however. The execution time required for 1000 iterations is longer for multi-objective than for single-objective optimization. Using multiple objectives is 20% slower than using only MQ as the single objective. But the advantages of multi-objective optimization are worth this increase in execution time, because it prevents unnecessary changes being made to the code structure (that is, changes which do not improve the MQ value).

Comparing the two multi-objective approaches (that is, the two metrics normal CP and exponential CP), we see that there is no real difference. Both find solutions of the same quality, and both are equally capable of finding a good approximation to the Pareto-front. Furthermore, both result in approximately the same execution time for 1000 iterations. Therefore, on the basis of the performed experiments, we cannot detect any significant difference between the two metrics.

When we compare the four MQ metrics, we see that all of these metrics perform well. That is, they are all capable of improving the code structure. There are, however, some differences between the metrics.

First of all, for the three confidence-based metrics, the one taking the maximum to combine two confidence values shows a slightly worse performance than the other two. This is based on the fact that on the training set, a higher MQ value before restructuring corresponds to a higher MQ value after restructuring. However, on the test set, the metric using the maximum shows a higher MQ value before, but a lower MQ value after restructuring. This thus indicates that too many relations are detected due to a high confidence in one direction, while they turn out not to be related when the test set is considered.

Secondly, the lift-based technique does not appear to show much difference to the confidence-based techniques. One difference is that it appears to be slightly worse than the confidence-based techniques for single-objective optimization, while it appears to be slightly better for multi-objective optimization. Therefore, as multi-objective optimization is favorable over single-objective, the use of the lift-based technique might be a good choice. However, it should be noted that neither difference is truly significant.

A clearer difference between the lift-based and confidence-based techniques, is when we compare the execution times. The lift-based metric shows a 10% slower execution of the 1000 iterations than the confidence-based metrics (which are all equally fast). Therefore, the use of confidence-based metrics seems to be preferred over the use of the lift-based metric.

However, it should be noted that the confidence-based metrics have a theoretical problem (see Chapter 3), so according to the theory, the lift-based metric is preferred. The results presented in this chapter, however, do not indicate that this is a problem in practice.

8.4.1 Threats to validity

The experiments in this chapter were designed and performed in such a way, that the results are meaningful and valid. However, some issues can threaten the validity of the results, and thus the validity of the conclusion.

The main issue is that the experiments are all performed on only one case study. While the results are positive on this case study, things might not look so nice on another case study. Therefore, it is important for future research to focus on validating the obtained results on multiple case studies. Besides validating the results, such research might also be capable to show clearer differences between the proposed metrics.

Another issue is the non-determinism of the algorithm. As the genetic algorithm makes random decisions, the results are influenced a great deal by chance. To prevent this issue from becoming a problem, multiple runs of the algorithm were performed. The results of these runs were then aggregated, and 95% confidence intervals calculated. However, even the use of confidence intervals does not provide us with absolute accuracy.

Furthermore, the commit history was split into a training set and a test set, in order to prevent the problem that an increase in MQ is a result of "over-fitting" the history by our improved code structure. As the test set is not used during the optimization process, it shows how the MQ value changes on another set of commits. Here, we noticed an increase in MQ even on the test set. For single-objective optimization, however, this increase was much smaller than on the training set, thus indicating that over-fitting indeed occurred there. For multi-objective optimization, this was not so much a problem.

9. Conclusions

As software systems evolve, the quality of the modularization of the code degrades. Entities might no longer be placed in appropriate modules, making the system difficult to maintain, and the structure hard to understand. To solve these problems, it is important to improve the modularization quality by reengineering the code.

Since reengineering code manually is labor intensive work, it is interesting to find techniques for automating this process. Therefore, a platform was developed which can be used for empirical research into automated reengineering. This platform was also used to validate a proposed approach to automated reengineering.

A number of metrics based on evolutionary coupling have been defined which are capable of quantifying the the quality of a modularization of the code. These metrics use the change history of the source code to detect relations between entities. A metric value is then calculated based on the fact that related entities should be part of the same module. A hierarchical module structure is supported by the metrics, as opposed to just a flat module structure, as this is closer to modularization of real systems.

All metrics were validated on a case study of a successful reengineering effort. That is, metric values were calculated for two versions of a system: one version before and one version after a successful reengineering effort. The results indicated that some of the proposed metrics indeed measured an increase in modularization quality. For the module-level techniques of detecting relations (see Chapter 3), the set-based technique performed best, while the other module-level techniques appeared to have some problems. Therefore, we can conclude that for detecting relations on the module-level, it is best to use the set-based technique. On the entity-level, things were less clear. All entity-level techniques showed promising results, and all appeared to be capable of quantifying the modularization quality.

Furthermore, a technique was proposed for automated reengineering. This technique uses a genetic algorithm in order to optimize the code structure. The search for an improved structure is guided by the defined evolutionary coupling metrics. Additionally, the genetic algorithm uses multi-objective optimization in order to minimize the number of changes which are made to a code structure. That is, while the optimization process attempts to find a structure which maximizes the modularization quality, it simultaneously attempts to minimize the number of changes. This way, the code structure is not changed so much that programmers have difficulty understanding the new structure, when they are already familiar with the old structure.

The proposed optimization algorithm was validated using a series of experiments. For this, a system was considered, of which the structure was automatically improved using the proposed techniques. The results showed that the improved structure indeed showed signs of a better modularization, indicating the success of the algorithm. Especially the multi-objective approach, which simultaneously minimized the number of changes, was successful as compared to optimizing only the modularization quality metric. However, these are results for only a single case study, so further validation is required.

9.1 Future work

While the proposed techniques used for automated reengineering showed potential, it is not yet ready for use. Further research into automated reengineering is required to both validate the proposed techniques, and possibly propose more, better techniques. The platform developed for the research into automated reengineering can be used for that purpose. New metrics and optimization strategies can be added to the platform, and tools for validation are already provided.

Ideas for future research into automated reengineering have already been proposed in Chapter 3 and Chapter 6. These include possible improvements on the proposed evolutionary coupling metrics. For example, with evolutionary coupling, it is possible to measure the strength of a relation between two entities. This can then be used to make relations for which there is strong evidence more important than relations with weak evidence. This also removes the need of setting a threshold value for when the evidence is considered strong enough to decide there is a relation. For more suggestions of future work with respect to the metrics, see Section 3.5.

For the optimization, there are possibilities of using different optimization algorithms. It would then be especially interesting to compare the performance of multiple algorithms. This then makes it possible to decide whether one algorithm is better than another for this specific problem. See Section 6.5 for more ideas on future work for the optimization strategy.

The current implementation of Fisher's exact test, which is used to perform the statistical test for detecting lift-based relations, uses log-factorials to prevent integer overflows. However, this implementation accumulates rounding errors resulting in wrong p-values. This can potentially lead to wrong decisions on whether two entities are related. Therefore, it is important to investigate to what extent this is a problem in practice, or to investigate other techniques for performing the calculations without the problem of accumulating rounding errors.

Furthermore, future research can focus on performing a more thorough validation of both the metrics and the optimization technique proposed in this thesis. Validation has only been performed on a single case study, which means the results might be specific to that system. Therefore, it is important that validation is performed on other systems, such that conclusions on the general performance of the proposed technique can be given.

Additionally, the validation performed only used statistical techniques to draw conclusions from the results. Such an approach to validation, however, does not validate that the improved structure is indeed a better structure, which should be preferred. It is only a first step towards validation. Once the approach has proven to be promising using such a theoretical approach to validation, it is important to perform a more practical validation study by consulting experts.

Finally, the current implementation of the platform does not actually perform any changes to the code. Only the structure of the code is changed in a representation of the code structure stored in internal memory, without modifying the actual code. This functionality is sufficient as long as it is only used for research, such as done in this thesis. However, modifying the code is of course a necessary part of reengineering, and should thus eventually be implemented.

Bibliography

[1] Abdeen, H., Ducasse, S., Sahraoui, H., and Alloui, I. (2009). Automatic package coupling and cycle minimization. In *2009 16th Working Conference on Reverse Engineering*, pages 103–112. IEEE.

[2] Agrawal, R., Imielinski, T., and Swami, A. (1993). Mining association rules between sets of items in large databases. In *ACM SIGMOD Record*, volume 22, pages 207–216. ACM.

[3] Agrawal, R. and Srikant, R. (1994). Fast algorithms for mining association rules. In *Proceedings of the 20th International Conference on Very Large Data Bases, VLDB*, volume 1215, pages 487–499. Citeseer.

[4] Anquetil, N. and Laval, J. (2011). Legacy software restructuring: Analyzing a concrete case. In *15th European Conference on Software Maintenance and Reengineering*, pages 279–286.

[5] Bain, L. and Engelhardt, M. (1992). *Introduction to probability and mathematical statistics*. Duxbury Press Boston.

[6] Baron, M. (2007). *Probability and statistics for computer scientists*. Chapman & Hall/CRC.

[7] Burson, S., Kotik, G. B., and Markosian, L. Z. (1990). A program transformation approach to automating software re-engineering. In *Fourteenth Annual International Computer Software and Applications Conference, 1990. COMPSAC 90. Proceedings.*, pages 314–322. IEEE.

[8] Ciupke, O. (1999). Automatic detection of design problems in object-oriented reengineering. In *Technology of Object-Oriented Languages and Systems, 1999. TOOLS 30. Proceedings*, pages 18–32. IEEE.

[9] D'Ambros, M. and Lanza, M. (2006). Reverse engineering with logical coupling. In *13th Working Conference on Reverse Engineering, 2006. WCRE'06.*, pages 189–198. IEEE.

[10] Deb, K. (2005). Multi-objective optimization. *Search Methodologies*, pages 273–316.

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

[12] DeRemer, F. and Kron, H. (1976). Programming-in-the-large versus programming-in-the-small. *IEEE Transactions on Software Engineering*, (2):80–86.

[13] Doval, D., Mancoridis, S., and Mitchell, B. (1999). Automatic clustering of software systems using a genetic algorithm. In *Software Technology and Engineering Practice, 1999. STEP'99. Proceedings*, pages 73–81. IEEE.

[14] Durillo, J. J. and Nebro, A. J. (2011). jMetal: A Java framework for multi-objective optimization. *Advances in Engineering Software*, 42:760–771.

[15] Eick, S., Graves, T., Karr, A., Marron, J., and Mockus, A. (2001). Does code decay? Assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12.

[16] Ferreira, J., Fonseca, C., and Gaspar-Cunha, A. (2007). Methodology to select solutions from the pareto-optimal set: a comparative study. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 789–796. ACM.

[17] Fischer, M. and Gall, H. (2006). Evograph: A lightweight approach to evolutionary and structural analysis of large software systems. In *13th Working Conference on Reverse Engineering, 2006. WCRE'06.*, pages 179–188. IEEE.

[18] Fisher, R. (1935). *The design of experiments.* Oliver & Boyd.

[19] Fisher, R. (1954). *Statistical methods for research workers.* Oliver & Boyd.

[20] Fowler, M. (2004). Inversion of control containers and the dependency injection pattern. Available at `http://www.martinfowler.com/articles/injection.html`.

[21] Fowler, M. and Beck, K. (1999). *Refactoring: improving the design of existing code.* Addison-Wesley Professional.

[22] Garey, M. and Johnson, D. (1979). *Computers and intractability.* Freeman San Francisco, CA.

[23] Gini, C. (1921). Measurement of inequality of incomes. *The Economic Journal*, 31(121):124–126.

[24] Google Guice (2011). Available at `http://code.google.com/p/google-guice/`.

[25] Griswold, W. and Notkin, D. (1993). Automated assistance for program restructuring. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2(3):228–269.

[26] Han, J., Pei, J., Yin, Y., and Mao, R. (2004). Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data mining and knowledge discovery*, 8(1):53–87.

[27] Harman, M. and Jones, B. (2001). Search-based software engineering. *Information and Software Technology*, 43(833):839.

[28] Hassan, A. E. and Holt, R. C. (2004). Predicting change propagation in software systems. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pages 284–293. IEEE.

[29] Haupt, R., Haupt, S., and Haupt, S. (1998). *Practical genetic algorithms.* Wiley Online Library.

[30] JDMP (2011). Java Data Mining Package. Available at `http://www.jdmp.org/`.

[31] Jørgensen, N. (2001). Putting it all in the trunk: incremental software development in the FreeBSD open source project. *Information Systems Journal*, 11(4):321–336.

[32] Kagdi, H., Gethers, M., Poshyvanyk, D., and Collard, M. L. (2010). Blending Conceptual and Evolutionary Couplings to Support Change Impact Analysis in Source Code. In *17th Working Conference on Reverse Engineering (WCRE), 2010*, pages 119–128. IEEE.

[33] Kagdi, H. and Maletic, J. I. (2007). Combining single-version and evolutionary dependencies for software-change prediction. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, page 17, Washington, DC, USA. IEEE Computer Society.

[34] Kamiya, T., Kusumoto, S., and Inoue, K. (2002). CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, pages 654–670.

[35] Kirkpatrick, S., Gelatt, C., and Vecchi, M. (1983). Optimization by simulated annealing. *science*, 220(4598):671–680.

[36] Lehman, M. M. (1980). On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, 1:213–221.

[37] Lin, W., Tseng, M., and Su, J. (2002). A confidence-lift support specification for interesting associations mining. *Advances in Knowledge Discovery and Data Mining*, pages 148–158.

[38] Lukasiewycz, M., Glaß, M., and Reimann, F. (2011a). Opt4j Documentation — The optimization framework for Java. Available at `http://opt4j.sourceforge.net/documentation/2.4/book.pdf`.

[39] Lukasiewycz, M., Glaß, M., Reimann, F., and Teich, J. (2011b). Opt4J - A Modular Framework for Meta-heuristic Optimization. In *Proceedings of the Genetic and Evolutionary Computing Conference (GECCO 2011)*, Dublin, Ireland.

[40] Mancoridis, S., Mitchell, B., Chen, Y., and Gansner, E. (1999). Bunch: A clustering tool for the recovery and maintenance of software system structures. In *IEEE International Conference on Software Maintenance, 1999.(ICSM'99) Proceedings.*, pages 50–59. IEEE.

[41] Martin, R. (1994). OO design quality metrics — An analysis of dependencies. *Workshop on Pragmatic and Theoretical Directions in Object-Oriented Software Metrics.*

[42] McQuitty, L. (1957). Elementary linkage analysis for isolating orthogonal and oblique types and typal relevancies. *Educational and Psychological Measurement.*

[43] Mens, T. and Tourwé, T. (2004). A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139.

[44] Mitchell, B. (2006). Clustering software systems to identify subsystem structures.

[45] Mitchell, B. and Mancoridis, S. (2003). Modeling the search landscape of metaheuristic software clustering algorithms. In *Proceedings of the 2003 international conference on Genetic and evolutionary computation: PartII*, pages 2499–2510. Springer.

[46] O'Keeffe, M. and Cinnéide, M. Ó. (2006). Search-based software maintenance. In *Proceedings of the 10th European Conference on Software Maintenance and Reengineering*, pages 249–260. IEEE.

[47] Papadimitriou, C. and Steiglitz, K. (1982). *Combinatorial optimization: algorithms and complexity*. Prentice-Hall, Inc.

[48] Praditwong, K., Harman, M., and Yao, X. (2010). Software module clustering as a multi-objective search problem. *IEEE Transactions on Software Engineering*, pages 264–282.

[49] Schaeffer, S. (2007). Graph clustering. *Computer Science Review*, 1(1):27–64.

[50] Šíma, J. and Schaeffer, S. (2006). On the NP-completeness of some graph cluster measures. *SOFSEM 2006: Theory and Practice of Computer Science*, pages 530–537.

[51] Spring Framework (2006). Spring Web Flow Change Log. Available at `http://static.springsource.org/spring-webflow/docs/1.0.x/changelog.txt`.

[52] SQLite (2011a). Available at `http://www.sqlite.org/`.

[53] SQLite (2011b). Limits in SQLite. Available at `http://www.sqlite.org/limits.html`.

[54] Srikant, R. and Agrawal, R. (1997). Mining generalized association rules. *Future Generation Computer Systems*, 13(2):161–180.

[55] SVNKit (2011). Subversion for Java. Available at `http://svnkit.com/`.

[56] Tan, P., Kumar, V., and Srivastava, J. (2002). Selecting the right interestingness measure for association patterns. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 32–41. ACM.

[57] Theil, H. (1967). *Economics and information theory*. North-Holland Amsterdam.

[58] Vanya, A., Premraj, R., and Vliet, H. (2011). Approximating change sets at philips healthcare: A case study. In *2011 15th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 121–130. IEEE.

[59] Verbeek, A. and Kroonenberg, P. (1985). A survey of algorithms for exact distributions of test statistics in $r \times c$ contingency tables with fixed margins. *Computational Statistics & Data Analysis*, 3:159–185.

[60] Yourdon, E. and Constantine, L. (1976). *Structured design*. Yourdon, Inc.

[61] Zimmermann, T., Diehl, S., and Zeller, A. (2003). How history justifies system architecture (or not). In *Sixth International Workshop on Principles of Software Evolution, 2003. Proceedings.*, pages 73–83. IEEE.

[62] Zimmermann, T. and Weissgerber, P. (2004). Preprocessing cvs data for fine-grained analysis. In *Proceedings of the 1st International Workshop on Mining Software Repositories (MSR 2004)*, pages 2–6. Citeseer.

[63] Zimmermann, T., Weissgerber, P., Diehl, S., and Zeller, A. (2005). Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, pages 429–445.