TU WIEN Informatics

Software Complexity of a Monadic Style in Object-Oriented Programming

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Julian Kotrba, BSc
Matrikelnummer 01427123

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr. Franz Puntigam

Wien, 26. August 2020

Julian Kotrba	Franz Puntigam

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Software Complexity of a Monadic Style in Object-Oriented Programming

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Julian Kotrba, BSc
Registration Number 01427123

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr. Franz Puntigam

Vienna, 26th August, 2020

Julian Kotrba	Franz Puntigam

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Julian Kotrba, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 26. August 2020

Julian Kotrba

Danksagung

An dieser Stelle möchte ich mich bei all denjenigen bedanken, die mich während meines Studiums aber auch im speziellen während der Anfertigung der vorliegenden Arbeit gefördert und immer unterstützt haben. Spezieller Dank gilt dabei meiner Lebenspartnerin, meiner Familie und meinen engsten Freundinnen und Freunden.

Besonderer Dank gilt auch Herrn Professor Puntigam für die fachkundige Unterstützung und die lehrreichen Diskussionen, welche mich immer in die richtige Richtung gelenkt und meinen Horizont erweitert haben.

Zu guter Letzt bedanke ich mich bei meinen Eltern, welche mir nicht nur das Studium ermöglichten, sondern auch stets an mich glaubten und mich dadurch bestärkten.

Acknowledgements

At this point, I would like to thank everyone who encouraged and always supported me during my studies, but also especially during the preparation of this thesis. In particular, I wish to acknowledge the ongoing support of my life partner, my family and my closest friends.

My sincere appreciation also belongs to my supervisor Professor Puntigam for his expert support and the instructive discussions, which have always guided me in the right direction and have broadened my horizons.

Last but not least, I would like to thank my parents, who not only made it possible for me to study, but also always believed in me and thereby strengthened me.

Kurzfassung

Über die letzten Jahre hinweg konnte ein Trend in der Entwicklung von Multiparadigmen-Programmiersprachen und der Erweiterung von Programmiersprachen mit Features aus anderen Programmiersprachen oder Programmierparadigmen festgestellt werden.

Die vorliegende Arbeit beschäftigt sich mit den Auswirkungen der Verwendung eines Konzeptes aus der funktionalen Programmierung, welches im Bereich der objektorientierten Programmierung angewendet wird. Im speziellen wurde die praktische Auswirkung eines monadischen Programmierstiles in der Datenzugriffsschicht auf die Softwarekomplexität in einer bestehenden, in Java programmierten, Android Anwendungssoftware untersucht. Die Ergebnisse wurden durch Messung der Komplexitätsmetriken *Non-Comment Lines of Code*, *Cyclomatic Complexity* und *Halstead Difficulty* vor und nach dem Umschreiben von Teilen des Quellcodes ermittelt.

Um sicherzustellen, dass Komplexitätsänderungen nach dem Umschreiben des Quellcodes auf den monadischen Programmierstil zurückzuführen sind, um eine hohe Reproduzierbarkeit erreichen und mögliche Verzerrungen der Ergebnisse minimieren zu können, wurde zusätzlich im Rahmen dieser Arbeit ein umfangreiches Regelwerk für den Umschreibungsprozess definiert. Teile des Quellcodes wurden folglich unter Einhaltung dieser definierten Regeln adaptiert.

Bei der Bewertung der Gesamtergebnisse konnte eine Diskrepanz zwischen den verwendeten Metriken zur Messung der Komplexität festgestellt werden. Eine detailliertere Analyse zeigte jedoch, dass unter bestimmten Umständen eine deutliche Verringerung (Verbesserung) der Komplexität erreicht werden konnte. Liegen diese Umstände nicht vor, wurde teilweise eine deutliche Verschlechterung beziehungsweise ein Gleichbleiben der Komplexität erzielt.

Abstract

In recent years, a trend in developing multi-paradigm programming languages and the extension of programming languages with features from other programming languages or programming paradigms has been observed.

This thesis researches the impact of the usage of a concept from the functional programming paradigm in the field of object-oriented programming. In particular, the practical impact of a monadic programming style in the data access layer on the software complexity of an Android application software programmed in Java was examined. The results were determined by the measurement of the complexity metrics *Lines Of Code*, *Cyclomatic Complexity* and *Halstead Difficulty* before and after rewriting parts of the source code.

To ensure that changes in software complexity after rewriting the source code can be attributed to the monadic programming style, to achieve a high level of reproducibility and to minimize rewriting bias, an extensive set of rules for the rewriting process was also defined. The rewriting of parts of the source code was then conducted in compliance with the established rules.

The evaluation of the overall results showed a disconnect between the used software complexity metrics. However, a detailed analysis showed that under certain circumstances, a significant reduction (improvement) of the software complexity could be achieved. Otherwise, the software complexity remained unchanged or has significantly increased.

Contents

CHAPTER 1

Introduction

1.1 Motivation & Problem Statement

In 1958, the first functional programming language, called LISP, was invented by John McCarthy [Tur13]. 60 years later, the functional programming paradigm and its techniques are still getting a lot of attention in the software engineering and programming language communities, although it is considered to be more challenging to learn than the object-oriented programming paradigm [KY17]. Reasons for this hype include that a functional programming style makes a program more robust and functional programs are easier to test compared to imperative ones [Hin09]. Conversely, the use of object-oriented programming techniques helps to improve the understandability of complex systems, since the abstract or concrete design of objects is often closely related to things in the real world [SM17]. Due to the close design of the software to the real world, problems can be detected early in the design phase, hence long-term maintainability and revising work can be reduced [SM17].

Both paradigms provide their own benefits when used, but it is also possible to use multiple programming paradigms. For example, it is popular to include functional programming constructs into non-functional programming languages or to create programming languages that build upon the concepts of multiple paradigms [Nar09, PSG12]. The programming language Scala, for instance, allows programmers to use the techniques from both worlds, while Java is located more on the imperative side, but still provides support for some functional programming features. Since the introduction of Java 8 it has been possible to use lambda expressions through functional interfaces. Functional interfaces also allow one to create higher-order functions (i.e. functions which take one or more functions as parameters or return a function as a result). Another supported functional programming feature in Java is the *Optional* class. *Optionals* are used to express the absence of values and have the advantage, that safer code can be created [TMM18]. The underlying theory is based on *Monads*, which have their origins in cate-

gory theory. Monads are type constructors that abstract sequential computations. In the paper [Nag19], Gergely Nagy found an advantage in using a monadic structure for error handling. In particular, he showed that using a monadic way to handle errors has a positive impact on the complexity of the source code [Nag19]. Because of a tight relationship between software maintainability and complexity, writing low-complexity software makes the codebase more maintainable and therefore helps to cope with the requirements in a fast-paced environment [TSZ09].

Another common software engineering technique to improve software maintainability is called *Separation of concerns* (SoC). SoC divides source code into logical and reusable parts. For example, the implementation of the use case of borrowing a book from a library could be divided into different services. An *AccountService*, a *BookService* and a *LibraryService*, where the *AccountService* is responsible for managing library users, the *BookService* for managing all the books in the library (e.g. rental statuses of books) and the *LibraryService* for taking care of the whole rental process (e.g. check account ⇒ check book status ⇒ rent book). Each of the services provides one or more methods to communicate with the instance. These methods can be roughly divided into methods that return a result and methods that perform a task. *BookService#borrowBook*, for example, is a command method, while *AccountService#getAccount* is a query method. In literature, the strict separation between those two kinds of methods is known as the *Command-Query Separation* (CQS) principle [Mey88]. By definition, query methods return results, but are not allowed to change the state of objects, while command methods are allowed to change the state of objects, but must not return results [Mey88]. In the previous example, the *LibraryService* depends on the other two services. In case of a new book rental, methods of the dependencies will be called sequentially (e.g. *AccountService#getAccount* ⇒ *BookService#checkBook* ⇒ *BookService#borrowBook*), taking possible errors into account. The sequential execution in this example could also be expressed with monadic composition.

Due to the fact, that the use of monadic structures can reduce complexity [Nag19] and that the usage of query and command methods like in the previous example could be expressed with monadic composition, this work analyzes the impacts of monadic methods (i.e. methods returning monadic structures) of the data access layer on software complexity.

It is important to mention that the definition of command methods is relaxed for quantitative analysis, so that it is allowed to return values. There are two reasons for this: First, it is common in Java to return values from command methods in the data access layer of the project. For example, the *insert* method in a database service often returns the inserted object or the ID of the inserted object. Second, the rewritten monadic version of a command method must return a monad, so that it can be called a monadic method.

1.2 Aim of the Work

The use of the functional or object-oriented programming paradigm offers paradigm-specific advantages. However, the simultaneous use of several paradigms provides additional benefits [TMM18, Nar09, PSG12]. This work researches the impacts of using properties coming from the functional programming paradigm in a mainstream object-oriented programming language. More precisely, the impact of the usage of monadic query and command methods of the data access layer on software complexity was examined.

Starting from this aim, the following research question can be derived: ***What practical impact does the usage of monadic query and command methods of the data access layer in application software have on software complexity, using the mainstream object-oriented programming language Java?*** This question can be further broken down into the following sub-questions:

- Can the software complexity be reduced by using monadic query and command methods?
- How do the different results of the complexity metrics relate to each other? (e.g. do all results say 'complexity decreased/increased'?)

The overall aim of this thesis is to draw conclusions about the correlation between a monadic programming style in an object-oriented programming language and the complexity of the underlying source code.

1.3 State of the Art

The book *Programming Languages: Principles and Paradigms* [GM10] and the paper *Concepts and paradigms of object-oriented programming* [Weg90] provide fundamental and still up-to-date knowledge about programming paradigms, including functional and object-oriented programming. In his work [Pet18], Tomas Petricek gives a broad understanding of the history of *Monads* and explains different perspectives regarding this subject. Teatro et al. demonstrate how to compactly implement *Monads* in C++, an object-oriented programming language [TMM18].

Regarding the key question of this work, the paper *Comparing software complexity of monadic error handling and using exceptions* [Nag19] shows that using monadic structures for error handling has a positive impact on software complexity. Based on these results, this work examines whether the usage of monadic query and command methods in a mainstream object-oriented programming language can achieve the same results.

The complexity measurement is done by using three different techniques to measure software complexity. Two of the techniques are based on research papers, while one is a trivial metric. The trivial metric is called *Non-Comment Lines of Code* (NCLOC) and works by counting the source lines of code (leaving out comment and blank lines). The

second technique is called *Cyclomatic Complexity* and was first introduced in the paper *A Complexity Measure* [McC76]. The third and final technique was developed by Maurice Howard Halstead and presented in the book *Elements of software science* [Hal77].

1.4 Methodological Approach

1. **Literature Review**

   The search for relevant papers and journals was conducted by a backward and forward reference search based on the core literature. Additionally, various search engines like IEEE Xplore[1], ACM Digital Library[2] and SpringerLink[3] were used to find related papers. The collected literature served as a theoretical basis for the practical part of this thesis.

2. **Selection of a Java-based *Open Source Software* (OSS) Project**

   Next, an appropriate project for the quantitative analysis needed to be selected. The restrictions for the selection were that the project must fall under the category of an application software and must be written in the mainstream object-oriented programming language Java. Furthermore, no monadic structures are allowed to be used in the selected project and there also has to be a data access layer. The open-source Android application software OpenKeychain[4] met these restrictions and was therefore chosen for the study.

   Regarding the search for an OSS project, GitHub[5] was used as a platform.

3. **Rewrite Rules**

   To make the rewriting process transparent and replicable, as well as keeping the rewriting bias as small as possible and maintaining objectivity, an extensive set of rules for the rewriting process was created.

4. **Monadic Programming Style**

   Following the definition of the rewrite rules and the selection of a project, query and command methods of the data access layer were identified and then rewritten in a monadic programming style. Source code which uses the rewritten methods had to be rewritten as well to make the project compile again. All code changes were made in compliance with the established set of rules for the rewrite process. Furthermore, software tests were run regularly and a code review was conducted to minimize programming mistakes.

[1] https://ieeexplore.ieee.org/Xplore/home.jsp
[2] https://dl.acm.org/
[3] https://link.springer.com/
[4] https://www.openkeychain.org/
[5] https://github.com/

5. **Evaluation**

   This thesis uses a quantitative evaluation approach. Before changes to the source code were made, the complexity metrics *NCLOC*, *Cyclomatic Complexity* [McC76] and *Halstead Difficulty* [Hal77] were collected for all methods. After applying a monadic style to query and command methods in the data access layer and refactoring the places of use of those methods, the complexity metrics were collected again for the resulting code. Then the results for all modified methods were compared and conclusions were drawn.

1.5 Structure of the Work

Following this chapter, which has given a general overview of this thesis, programming paradigms are explained. In particular, the chapter Programming Paradigms deals with the object-oriented and the functional programming paradigm (see Chapter 2).

Based on the explanation of the function programming, the history and common use cases of monads are covered in detail in Chapter 3.

Chapter 4 first gives an introduction to software complexity. Following the introduction, five different metrics and metric sets for measuring software complexity are discussed.

After explaining the required basic knowledge, the necessary information for the evaluation is described in Chapter 5. This information includes the tools used, the presentation of a set of rules for the rewriting process, the procedure of the rewriting process and how it is ensured that as few mistakes as possible are made during the rewriting process.

Chapter 6 presents the results that were obtained from the rewriting of parts of the source code. First, the results are generally analyzed and then interpreted in detail. This chapter ends with a discussion about the results.

The last chapter concludes with a summary of the key findings and gives an outlook towards future research (see Chapter 7).

CHAPTER 2

Programming Paradigms

This chapter first gives a brief introduction to programming paradigms in general, followed by a detailed explanation of the programming paradigms relevant to this work, namely the object-oriented and functional programming paradigm. For each paradigm, a brief review of the background is given and fundamental concepts are explained.

2.1 Introduction

The word *paradigm* has its origin in the Greek language and can be described as *a pattern* or *a model* [Pre20]. In the context of programming, paradigms are a categorization of different styles of programming. Some commonly known programming paradigms are imperative programming, object-oriented programming, functional programming and logic programming, whereby the last two can be summarized under the paradigm declarative programming [GM10].

Imperative programming is a programming style closely related to how computers work [GM10, Weg90]. The computation for solving a specific problem consists of a sequential execution of steps, which lead to the solution of the problem. While executing these steps, the program state is modified. In summary, imperative programming is about *how* something is computed. The *how* is expressed by the step-by-step execution. In contrast, declarative programming has a close relation to mathematics and logic and is about *what* is to be computed [Weg90].

Two code listings are used to illustrate the difference between the imperative (*how*) and the declarative (*what*) programming approach. Both listings show a program for counting string occurrences in a list of strings. Listing 2.1 shows an imperative implementation of the *countOcc* method, written in the programming language Java, while Listing 2.2 is written in a declarative programming style using the functional programming language Haskell.

Listing 2.1: Imperative *countOcc*

```
int countOcc(String item, List<String> strings) {
    int count = 0;
    for (String line : strings) {
      if (item.equals(line)) {
        count++;
      }
    }
    return count;
}
```

Listing 2.2: Declarative *countOcc*

```
filterS :: (String -> Bool) -> [String] -> [String]
filterS _ [] = []
filterS f (x:xs)
  | f x = x : filterS f xs
  | otherwise = filterS f xs

lengthS :: [String] -> Int
lengthS [] = 0
lengthS (x:xs) = 1 + lengthS xs

countOcc s = lengthS . filter (== s)
```

In the imperative implementation, an integer variable indicating the occurrence count of the passed string is first created. The passed list of strings is then explicitly iterated. Within the loop, it is checked whether elements of the list are equal to the passed string. If this is the case, the count variable gets incremented by one. After the iteration, the occurrence count gets returned. This step by step approach corresponds to the *how* or the imperative approach.

On the other hand, the declarative version defines the function *countOcc* by composing the two functions *filterS* and *lengthS*. Both functions show a naive implementation, which is strongly based on the Haskell Prelude[1] implementation. The *filterS* function filters the list so that it only contains elements matching the passed string. After filtering the list, the resulting list is passed to the *lengthS* function, which calculates and returns the number of elements in the filtered list. In the declarative version of *countOcc*, the call of the *filterS* function abstracts away how the list is filtered. This example reflects the behavior of the *what* or the declarative approach. The declarative version of the *countOcc* function can also be described as *referentially transparent* (see Section 2.3.3) [Hud89].

Object-oriented programming is a technique that uses objects and the communication between objects to perform computation (this is one of several different views [Nob09]).

[1] https://www.haskell.org/onlinereport/standard-prelude.html

In functional programming, the basic approach for writing programs is to apply functions to arguments [Hug89]. In contrast, problem-solving in the logic programming paradigm is based on using logical deductions [GM10].

The paradigms object-oriented programming (see Section 2.2) and functional programming (see Section 2.3) are explained in detail in the following sections. In contrast, the imperative and logic programming paradigm is not considered further, since the work in this thesis mainly builds upon the concepts of object-oriented and functional programming.

2.2 Object-oriented Programming

2.2.1 Background

In the early 1960s, Ivan Sutherland developed a system called Sketchpad, which was the first system using an object-oriented programming style [Ren82]. However, the term *object-oriented* has its roots in the time of the development of the programming system Smalltalk [Ren82]. The programming language itself is heavily influenced by Alan Kay's programming language called FLEX (Flexible Extendable Language), which in turn is based on the Simula programming language [Ren82]. Simula was the first language that introduced the concepts of classes, objects and inheritance [Weg90]. Classes and objects lay the foundation for the concepts of encapsulation, inheritance, subtypes and dynamic dispatch. These concepts are of fundamental importance for the object-oriented programming paradigm and are therefore explained in detail in the terminology section (see Section 2.2.2) [GM10]. After Smalltalk was invented around 1970, the language quickly gained popularity and was very successful, also in a commercial way [GM10]. Later, in the mid-1980s, the programming language C++ was released and in 1992, the team led by Jim Gosling introduced the first version of Java [KF19]. Nowadays, object-oriented programming is one of the best-known programming paradigms and according to the TIOBE[2] index of the year 2018, seven out of the ten highest ranked programming languages support object-oriented programming techniques [AYK19].

2.2.2 Terminology & Fundamental Concepts

James Noble quotes three different views on the object-oriented programming paradigm in his paper [Nob09]. The first view states that object-oriented programming is about modeling. The second view describes an object-oriented system as a structure, consisting of objects sending messages to each other. The third view defines object-oriented programming by its supporting techniques like data abstraction, polymorphism and inheritance. James Noble describes object-oriented programming as a combination of all three views [Nob09]. The concepts of objects, classes, encapsulation, inheritance, subtyping and dynamic dispatching form the basis for these views. However, the concept of classes is not essential to the object-oriented paradigm [GM10]. In order to explain the

[2] https://www.tiobe.com/tiobe-index/

object-oriented programming paradigm as a whole, this section covers the fundamental principles on which object-oriented programming is based.

Object

The term *object* can be described as a container that encapsulates a certain state by using so-called instance variables and provides an interface (not to be confused with Java interfaces) for manipulating and accessing this state [Weg90, GM10]. Communication via the interface is accomplished by calling methods which are also referred to as member functions. s [GM10]. In the Smalltalk world, calling methods is also referred to as *sending a message to an object*.

Class

A class is a structure which describes how the state and the communication interface of an object look like. This description is used as a template for creating objects [Weg90]. In Java, a particular description of a member variable consists of a name, a type, and a visibility modifier, while the description of a method consists of a name, a signature, a visibility modifier and an implementation [GM10]. The class itself also has a name, a visibility modifier and a constructor (method for creating objects). Listing 2.3 shows a simple Java class representing a name of a person. The name consists of a first and a surname, which is stored in two instance variables.

Listing 2.3: A class written in Java

```
public class Name {
    private String firstName;
    private String surname;

    public Name(String firstName, String surname) {
        this.firstName = firstName;
        this.surname = surname;
    }

    public String getFirstName() {
        return this.firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getSurname() {
        return this.surname;
    }
```

```
    public void setSurname(String surname) {
        this.surname = surname;
    }

    public String getFullName() {
        return this.firstName + " " + this.surname;
    }
}
```

The member functions *getFirstName*, *getSurname* and *getFullName* can be used to read the current state while the member functions *setFirstName* and *setSurname* are used to change the state of an object. For creating a new Java object in memory (an instance), the *new* keyword is used. In particular, Listing 2.4 shows how the instantiation of the *Name* class (see Listing 2.3) is done.

Listing 2.4: Instantiation of the *Name* class

```
Name name = new Name("Jane", "Doe");
```

The constructor of the *Name* class is used to initialize the two member variables *fistName* and *surname*. After the instantiation, the newly created variable *name* is an instance of the *Name* class.

Listing 4.1 shows another example for a Java class. This class provides the public method *buildName*, which creates and returns a *Name* object from a passed string. No parameters are required to instantiate the class.

Encapsulation

The concept of encapsulation is about wrapping each value "in an encapsulation (its type) which provides the operations that manipulate it" [GM10, p. 265]. Information hiding then enables making certain data or operations invisible to the outside world [GM10]. Access to the data or operations is enabled through a defined interface (e.g. methods; not to be confused with Java interfaces).

In Java, information hiding is accomplished through visibility modifiers. In the *Name* class example from the previous section (see Listing 2.3), the two instance variables are hidden from the outside world using the *private* visibility modifier. The *public* methods allow the instance variables to be read and updated.

Subtyping

Subtyping is a form of polymorphism, particularly *inclusion polymorphism*, and describes a relation between classes [CW85]. A definition of whether a type is a subtype of another type can be found in the *Liskov Substitution Principle*, formulated in 1994 by Barbara

Liskov and Jeannette Wing. The *Liskov Substitution Principle* defines the subtype relation as follows:

> "Let ϕ(z) be a property provable about objects x of type T. Then ϕ(y) should be true for objects y of type S where S is a subtype of T." [LW94, p. 1812]

In other words, a type S is a subtype of a type T, if and only if an object of type S can be used where an object of type T is expected [LW94].

Listing 2.5, for instance, creates a class *TitledName*, which is a subtype of the previously created *Name* class (see Listing 2.3). As a result, any method expecting objects of the type *Name* is also accepting objects of the type *TitledName*. This works because instances of *TitledName* also contain all members of the *Name* class [GM10]. Furthermore, non-private methods from the parent class can be redefined in the subclass, which is also known as method overwriting [GM10]. Listing 2.5 overwrites the *getFullName* method from the parent class.

Listing 2.5: Subtype of the *Name* class

```
public class TitledName extends Name {
    private String preNominalLetters;

    public TitledName(String preNominalLetters,
            String firstName, String surname) {
        super(firstName, surname);
        this.preNominalLetters = preNominalLetters;
    }

    @Override
    public String getFullName() {
        return preNominalLetters + "␣" + super.getFullName();
    }
}
```

Inheritance

Inheritance is a concept that deals with code reusability and sharing behavior. For example, if one class is a subclass of another one (superclass) and the subclass does not overwrite any of the methods from the superclass, then the subclass inherits all of the instance variables and methods from the superclass [GM10]. This means that the subclass can also use all inherited properties, which are defined as non-private in the superclass. In the previous example, in which the class *TitledName* (see Listing 2.5) is a subtype of the class *Name* (see Listing 2.3), the *TitledName* class inherits all of the

methods, except for *getFullName*, from the superclass. One could now ask about the difference between inheritance and subtyping. Gabbrielli et al. say that subtyping is about using an object in other contexts, while inheritance is about the possibility of reusing object manipulating code [GM10].

Dynamic Dispatch

The dynamic dispatch technique is responsible for choosing to call the correct method of an object [GM10]. In the previous section about subtyping (see Section 2.2.2), the term *method overwriting* was introduced. If a method is overwritten, there theoretically exist two or more versions of the same method [GM10]. Dynamic dispatch performs its task by calling the method of an object's dynamic type. The dynamic type corresponds to the type which an object has during runtime and not the type of the reference to an object [GM10]. Listing 2.6 exemplifies this by using the previously defined classes *Name* (see Listing 2.3) and *TitledName* (see Listing 2.5). In this example, two objects with the static type *Name* are created. However, the variable *n2* refers to an object with the dynamic type *TitledName*. When the method *getFullName* is called on both objects, each invokes the method of its dynamic type. This technique is considered the heart of the object-oriented programming paradigm [GM10].

Listing 2.6: Dynamic Dispatch

```
Name n1 = new Name("John", "Doe");
Name n2 = new TitledName("Dr.", "Jane", "Doe");

System.out.println(n1.getFullName()); // John Doe
System.out.println(n2.getFullName()); // Dr. Jane Doe
```

2.3 Functional Programming

2.3.1 Background

The origin of the functional programming paradigm reaches back to the 1930s due to its strong influence from the lambda calculus, which is often referred to as the first functional language [Hud89]. The lambda calculus was invented by Alonzo Church and can be described as a formal system for computation using functions. In 1958, about 30 years later, the programming language LISP (short for *List Processing*) was invented by John McCarthy [Tur13].

LISP was the first functional programming language, but its relationship to the lambda calculation was rather small [Hud89]. At first, LISP was based on Kleene's theory of first-order recursive functions, but over time all versions were based on Church's lambda calculus [Tur13]. Between LISP and the development of the still well-known functional programming language Haskell in 1987, several other functional programming languages were devised. Examples include ISWIM (If you See What I Mean), APL (A

Programming Language), FP (Functional Programming), ML (Meta Language) and Miranda [Hud89, Tur13]. ISWIM, for instance, introduced *let* and *where* clauses [Hud89], which are still used in programming languages like Haskell. The FP programming language, "which came after APL, was certainly influenced by the APL philosophy" [Hud89, p. 372] and was very influential for the functional programming paradigm due to John Backus' Turing Award lecture *Can programming be liberated from the von Neumann style?* [Hud89]. Around the same time, the programming language ML, which was considered "the most practical functional language at the time it appeared" [Hud89, p. 374], was also under development. Two years before the development of Haskell started, the lazy and pure functional programming language Miranda was released and had a commercial success [Tur13]. In 1990, Haskell 1.0 was released. Despite the success of the language Miranda, Haskell gained the upper hand a few years later [Pey07]. This can be attributed to the fact that Haskell is free for commercial use [Pey07]. Moreover, Haskell is a feature-rich and purely functional programming language with design influences reaching from ISWIM to Miranda [Hud89].

The following sections explain the fundamental concepts of the functional programming paradigm (see Section 2.3.3) in more detail. Due to its strong influence on the functional programming paradigm, the lambda calculus is explained separately (see Section 2.3.2).

2.3.2 Lambda Calculus

This section provides a deeper understanding of the lambda calculus, but in order to remain within the scope of this work, only the untyped lambda calculus is discussed in the following. Furthermore, the whole section and all of the presented formulas are based on the paper [Hud89].

The lambda calculus was developed by Alonzo Church in the 1930s and is an abstract model that uses function abstraction and application for computation. The syntax of the lambda calculus is defined by a set of lambda expressions (Exp) consisting of three parts. The first part is simple names, called identifiers (Id). The second part has the form ($e_1\ e_2$) and is called an application. e_1 and e_2 are also expressions themselves ($e_1, e_2 \in Exp$). Specifically, e_1 is a function and e_2 is an argument that is applied to the function. The last type of expressions are abstractions. They have the form $\lambda x.e$ and represent functions, where x is a formal parameter ($x \in Id$) and e is the body of a function ($e \in Exp$). Evaluation of lambda calculus expressions happens through applying rewrite rules, which are based on substitution. The basic substitution is written as $[e_1/x]e_2$, which means that e_1 replaces x in e_2. To avoid name conflicts, x must occur free in e_2. The notion of free variables for expressions is defined as follows:

$$fv(x) = \{x\}\ where\ x \in Id$$

$$fv(e_1\ e_2) = fv(e_1) \cup fv(e_2)$$

$$fv(\lambda x.e) = fv(e) \setminus \{x\}$$

These rules can be used to define the substitution rules for all expressions in the lambda calculus:

$$[e/x_1]x_2 = \begin{cases} e, & \text{if } x_1 = x_2 \\ x_2, & \text{otherwise} \end{cases}$$

$$[e_1/x](e_2\ e_3) = ([e_1/x]e_2)([e_1/x]e_3)$$

$$[e_1/x_1](\lambda x_2.e_2) = \begin{cases} \lambda x_2.e_2, & \text{if } x_1 = x_2 \\ \lambda x_2.[e_1/x_1]e_2, & \text{if } x_1 \neq x_2 \text{ and } x_1 \notin fv(e_1) \\ \lambda x_3.[e_1/x_1]([x_3/x_2]e_2), & \text{otherwise, } x_3 \neq x_1,\ x_3 \neq x_2 \text{ and } x_3 \notin fv(e_1) \cup fv(e_2) \end{cases}$$

Using the substitution rules as a base, the lambda calculus rewrite rules are defined as follows:

α-conversion

$$\lambda x_1.e \iff \lambda x_2.[x_2/x_1]e,\ \textit{where}\ x_2 \notin fv(e)$$

The α-conversion states that x_1 can be substituted with x_2, if x_2 does not occur free in e. Basically, the α-conversion expresses the renaming of formal parameters. Listing 2.7 shows an example of the α-conversion in Haskell syntax. The lambda expressions stored in the variables *f1* and *f2* do not differ, except for the names of the formal parameters. They are also called *lambda equivalent* [GM10].

Listing 2.7: α-conversion example

```
f1 = \x -> x
f2 = \y -> y
```

β-conversion

$$(\lambda x.e_1)e_2 \iff [e_2/x]e_1$$

The second rewrite rule is called β-conversion. This rule describes the application of arguments to functions. Specifically, the rule says that the application of the argument e_2 to the lambda expression equals the lambda expression e_1, where x is replaced by e_2. The Haskell example in Listing 2.8 illustrates the β-conversion by providing a function *f*, which shows the equivalence between the left and the right side of the β-conversion.

Listing 2.8: β-conversion example

```
f a = (\x -> x + x) a == a + a
```

η-conversion

$$\lambda x.(e\ x) \iff e, \text{ if } x \notin fv(e)$$

In the η-conversion, the abstraction on the left side has another application $(e\ x)$ in its function body. So if a parameter a is applied to the abstraction, it is directly passed on to the lambda expression in the function body. If x does not occur free in e, then the abstraction itself is equal to the lambda expression inside the function body. Listing 2.9 shows an example for the η-conversion.

Listing 2.9: η-conversion example

```
y = \x -> x
f a = (\x -> (y x)) a == y a
```

Fixpoint Theorem

The fixpoint theorem implies that "every lambda expression e has a fixpoint e' such that $(e\ e') \overset{*}{\Leftrightarrow} e'$." [Hud89, p. 366]. The symbol $\overset{*}{\Leftrightarrow}$ expresses intraconvertibility, which means that the left expression can be derived from the right side and the right expression from the left side by applying applying zero or more α-, β-, η-conversions. The message of this theorem is that "any recursive function may be written nonrecursively" [Hud89, p. 366]. The lambda calculus uses this theorem to simulate recursion [Hud89].

2.3.3 Fundamental Concepts

The functional programming paradigm is a subcategory of the declarative programming paradigm [GM10]. Unlike in imperative languages, no implicit state exists in declarative programming languages [Hud89, GM10]. In (pure) functional programming, the program state is handled explicitly using the "underlying model of computation" [Hud89, p. 361], the function. "Higher-order functions and recursion are the basic ingredients of this stateless computational model." [GM10, p. 334].

Functions

A function in the sense of mathematics is a mapping of values in a set A to values in a set B, which is often written as $f : A \to B$. Functional programming is based on the application of functions, where pure functions behave like functions from mathematics [Hug89]. A pure function is a function without side effects (e.g. modifying the value of a variable) where it is always returned the same result for the same argument. [Pey07]. If functions are pure, they are also referred to as *referentially transparent*, which means that variables or expressions can be replaced by their values or results [Hug89]. *Referential transparency* also opens up the possibility of equational reasoning about programs [Hud89]. Paul Hudak refers to this *pure* programming style as "the hallmark of the functional programming paradigm"[Hud89, p. 362].

Recursion

Recursion in the context of functions is the existence of a function call to the same function within the function body [GM10]. This concept is one of the fundamental concepts in stateless computation, hence it can be used to handle state explicitly [GM10]. The Haskell recursion example (see Listing 2.10) shows a naive implementation of the Haskell Prelude function *length* called *countElems*. In this example, the recursive helper function *countElemsHelper* carries the count state by calling itself with the previous count (c) plus one and the list tail until the passed list is empty.

Listing 2.10: Recursion example

```
countElems xs = countElemsHelper 0 xs
  where
    countElemsHelper c [] = c
    countElemsHelper c (x:xs) = countElemsHelper (c+1) xs
```

Higher-Order Functions

The term *higher-order function* defines a function, which takes one or more functions as an argument and/or returns a function as a result. They are another fundamental concept for stateless computation [GM10] and can be used to improve modularity [Hud89]. The *map* function of the Haskell Prelude is a well-known and widely used example for a higher-order function. The function has the type $map :: (a \to b) \to [a] \to [b]$ and returns a new list by applying the passed function $((a \to b))$ to each element of the passed list $([a])$.

CHAPTER 3

Monads

3.1 Background

The last chapter introduced the basics of functional programming (see Section 2.3). Furthermore, the term *pure function* was explained. The programming languages Miranda and Haskell, for instance, are pure languages. In Haskell, monads are used to integrate side effects such as network requests and keyboard input handling into the language [Wad93].

The concept of monads was introduced in 1958 and originates from category theory, a subcategory of mathematics [Pet18]. A monad in the mathematical sense is referred to as a *triple*, *triad*, *standard construction* or *fundamental construction* and is described by the triple (T, η, μ) on a category C [Rot86]. In category theory, a functor is a mapping from one category C into another category D, written as $functor\ F : C \to D$ [Rot86]. The T in the triple is a functor which is a mapping from the category C into the same category C: $T : C \to C$, called endofunctor [Rot86]. The symbols η and μ are two natural transformations where η transforms the identity of C and μ transforms $T \circ T$ to the endofunctor T [Rot86].

Eugenio Moggi and Philip Wadler introduced the term monad to computer science. Moggi used it in the area of the semantics of programming languages and logical reasoning, while Wadler first used it as a tool for programmers [Pet18]. In 1990, Philip Wadler defined a monad as an object wrapper M, called an operator, consisting of the three functions $map :: (x \to y) \to (M\ x \to M\ y)$, $unit :: x \to M\ x$ and $join :: M(M\ x) \to M\ x$, which satisfy specific laws [Wad90] (see Section 3.2.1).

In the paper *Monads for functional programming*, the same author defines monads as "a triple (M, unit, $\star$) consisting of a type constructor M and two operations of the given polymorphic types" [Wad93, p. 239]. It is stated that *unit* is defined as the previous definition and $\star$ is defined as $\star :: M\ a \to (a \to M\ b) \to M\ b$ [Wad93]. However, both

definitions are equal since $m \star k = join(map\ k\ m)$ [Wad93, Pet18]. The following of this work specifically relates to the second definition since it is the one commonly used today. Furthermore, the $\star$ function is often referred to as *bind* and is defined by Haskell as the infix operator >>=. The *unit* operation is in Haskell named *return*. In the following, the name *bind* is preferred in writing and for code examples, the infix operator >>= is used.

3.2 Definition

A type constructor is a constructor that takes zero or more types as an argument and returns a new type [HPF99]. A type constructor with zero type arguments is called a nullary type constructor [HPF99]. For example, the Haskell data type *Bool* is a nullary type constructor with the two value constructors *True* and *False*. Listing 3.1 shows another example of a nullary type constructor. On the other hand, a constructor taking one type parameter is called a unary type constructor.

Listing 3.1: Nullary type constructor

```
data Tetromino = I | O | T | J | L | S | Z
```

In the world of computer science, a monad is defined by a unary type constructor M and the two functions *unit* and *bind* with the following signatures [Wad93]:

$$unit\ ::\ a \to M\ a \tag{3.1}$$

$$bind\ ::\ M\ a \to (a \to M\ b) \to M\ b \tag{3.2}$$

The *unit* function (see 3.1) takes a value of the type a and returns a monad Ma which wraps the passed value of the type a. The function *bind* (see 3.2) is used to transform values wrapped inside a monad by applying the passed function of the type $a \to M\ b$ to the value wrapped inside the monad of the type $M\ a$. Both of the operations have to satisfy the laws *Left Identity*, *Right Identity* and *Associativity*, which are explained in the following subsection.

3.2.1 Monad Laws

Left Unit

$$(unit\ a) >>= f = f\ a \tag{3.3}$$

Equation 3.3 shows the definition of the *Left Unit* or *Left Identity* law [Wad93]. This law stipulates that if a value a gets wrapped into a monad using the function *unit* and then transformed by applying the *bind* function with a function f, the result must be equal to the application of f to a. The code example in Listing 3.2 shows how it can be verified if this law holds for specific values in Haskell. If, for example, the *check* function is called

with the *dup* function from the same code listing and the value *2*, *True* is returned as a result. In practice, monad laws can be tested with value generators.

The example in Listing 3.2 uses the built-in *Maybe* monad. For now, it is just essential to know that the type *Maybe* is a monad in Haskell. Later (see Section 3.3), the behavior of *Maybe* and other types of monads is discussed in detail.

Listing 3.2: Left identity law example

```
check f a = ((return a) >>= f) == f a
dup = \x -> Just (x * x)
```

Right Unit

$$m >>= unit = m \tag{3.4}$$

The *Right Unit* or *Right Identity* law, defined in Equation 3.4 [Wad93], shows that calling the *bind* operation with a monad m and the *unit* function as parameters must return the monad m itself as a result. From this, it can be said that if *unit* is applied to a value inside a monad, the value must only be wrapped into a monad again. Hence, the operation *unit* does not change the monad m. Listing 3.3 illustrates this law by showing a function for checking if the law holds for a monadic value in Haskell. If the function *check* from Listing 3.3 is called with the value *Just 1* or *Nothing*, *True* is returned as a result again.

Listing 3.3: Right identity law example

```
check m = (m >>= return) == m
```

Associativity

$$m >>= (\backslash x \rightarrow f\ x >>= g) = (m >>= f) >>= g \tag{3.5}$$

In mathematics, a binary operation $\circ$ is called associative if and only if $a \circ (b \circ c) = (a \circ b) \circ c$ holds [GM10]. The associativity law must also apply to the operator *bind*, as Equation 3.5 shows. From this, it follows that "the order of parentheses in such a computation is irrelevant" [Wad93, p. 243]. The following code example (see Listing 3.4) shows again how it can be checked if the associativity law holds for specific values in Haskell. For example, when calling the function *check* with the value *Just 2* or *Nothing* for the parameter m, *f1* for the parameter f and *f2* for the parameter g, the function returns *True*.

Listing 3.4: Associativity law example

```
f1 = \x -> Just (x * 2)
f2 = \y -> Just (y * 4)

check m f g = (m >>= (\x -> f x >>= g)) == ((m >>= f) >>= g)
```

3.3 Types of Monads

There exist many different types of monads. This section emphasizes fundamental and widely used monads and explains them in detail. Each monad section is structured in the following way: First, a brief introduction is given. Second, it is shown how the monad can be implemented. Unless otherwise stated, the programming language Haskell is used. If monad instances are declared in Haskell, applicative and functor instances must also be declared. For convenience, however, the application and functor instances are omitted. Based on the implementation, examples are given and use cases of the monad are discussed.

3.3.1 Identity Monad

Introduction

A monad in its simplest form is known as the *Identity* monad [Wad93]. The *unit* operator is equal to the identity function, which is defined as $f(x) = x$ [Wad93, GM10]. The function *bind* does nothing other than applying the function to the value (e.g. $m >>= k = k\ a$) [Wad93]. In literature, the monad itself, *type* $M\ a = a$, is defined by a type constructor that returns the passed type as a result [Wad93].

Implementation

As stated before (see Section 3.3.1), in literature, the *identity* monad is defined as a parameterized type synonym for an arbitrary type a. In Haskell, however, it is not possible to define a monad instance from such types. Listing 3.5 shows what a possible implementation could look like in Haskell.

Listing 3.5: *Identity* monad implementation

```
newtype Id a = Id a

instance Monad Id where
  return = Id
  (Id x) >>= f = f x
```

First, the unary type constructor Id, with one nullary value constructor, is declared. In this definition, a value of the passed type a gets wrapped in Id. The declaration of the monad instance starts in the following source line and implements the functions *return* and $>>=$ (*bind*). *return* is defined by $return = Id$, which is a shorter writing style for $return\ x = Id\ x$. The *bind* operation simply applies the function to the unwrapped value x.

Usage

The *Identity* monad has limited practical relevance, but a possible application is the use as a monad transformer base case [Jon95]. A monad transformer is another type

constructor that can be used to combine two different monads [Jon95]. For example, the Haskell package *transformers*[1] defines the *reader* monad by reusing the corresponding monad transformer *StateT* combined with the *identity* monad (see Listing 3.6) [GP01].

Listing 3.6: *Reader* monad declaration using transformer

```
type Reader r = ReaderT r Identity
```

Another possible application is the use in functions, where monadic arguments are expected. Consider a function with the signature "intComputation :: **Monad** m =>m **Int** -> m **Int** -> m **Int**" in Haskell syntax. This function takes two monadic integer values and returns a monadic integer, which is the product of the passed values. Due to the monadic values, the function *intComputation* can be reused in different ways. For example, it can be called with values wrapped in an *IO* monad. This opens up the possibility of printing debug messages to the standard output. If one simply wants to perform the computation on integer values, the *identity* monad can be used. Listing 3.7 illustrates both of the mentioned use cases of the *intComputation* function.

Listing 3.7: *Identity* monad use case

```
debuggableVal x = do
  putStrLn $ "Debug: val = " ++ (show x)
  return x

intComputation :: Monad m => m Int -> m Int -> m Int
intComputation m1 m2 = (*) <$> m1 <*> m2

runId = intComputation (Id 5) (Id 3)
runDebug = intComputation (debuggableVal 5) (debuggableVal 3)
```

3.3.2 Exception Monad

Introduction

The second monad to be introduced is called the *Exception* monad. This monad is used for computations that either return a value or fail with an exception [Wad93]. In particular, this monad wraps either a result with the type of a passed type parameter or an exception with the type of a fixed or passed type parameter. The *unit* function's implementation forwards the passed parameter to the value constructor which defines a successful computation. This is equal to the *unit* implementation of the *Identity* monad. On the other hand, *bind* cannot be implemented like in the *Identity* monad. This is due to the two value constructors of the *Exception* monad data type since the check for all possible *Exception* monad values must be exhaustive. In other words, *bind* must handle the failure case as well as the successful case of the passed monad. As stated at the beginning of this section, the failure case *Exception* monad wraps either an exception of

[1] https://hackage.haskell.org/package/transformers

the type of a fixed or passed type parameter. For example, the paper [Wad93] defines the type of the exception as *String*, while the official Haskell Wikipedia defines it with an extra type parameter [Has20]. In the following of this work, the latter approach is used because it offers better flexibility compared to a fixed type.

Implementation

Based on the introduction of the *Exception* monad, the data type is named *ResultOrError* and consists of two type constructors, *e* and *a*, and two value constructors, *Return* and *Raise*. The naming of the value constructors is adopted by [Wad93]. In Haskell, the type of *Exception* monad is called *Either*, which offers better flexibility since the naming also allows two different success values to be held [The01a].

The first line in Listing 3.8 defines the new type *ResultOrError*. For example, the expression "Return 'a'" constructs a value of the type *ResultOrError e Char*, while the expression "Raise "404"" constructs an error-indicating value of the type *ResultOrError [Char] a*.

Listing 3.8: *Exception* monad implementation

```
data ResultOrError e a = Raise e | Return a

instance Monad (ResultOrError e) where
  return = Return
  (>>=) (Raise err) _ = Raise err
  (>>=) (Return x) f = f x
```

As already mentioned in the introduction of this monad, the *return* operation wraps the passed value in the successful value constructor *Return*. In *bind*, pattern matching checks for both possible values of the passed monad. If the value is indicating an error (*Raise err*), the passed function is ignored and a failure value with the same exception value *err* of the passed monad gets returned. In case of a passed successful value *Return x*, the passed function gets applied to the unpacked value *x*. The result of the function application is also the result of *bind*.

Usage

Exception monads can be used to deal with errors in computations in a pure way [Nag19]. This monad is particularly useful, if the specific reason for failure in computation is of importance, since it can be embedded in the *ResultOrError* data type. Consider the *head* function from the Haskell Prelude[2]. For non-empty lists, this function returns the first element of the list. If an empty list is passed, the function prints an error and terminates the program [HPF99]. This function can also be implemented safely using the *ResultOrError* data type (see Listing 3.9).

[2] https://www.haskell.org/onlinereport/standard-prelude.html

Listing 3.9: Safe *head* function using the *Exception* monad

```
safeHead :: [a] -> ResultOrError String a
safeHead [] = Raise "List must not be empty"
safeHead (x:xs) = Return x
```

Instead of printing an error and terminating the program for empty lists, the function *safeHead* returns the value constructor *Raise* with a string message argument. Pattern matching can be used to react to the returned failure. Since *ResultOrError* is also a monad, chaining multiple computations with the same *ResultOrError* type together or passing this type to a function that expects a monad is possible.

Consider the use case of parsing two strings in JavaScript Object Notation (JSON) format and returning a tuple of both successfully parsed JSONs. In case of a parsing error, the failure reason should be returned. Also, consider the function *parseJson* with the signature $String \rightarrow ResultOrError\ Exception\ Json$ as given. This function tries to create a JSON value out of a passed string. If the parsing is successful, the value of the type *Json* gets wrapped in the *Return* value constructor. Otherwise, *Raise* with the reason for the failure is returned. For keeping the code examples short, the implementation of *parseJson Json* and *Exception* is omitted. For the implementation of the use case, the monadic operation *bind* can be used. Listing 3.10 shows a naive implementation.

Listing 3.10: Parse JSON use case

```
data Json = ...

parseJson :: String -> (ResultOrError String Json)
parseJson s = ...

jsonUseCase1 s1 s2 =
  (parseJson s1) >>= \j1 ->
    (parseJson s2) >>= \j2 ->
      return (j1, j2)
```

If one of the *parseJson* computations fails, the whole use case computation returns the value constructor *Raise* with a value of the type *Exception*. This code example can also be written more concisely since the programming language Haskell provides syntactic sugar for sequential computations using monads [Pet18]. Listing 3.11 shows the refactored version using the *do notation*.

Listing 3.11: Parse JSON use case with *do notation*

```
data Json = ...

parseJson :: String -> (ResultOrError String Json)
parseJson s = ...
```

```
jsonUseCase2 s1 s2 = do
  j1 <- parseJson s1
  j2 <- parseJson s2
  return (j1,j2)
```

Because *ResultOrError* is also a monad, a value of this type can also be passed to the previously defined function *intComputation* (see Listing 3.7). If both of the passed values are equal to the value constructor *Return*, the product of both integers is wrapped into a monad again and returned. In case one of the passed monads has the error state, an error gets returned.

3.3.3 Maybe Monad

Introduction

The previous section introduced and described the *Exception* monad, with which error messages of any type can be wrapped (see Section 3.3.2). The function *safeHead* in the code example of the previous section shows how the introduced type can be used (see Listing 3.9). In this code example, the function returns *Raise "List must not be empty"* if the passed list does not contain any elements. Otherwise, the head element gets returned. The only thing that can go wrong is that the passed list does not contain any elements. One may now claim that returning an error state with a specific error message is unnecessary because, in the event of an error, the reason is always the same. This is where the *Maybe* monad comes into play.

In contrast to the *Exception* monad, the *Maybe* monad consists of a successful state, usually referred to as *Just*, and an error state, which contains no information about the error and is often referred to as *Nothing* [Wad90, Jon95]. The data type itself is often named as *Maybe* [Wad90, Jon95].

Implementation

An implementation of the *Maybe* monad can be achieved in a similar way to the *Exception* monad (see Listing 3.8). In the data type declaration, the only difference is that the type constructor accepts only one type argument, whereas the value constructor that indicates the error does not accept any arguments. The monad instance declaration also adapts the different constructors. Listing 3.12 shows the implementation using the common naming as described in the introduction (see Section 3.3.3). The implementation and declaration of the *Maybe* data type and the monad instance in this work equals the declaration and implementation in Haskell [The01b].

Listing 3.12: *Maybe* monad implementation

```
data Maybe a = Nothing | Just a
```

```
instance Monad Maybe where
  return = Just
  (>>=) (Just x) f = f x
  (>>=) Nothing _ = Nothing
```

Usage

All of the examples from the previous section about the usage of *Exception* monads (see Section 3.3.2) can be adopted almost effortlessly for the *Maybe* monad. For example, Listing 3.13 shows the implementation of the *safeHead* function, which now returns the type *Maybe a* instead of *ResultOrError String a*.

Listing 3.13: Safe *head* function using the *Maybe* monad

```
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x:xs) = Just x
```

In the example of the JSON use case, only the *parseJson* function needs to be changed. After changing the signature from *parseJson* to $String \rightarrow Maybe\ Json$, the *jsonUseCase* function continues to work without any changes, since the *do notation* works with any monad [Pet18]. The updated JSON use case is presented in Listing 3.14.

Listing 3.14: Parse JSON use case with *do notation*

```
data Json = ...

parseJson :: String -> Maybe Json
parseJson s = ...

jsonUseCase s1 s2 = do
  j1 <- parseJson s1
  j2 <- parseJson s2
  return (j1,j2)
```

3.3.4 List Monad

Introduction

A *List* monad is a monad instance, declared on the list type. In Haskell, for instance, the list of characters ['x', 'y', 'z'] is a simpler representation of *'x':('y':('z':[]))* [HPF99]. The list type is built into the Haskell language, but the Glasgow Haskell Compiler declares it as the unary type constructor "**data** [] a = [] | a : [a]" [The09]. In this data type declaration, *[]* represents the empty list while *:* is a right associative infix operator that adds the left argument to the first position of the right list argument [HPF99].

When the *list* monad instance is declared, the functions *unit* and *bind* must be implemented. The *list* monad's *unit* function puts the received element into a list, i.e. a list containing a single item is returned. The function *bind* is defined by the signature $[a] \to (a \to [b]) \to [b]$ and applies the passed function to each of the elements, but also flattens the result to conform with the type of the function [Wad93].

Implementation

In Haskell, the *List* monad declaration is directly defined on the built-in list type *[a]*, where *a* can be substituted by an arbitrary type. For example, the list ['x','y','z'] has the type *[Char]*. For illustration purposes, the *List* monad instance in this section is declared on a newly created list type. The custom list type declaration is implemented as in [Hug89], however the names *cons* and *nil* were already used in the LISP programming language for the same purposes [HPF99]. The source code line 1 in Listing 3.15 declares the list type named *List*, which is either the nullary value constructor *Nil* or the binary value constructor *Cons*. *Nil* represents the empty list, while *Cons* consists of an element with the type argument's type and another recursively defined *List a*. Three different lists of the type *List Char* are shown from source code lines 3 to 5 to give an example on how lists can be constructed using the new type. In each line, the subsequent inline comment shows the analog list with the built-in Haskell list type. The declaration of the *list* monad begins with line 7. As in source line 4, the *return* function also returns a list with a single element.

In the >>= operation, a *Nil* parameter is handled the same as the *Nothing* parameter of the *Maybe* type (see Listing 3.12). The function parameter is ignored (in Haskell expressed by "_") and the same value constructor that represents an empty value is returned. However, the second case to be handled is implemented differently compared to the *Maybe* monad. The reason for this is that the function *f* must be applied to every element in the *List* monad. Otherwise, the right identity and associativity monad law would be violated (see Equation 3.4 and 3.5). At line 10, >>= applies the passed function to the first element of the list and then calls itself recursively with the rest of the list. All single results are lists themselves, but to match the function signature of *bind*, the operator (<>) concatenates the single lists. The operator (<>) is defined in the semigroup class and defines an associative binary operation [The01c]. In the instance declaration for the type *List a*, this operator receives two lists and returns their concatenation. The declaration is omitted. In this section, any reference to a source code line without an explicit specification of the related listing refers to Listing 3.15.

Listing 3.15: *List* monad implementation

```
1 data List a = Nil | Cons a (List a)
2
3 l1 = Nil :: List Char -- []
4 l2 = Cons 'x' Nil -- ['x']
5 l3 = Cons 'x' (Cons 'y' (Cons 'z' Nil)) -- ['x','y','z']
6
```

```
7  instance Monad List where
8    return x = Cons x Nil
9    (>>=) Nil _ = Nil
10   (>>=) (Cons x xs) f = (<>) (f x) ((>>=) xs f)
```

Usage

Compared to the *Maybe* monad, the *List* monad holds multiple elements of the same type. A call to the function *bind* means that this function is applied to every element. Consider the functions *replicateL* and *showL*, which are replicas of the Haskell functions *replicate* and *show*. *replicateL* takes a replicate factor i and the element x to replicate and returns a *List a* containing the element x i times, while *show* creates a list of characters (*List Char*) out of a passed element. By using the monad's *bind* function, these functions can be executed sequentially (see Listing 3.16).

Listing 3.16: *List* monad usage

```
replicateL :: Int -> a -> List a
replicateL i x = ...

showL :: a -> List Char
showL x = ...

l = Cons 1 (Cons 2 (Cons 3 Nil))
l >>= (replicateL) >>= showL -- Cons '1' (Cons '1' (Cons '2' (
    Cons '2' (Cons '3' (Cons '3' Nil)))))
```

Analogously, this example can also be written with the built-in list type and the Haskell functions for *replicateL* and *showL* (see Listing 3.17). However, the output of both functions is different, since Haskell provides syntactic sugar for the type *[Char]*, which is the same as the type *String* [HPF99]. Strings can be defined by writing characters within quotation marks and can be used interchangeably with the type *[Char]*.

Listing 3.17: *[a]* monad usage

```
l = [1,2,3]
l >>= (replicate 2) >>= show -- "112233"
```

3.3.5 State Monad

Introduction

In chapter 2, the differences between the imperative and declarative programming paradigm were explained and furthermore, it was mentioned that no implicit states exist in declarative programming [Hud89]. As a consequence, state must be handled explicitly. With pure functional programming languages, state can be handled by passing it around

functions (see Section 2.3.3) [Wad90]. However, passing around the state can be tedious and error-prone [Wad93, Wad90]. The *state* monad provides an abstraction for passing around parameters [Wad90].

A key difference between the previously introduced monads and the *State* monad is that the *State* monad not just wraps a passed type a, but a function with the signature $State \rightarrow (a, State)$. This function takes a state and returns a tuple, which contains a value of type a (e.g. a computed pseudo-random number) and a state of the type *State* (e.g. a pseudo-random number generator) [Wad93]. In summary, the *State* monad offers a way to perform state-based computations.

Implementation

The naming and implementation of the *State* monad in this section is heavily inspired by [Wad93]. However, the newly introduced *State* monad type uses a second type constructor argument s for specifying the type of the state. In Listing 3.18, the first line declares a new type called *State*. The unary value constructor S (short for State) takes a function of the type $s \rightarrow (a, s)$, which can be accessed via the accessor *execState*. After the type declaration, the monad instance is declared for *State s*. Since the monadic value wraps a function, the *return* operation must also return the same monadic value that wraps a function with an identical signature. In short, *return* wraps a function that takes a state of type s and puts it into a tuple with the value passed to *return* [Wad93].

The *bind* function receives a monad *State s a*, which wraps a function m that takes a state of type s and produces a tuple of the type *(a,s)* and a function k, which takes a value of type a and produces a monad of the type *State s b*. At first, the lambda expression with the formal parameter *s0* is wrapped inside the value constructor S. In this lambda expression, the function m is first applied to the formal parameter, which results in a value a and a new state *s1*. To receive a type *(b,s)*, which must be the result of the lambda expression, the state function of the application of k to a is extracted and then applied to the state *s1*. Finally, the lambda expression has the type $s \rightarrow (b, s)$. [Wad93]

Furthermore, two helper functions *fetch* and *assign* are defined. *fetch* is a wrapped function that puts a received state s into a 2-tuple containing s two times. *assign*, on the other hand, receives a state s and returns a wrapped function that puts s in its returned 2-tuple. These functions can be used to read and update a state. The functionality of the *state* monad and the helper functions is exemplified in the following section.

Listing 3.18: *State* monad implementation

```
newtype State s a = S { execState :: s -> (a,s) }

instance Monad (State s) where
  return x = S $ \s0 -> (x,s0)
  (>>=) (S m) k = S $  \s0 ->
    let
```

```
      (a,s1) = m s0
    in execState (k a) s1

fetch :: State s s
fetch = S (\s -> (s, s))

assign :: s -> State s ()
assign s = S (\_ -> ((),s))
```

Usage

To finally combine everything about the *State* monad into an example, a simple game simulation is used in this section. The example's game state consists of an integer score and the *StdGen* pseudo-random number generator from the Haskell package *System.Random* (see Listing 3.19 line 1-4). Each time a pseudo-random number is generated, the generator returns the generated number and a new instance of a generator [The01d]. The next time a number is generated, the new instance must be used to generate a new pseudo-random number. Otherwise, the same number as before will be generated. In this game, each round a dice is rolled and the result is added to the score.

The function *rollDice* (see Listing 3.19 line 8-14) is responsible for generating a new pseudo-random number and updating the generator. First, the current stated is fetched using the previously defined *fetch* function. Then the generator is extracted and an integer number between [1,6] is generated. Following the generation, the pseudo-number generator is updated in the game state using the function *assign*. At the end of the function, the result of the generation is wrapped and returned.

The second function *updatePoints* (see Listing 3.19 line 16-21) is used to update the points of the game state accordingly. This is done by fetching the state followed by a state update using *assign* and the passed integer parameter.

For making a turn in the game, both functions can be executed sequentially using the monad's *bind* operation. However, to execute the whole function composition, the function of the *State* monad must be applied to an initial state (defined in Listing 3.19 line *6*). *run* (see Listing 3.19 line 29) can be used to extract and apply the function for a turn to the initial state. The result is a 2-tuple where the second value is the updated state after a turn.

In all functions except *run*, the explicit passing of the game state is hidden in the *State* monad *State*.

Listing 3.19: *State* monad example

```
1 data GameState = GameState
2   { points :: Int
3   , generator :: StdGen
4   } deriving Show
```

```
5
6  initGameState = GameState 0 (mkStdGen 0)
7
8  rollDice :: State GameState Int
9  rollDice = do
10   state <- fetch
11   let gen = generator state
12   let (r, gen') = randomR (1,6) gen
13   assign (state { generator = gen' } )
14   return r
15
16 updatePoints :: Int -> State GameState ()
17 updatePoints c = do
18   state <- fetch
19   let currentPoints = points state
20   assign (state { points = currentPoints + c })
21   return ()
22
23 turn :: State GameState ()
24 turn = rollDice >>= updatePoints
25
26 run = execState turn initGameState
```

3.3.6 Reader Monad

The state *Reader* monad is a subset of the *State* monad [Wad93] and is used to pass around an enclosing environment [Jon95]. Unlike in the *State* monad, the passed environment (or state) is only read and not updated. This behavior is also reflected in the data type and monad declaration. The value constructor R wraps a function which does not return a tuple with a state as in the *State* monad, but only returns a type a. For example, the environment can consist of a database connection or other dependencies that are required in computations. Listing 3.20 shows the implementation of the *Reader* monad. However, a detailed explanation is omitted since the implementation is largely covered in the section on the *State* monad (see Section 3.3.5).

The function *return* receives a parameter x and wraps a function that only returns the passed parameter x. The formal parameter, which represents the environment or the state, is ignored. The main difference in *bind* is that the application of m to the environment e does just return a single value a.

Listing 3.20: *Reader* monad implementation

```
newtype Reader s a = R { execState :: s -> a }

instance Monad (Reader s) where
```

```
return x = R $ \e -> x
(>>=) (R m) k = R $  \e ->
  let
    a = m e
  in execState (k a) e
```

CHAPTER 4

Software Complexity

4.1 Introduction

Already in the year 1985, the maintenance stage of the software development life cycle was considered the most expensive activity [YC85] and still, about 30 years later, "it is proven that the cost, time, and effort required for maintenance is very high" [CMSR17, p. 767]. The use of quality metrics during the development life cycle can positively impact the maintainability of the overall system [YC85].

A metric is defined by the IEEE standard glossary of software engineering terminology as a "quantitative measure of the degree to which a system, component, or process possesses a given attribute" [IEE90, p. 47-48]. In the field of software engineering, the more specific term *software quality metric* is defined similarly as follows: "A function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software possesses a given attribute that affects its quality" [IEE98, p. 3].

Software complexity is tightly coupled to the maintainability of a software system [TSZ09]. Complexity in terms of software engineering is defined by "the degree to which a system or component has a design or implementation that is difficult to understand and verify" [IEE90, p. 18]. It follows that the software complexity metric is a quantitative measure of the degree of how difficult software can be understood. Harrison et al. also relate software complexity to the measure of how well software systems can be understood and how easy they are to work with [HMKD82]. Furthermore, they also claim that the understandability, modifiability and testability of the software are subcategories of maintainability that are impacted by software complexity [HMKD82]. Therefore, reducing software complexity can result in better software maintainability.

In the year 1990, "more than two hundred software complexity measures" [Zus91, p. 2] have been introduced. Well-known metrics include the *Lines of Code* metric, *Cyclomatic*

Complexity, *Halstead Complexity Measures*, *C&K Method* and *MOOD Method*, while the latter two are complexity metric sets used for object-oriented programming languages [Zus91, TSZ09]. The quantitative analysis of this work uses the *Lines of Code* metric, *Cyclomatic Complexity* and *Halstead Difficulty*. To give a better overview, all of the above-mentioned metrics are discussed in the following section (see Section 4.2).

4.2 Software Complexity Metrics

4.2.1 Lines Of Code (LOC)

The *Lines Of Code* metric was one of the most discussed metrics [Zus91] and is considered as "one of the most widely used sizing metrics in industry" [BM14, p. 1]. Initially, this metric was used for measuring the programming progress [TSZ09]. However, the metric is also viewed as a complexity metric because an increasing number of *LOC* of a software system also increases the complexity of this system [BM14].

Various definitions exist: For example, Conte et al. define the *LOC* metric in their book *Software Engineering Metrics and Models* as follows:

> "A line of code is any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program headers, declarations, and executable and nonexecutable statements." [CDS86, p. 35]

The *IEEE Standard for Software Productivity Metrics* [IEE93] divides this metric into logical source statements (LSS) and physical source statements (PSS), where LSS counts executable, data declaration, compiler directive and comment source statements while PSS counts the lines of code or the number of nonblank lines. PSS is further divided into Non-Comment Lines of Code (NCLOC) and comment lines [IEE93]. *NCLOC* is defined by "lines of software that contain either executable, data declaration, or compiler directive source statements" [IEE93, p. 7]. In contrast, comment lines are defined as "lines that contain only comment source statements" [IEE93, p. 7].

In this thesis, the NCLOC metric is used for the quantitative analysis. If the NCLOC metric is now calculated for the source code of Listing 4.1, the result is $NCLOC = 8$.

Listing 4.1: Example for the NCLOC metric

```
public class NameService {

    public NameService() {

    }

    /**
```

```
     * Builds a {@link Name} from a String
     *
     * @param nameString The string containing the name. The {
        @code nameString} must have the following format: "
        firstName lastName"
     * @return Returns a {@link Name} object created from the
        passed string {@code nameString}
     */
    public Name buildName(String nameString) {
        String[] splitString = nameString.split("␣");
        return new Name(splitString[0], splitString[1]);
    }
}
```

4.2.2 Cyclomatic Complexity

This section about the *Cyclomatic Complexity* measure is based on the fundamental paper where it was originally defined: *A Complexity Measure* [McC76].

In 1976, Thomas J. McCabe first introduced the complexity metric *Cyclomatic Complexity*. This complexity measure is based on the number of paths in the graphical representation of a program (control-flow graph) [McC76]. To overcome the problem with infinite paths, this approach considers just "basic paths" [McC76, p. 308], with which theoretically all paths can be created by combination. In particular, the following formula is used for calculating the metric (see Equation 4.1). The equation consists of the variables e, n and p, where e stands for the number of edges, n is the number of vertices and p reflects the number of connected components.

$$v(G) = e - n + 2p \tag{4.1}$$

McCabe defines six properties of the cyclomatic complexity metric. The first property states that the *Cyclomatic Complexity* is always one or greater than one ($v(G) \geq 1$). A cyclomatic complexity of *1* corresponds to a sequential control-flow graph without branches or one path, as property four says. Even a sequential control-flow graph with an arbitrary number of nodes has a *Cyclomatic Complexity* of one. As already described above, the second property states that the *Cyclomatic Complexity* "$v(G)$ is the maximum number of linearly independent paths in G; it is the size of a basis set" [McC76, p. 309]. The third property defines that adding or deleting function statements does not affect the cyclomatic complexity. Property five indicates that adding a new edge to the control-flow graph adds *1* to previous cyclomatic complexity. Property six contains that the *Cyclomatic Complexity* can only be influenced by the structure of decisions of the control-flow graph.

In the following, two examples are given for the calculation of the *Cyclomatic Complexity*. Both examples are based on the examples in the paper [TSZ09] and consist of two different implementations of a method, which calculates the highest of three different numbers (see Listing 4.2 and 4.3). For each implementation, the corresponding control-flow graph is shown (see Figure 4.1a and 4.1b).

Listing 4.2: *max* implementation 1

```
int max(int x, y, z) {
  if (x > y) {
    if (x > z) {
      return x;
    } else {
      return z;
    }
  } else {
    if (y > z) {
      return y;
    } else {
      return z;
    }
  }
}
```

Listing 4.3: *max* implementation 2

```
int max(int x, int y, int z) {
  if (x > y && y > z) {
    return x;
  } else if (y > z && z > x) {
    return y;
  } else {
    return z;
  }
}
```

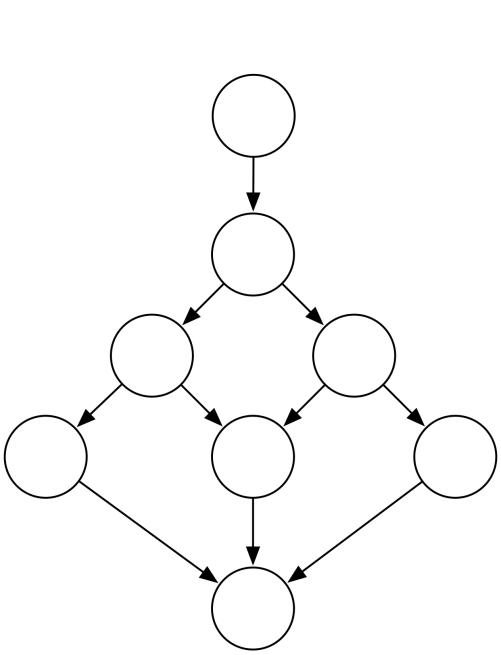
Based on the control-flow graph, the *Cyclomatic Complexity* can be calculated using the previously presented formula (see Equation 4.1). The control-flow graph 4.1a, which represents the program in Listing 4.2 has eight nodes ($n = 8$), ten edges ($e = 10$) and one consists of a single component ($p = 1$). Using the formula 4.1, this results in a cyclomatic complexity of $v(G) = 10 - 8 + 2 = 4$.

On the other hand, the control-flow graph 4.1b of the program 4.3 has nine nodes ($n = 9$), twelve edges ($e = 12$) and also one component ($p = 1$). This results in a cyclomatic complexity of $v(G) = 12 - 9 + 2 = 5$, which differs by *1* from the other implementation 4.1a.

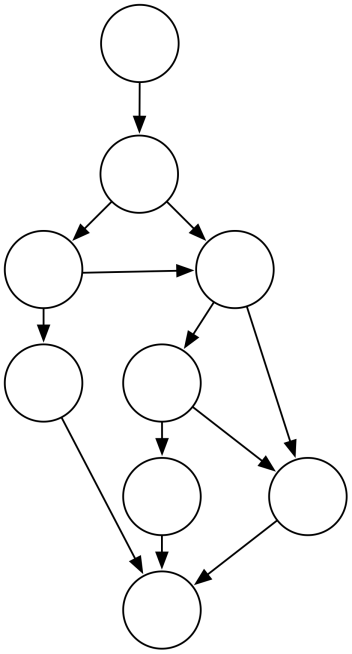
However, both results are considered as well structured [McC76]. In the paper [McC76], *10* has been set as the upper bound for the *Cyclomatic Complexity* of a module. Summarized it can be said that the *Cyclomatic Complexity* describes the complexity of the structure of a program based on the underlying control-flow graph.

4.2.3 Halstead Complexity Measures

In 1977, about one year after Thomas J. McCabe introduced the *Cyclomatic Complexity* (see Section 4.2.2), Maurice H. Halstead introduced the *Halstead complexity measures* in



(a) Control-flow graph of Listing 4.2



(b) Control-flow graph of Listing 4.3

Figure 4.1: Control-flow graphs

his work *Elements of Software Science* [Hal77]. If not stated otherwise, his book [Hal77] is used as the main reference for all of the following formulas and explanations.

All of the measures are based on the number of operators and operands of the program. Before explaining the different metrics in detail, a basic set of required metrics must be defined. η_1 is the number of unique or distinct operators and η_2 is the number of unique or distinct operands. The variables N_1 and N_2 are the corresponding total usage numbers of η_1 and η_2.

Based on the set of basic metrics, the particular measures of the *Halstead complexity measures* can be defined.

Program Length

The program length can be calculated by adding N_1 and N_2: $N = N_1 + N_2$. Equation 4.2 can be used to calculate an approximation of the program length.

$$\hat{N} = \eta_1 * \log_2(\eta_1) + \eta_2 * \log_2(\eta_2) \tag{4.2}$$

Program Volume

The program volume is a language-independent sizing metric. With this metric, it is possible to measure the size changes of an algorithm that has been translated from one programming language to another in a quantitative way. For calculating the program volume, the program length and the total vocabulary η ($\eta = \eta_1 + \eta_2$) is needed. The metric can then be calculated as shown in Equation 4.3.

$$V = N * \log_2 \eta \tag{4.3}$$

Program Level

Program level or the level of an implementation is a metric, which "emphasizes that growth in volume leads to a lower level of the program" [Abr10, p. 156]. On the other hand, a lower program volume (see Program Volume) leads to a higher program level, with the maximum being *1*. This metric is defined by the division of the potential Volume V^* with the program volume V: $V = \frac{V^*}{V}$. The potential volume V^* is the volume of a program with "the most succinct form in which an algorithm could ever be expressed" [Hal77, p. 20] (for a detailed explanation see [Hal77]). In the absence of V^*, however, the approximation formula shown in Equation 4.4 may also be used interchangeably in many cases.

$$\hat{L} = \frac{2}{\eta_1} * \frac{\eta_2}{N_2} \tag{4.4}$$

In addition to the program level, also the language level exits. However, an explanation of this metric is omitted for the sake of brevity.

Intelligence Content

Based on the program level and the program volume, the intelligence content can be defined as follows: $I = \hat{L} * V$. This metric answers the question "of how much is said in a program" [Hal77, p. 32].

Program Difficulty

Halstead defines the difficulty of understanding a program (D) as the division of *1* by the program level L (see Program Level): $D = \frac{1}{L}$. Since *1* is the maximum value for the program level and "values close to 1 are considered to be well written" [Abr10, p. 156] it is implied that *1* is the minimum value for D and therefore the optimal value.

Programming Effort

The idea of the programming effort is restricted to "the mental activity required to reduce a preconceived algorithm to an actual implementation in a language in which the

implementor (writer) is fluent" [Hal77, p. 46] and can therefore be calculated using the previously defined metrics program volume (see section 4.2.3) and program level (see Program Level) or program difficulty (see Program Difficulty). Equation 4.5 shows the calculation.

$$E = \frac{V}{L} = V * D \tag{4.5}$$

The estimated programming time can now be calculated by the division of the programming effort and the *Stroud number* S (see [Hal77] for a detailed explanation): $\hat{T} = \frac{E}{S}$.

4.2.4 C&K Method

Shyam R. Chidamber and Chris F. Kemerer introduced six new metrics in their paper *A Metrics Suite for Object Oriented Design* in 1994 [CK94], which later became known by the name *C&K metrics method* [TSZ09]. Compared to the previously described metrics *NCLOC* (see Section 4.2.1), *Cyclomatic Complexity* (see Section 4.2.2) and the *Halstead Complexity Measures* (see Section 4.2.3), the *C&K metrics method* consists of metrics designed for object-oriented systems. These metrics statically "measure the complexity in the design of classes" [CK94, p. 477].

The general idea of the *C&K metrics method* is to support the software development process. However, considered in more detail, the usage of the metrics of the *C&K metrics method* can, for instance, help to ensure consistency between the architecture and the structure of the application and to find areas with a high complexity [CK94]. The six metrics are described below.

C&K Method Metrics

Weighted Methods Per Class (WMC) The WMC metric describes the complexity of a class [CK94]. To calculate this metric, all complexity values $(c_1, ..., c_n)$ of the methods $(m_1, ..., m_n)$ of a class must be summed up. Equation 4.6 shows the mathematical description of the calculation. The paper [CK94] does not further specify how the complexity is calculated "in order to allow for the most general application of this metric" [CK94, p. 482].

$$WMC = \sum_{i=1}^{n} c_i \tag{4.6}$$

Consider the *Name* class declaration from the previous chapter about the object-oriented programming paradigm (see Listing 2.3). The WMC of this class is *6*, assuming that the *Cyclomatic Complexity* (see Section 4.2.2) is used for calculating the complexity and the constructor is counted as a method. Since there are no branches in any of the methods of

the class, each method has a complexity value of *1*. In this example, the WMC is equal to the number of methods (n).

This metric can be used to gain insight into the complexity of an application at class-level. The WMC metric positively correlates with the programming effort and has a negative correlation with the maintainability of the class because a high WMC means that the class has many complex methods and is therefore less maintainable [CK94].

Depth of Inheritance Tree (DIT) As the name already indicates, this metric directly corresponds to the inheritance depth of a class. The authors of [CK94] claim that the number of methods in a class harms the complexity of the class since the more methods are available, the more complex it will be to predict the behavior of the class [CK94]. Furthermore, the design complexity is affected by the number of methods[CK94]. The DIT metric measures the depth of the inheritance tree because in an inheritance tree, "the deeper a class is in the hierarchy, the greater the number of methods it is likely to inherit" [CK94, p. 483].

Considering the *Name* class (see Listing 2.3) from a previous chapter (see Chapter 2) again, the DIT is *1* because it does not inherit from any other class. However, when analyzing the class *TitledName* (see Listing 2.5), the DIT increases by one ($DIT = 2$) because this class inherits from the *Name* class (see Listing 2.3).

Number of Children (NOC) NOC refers to the number of direct subclasses of a class [CK94]. The viewpoints for this metric, taken from [CK94], are explained in the following: First, the more classes (children) inherit from a class (parent), the higher is the reusability. The second viewpoint states that a high NOC could indicate improper abstraction or incorrect use of subclassing. The third and final viewpoint states that the NOC metric can be used as an indicator of the need to test classes (e.g. a high NOC indicates prominent classes).

Analyzing the *Name* class (see Listing 2.3) leads to a NOC of 1 because *Name* has one subclass: *TitledName* (see Listing 2.5). Creating another subclass of the *Name* class would result in a NOC of 2.

Coupling between object classes (CBO) The CBO metric is defined as: "CBO for a class is a count of the number of other classes to which it is coupled" [CK94, p. 486]. A coupling between two classes exists, if instance variables and/or methods of one of the classes (see Section 2.2.2) are used in the other class [CK94].

A high CBO negatively impacts the modularity and maintainability of a system and also negatively affects the required test effort [CK94].

The *NameService* class (see Listing 4.1) from the LOC section (see Section 4.2.1) has a $CBO = 1$ because it uses the *Name* class constructor (see Listing 2.3; the constructor is counted as a method).

Response For a Class (RFC) The RFC metric is equal to the number of elements (cardinality) in a response set (RS) of a class: $|RS|$ [CK94]. The RS of a class C is a set containing all n methods of C $(m_1, ..., m_n)$ combined with all methods that are called in the methods m_1 to m_n [CK94]. A large RFC value negatively impacts the overall and testing complexity of a class.

An RFC calculation for the *NameService* class (see Listing 4.1) results in *4*. This is due to the two methods *NameService* and *buildName* and two called methods *split* and *Name* in *buildName* (a constructor is considered a method).

Lack of Cohesion in Methods (LCOM) The last of the six metrics defined in the *C&K Method Metrics* is called Lack of Cohesion in Methods (LCOM) and is defined as the following equation (see Equation 4.7) shows. All of the definitions regarding LCOM are directly taken from [CK94].

$$LOCM = \begin{cases} |P| - |Q|, & \text{if } |P| > |Q| \\ 0, & \text{otherwise} \end{cases} \tag{4.7}$$

A class C consists of instance variables and n methods $(m_1, ..., m_n)$. The set $\{I_i\}$ is a set that contains all of the used instance variables in the method m_i. For each of the n methods, one set exists: $\{I_1\}, ..., \{I_n\}$. P is then defined as $P = \{(I_i, I_j) | I_i \cap I_j = 0\}$, while Q is defined as $Q = \{(I_i, I_j) | I_i \cap I_j \neq 0\}$.

In other words, "the LCOM value provides a measure of the relative disparate nature of methods in the class" [CK94, p. 489]. A high LCOM indicates a hight cohesion between the methods of a class, which in turn "promotes encapsulation" [CK94, p. 489]. A low cohesion negatively impacts complexity and could also indicate that the class should be redesigned [CK94].

Consider the following example, taken from [CK94], where $\{I_1\} = \{a, b, c, d, e\}$, $\{I_2\} = \{a, b, e\}$ and $\{I_3\} = \{x, y, z\}$. In this example, a class consists of three methods (m_1, m_2, m_3), where method m_1 uses the instance variables a to e, method m_2 uses a, b and e and method m_3 uses x, y and z. The following set intersections can be made: $\{I_1\} \cap \{I_2\} = \{a, b, e\}$, $\{I_1\} \cap \{I_3\} = \{\}$ and $\{I_2\} \cap \{I_3\} = \{\}$. The LCOM metric is calculated by subtracting the number of non-empty intersections from the number of empty intersections: $LCOM = 2 - 1 = 1$.

4.2.5 MOOD

Similar to the *C&K Method Metrics*, in 1994, Fernando Brito e Abreu and Rogério Carapuça proposed the *MOOD* (Metrics for Object Oriented Design) metrics set, originally consisting of eight metrics for object-oriented systems [AC94]. They claim "that the MOOD metrics (except the Reuse Factor) can be combined to obtain a generic OO software system complexity metric" [AC94, p. 8].

Seven criteria were defined, used as a starting point for the proposed metrics (see [AC94] for a detailed description and the motivation behind them). The eight metrics are Method Inheritance Factor (MIF), Attribute Inheritance Factor (AIF), Coupling Factor (COF), Clustering Factor (CLF), Polymorphism Factor (PF), Method Hiding Factor (MHF), Attribute Hiding Factor (AHF) and Reuse Factor (RF). However, the RF will be skipped due to the above-mentioned fact and also because the RF is not included in the set for obtaining a generic complexity metric for object-oriented software systems. Therefore, the following section is structured in object-oriented concepts and within each, the related metrics are briefly explained. The descriptions of the single metrics and all the formulas are based on the original paper [AC94].

MOOD Metrics

Encapsulation and Information Hiding Related Both metrics, MHF and AHF, are related to the object-oriented programming concept of encapsulation and information hiding (see Section 2.2.2). $M_d(C_i)$ is a function which returns the number of all methods, the number of visible methods ($M_v(C_i)$) plus the number of hidden methods ($M_h(C_i)$), for a specific class C_i: $M_d(C_i) = M_v(C_i) + M_h(C_i)$.

The calculated value for the MHF metric is the proportion between the sum of all $M_h(C_i)$ and the sum of $M_d(C_i)$ over all classes (TC $\widehat{=}$ total classes) (see Equation 4.8).

$$MHF = \frac{\sum_{i=1}^{TC} M_h(C_i)}{\sum_{i=1}^{TC} M_d(C_i)} \tag{4.8}$$

AHF is calculated similarly to the MHF metric, except that attributes (instance variables - see Section 2.2.2) are considered instead of methods. Equation 4.9 shows the formula for the AHF metric calculation, where $A_h(C_i)$ and $A_d(C_i)$ are calculated analogously to $M_h(C_i)$ and $M_d(C_i)$.

$$AHF = \frac{\sum_{i=1}^{TC} A_h(C_i)}{\sum_{i=1}^{TC} A_d(C_i)} \tag{4.9}$$

When the results of AHF and MHF are evaluated, it should be ensured that the value does not fall below a recommended lower limit.

Inheritance Related MOOD metrics related to the object-oriented concept of inheritance (see Section 2.2.2) are the MIF and AIF metric. Before presenting the formula for calculating MIF and AIF, the basic class metrics required are briefly explained. $M_d(C_i)$ presents the number of methods defined in a specific class i, $M_n(C_i)$ is the number of newly added methods in a specific class i (non-overwritten methods), $M_n(C_i)$ is the number of overwritten methods in a specific class i and $DC(C_i)$ defines the number of descendants of a specific class i.

The Total number of Methods Inherited (TM_i) is defined by Equation 4.10 and the Total number of Methods Available (TM_a) can be calculated using the formula shown in Equation 4.11.

$$TM_i = \sum_{k=1}^{TC}[[M_d(C_k) - M_o(C_k)] * DC(C_k) - M_o(C_k)] \tag{4.10}$$

$$TM_a = \sum_{k=1}^{TC}[[M_n(C_k) * [1 + DC(C_k)]] \tag{4.11}$$

MIF is then defined as the division of TM_i and TM_a (see Equation 4.12).

$$MIF = \frac{TM_i}{TM_a} \tag{4.12}$$

AIF can be calculated in a similar way, but only attributes are considered instead of methods.

$$AIF = \frac{TA_i}{TA_a} \tag{4.13}$$

When the results are evaluated, it is checked whether the MIF or AIF value is within a recommended interval. A specific recommendation could look like the following example: "Keep the Method Inheritance Factor between 0.25 and 0.37" [AC94, p. 7].

Coupling and Clustering Related As with the *C&K Method Metrics*, there is also a metric in the *MOOD* metric set that describes the coupling (see Paragraph 4.2.4) of classes.

This metric (COF) can be calculated using the formula shown in Equation 4.14, where *is_client* is a function for two classes C_i and C_j that either returns 1, if C_i contains at least one reference to a method or attribute of the class C_j, or 0 otherwise. This value should be kept below a recommended upper limit.

$$COF = \frac{\sum_{i=1}^{TC}[\sum_{j=1}^{TC} is\_client(C_i, C_j)]}{TC^2 - TC} \tag{4.14}$$

Coupling and inheritance between classes form "a set of disjoined graphs where nodes represent classes and edges represent the relations" [AC94, p. 7] between them. Each graph in the set is called a *cluster* that can be reused. The coupling factor is then defined by the proportion between the total number of class clusters (TCC) and the total number of classes (TC) (see Equation 4.15). Unlike the COF, this metric should be kept above a recommended lower limit.

$$CLF = \frac{TCC}{TC} \tag{4.15}$$

Polymorphism Related The last one of the explained *MOOD* metrics, the Polymorphism Factor, relates to the concept of *polymorphism* (see Section 2.2.2). This metric is defined as the division of "the total number of possible different polymorphic situations" [AC94, p. 6] and the "maximum number of possible different polymorphic situations" [AC94, p. 6] in the whole system (see Equation 4.16). As with the first two metrics, this metric should also be within a recommended interval.

$$PF = \frac{\sum_{i=1}^{TC}[\sum_{j=1}^{DC(C_i)} M_o(C_j)]}{\sum_{i=1}^{TC}[M_d(C_i) * DC(C_i)]} \tag{4.16}$$

4.3 Summary

This chapter introduced software metrics for measuring the complexity of software systems. Over the years, many different complexity metrics have been introduced. For example, in the year 1990, the number of metrics already exceeded 200 [Zus91]. In this chapter, five commonly used complexity metrics were selected and explained in greater detail.

The first and trivial complexity metric is the *Lines Of Code* metric. This metric counts the number of source code lines. However, there are different definitions of what is counted and what is excluded from the count (see Section 4.2.1). The second metric explained is called *Cyclomatic Complexity* and is based on the number of branches (nodes and edges) of the control-flow graph of a program (see Section 4.2.2). Following the *Cyclomatic Complexity*, the *Halstead Complexity Measures*, a set of statically calculable complexity metrics, are explained (see Section 4.2.3). This chapter ends with two different metric suits for object-oriented programming systems, the *C&K Method* (see Section 4.2.4) and the *Metrics for Object Oriented Design (MOOD)* (see Section 4.2.5). Both metric suites measure metrics related to object-oriented design.

CHAPTER 5

Evaluation

This work's main contribution is to provide an extensive overview of the practical impact of a monadic programming style in a Java-based application software on software complexity. In particular, command and query methods of the data access layer are rewritten to a monadic version of the same method. The rewriting of these methods leads to further rewritings in the project. The analysis itself is carried out on an open-source Android application. The used Android application is explained in detail in a following section (see Section 5.1).

This chapter also defines a set of rules for the rewriting process. The rules determine what is allowed during rewriting and determine the conditions under which method signatures change (see Section 5.5).

In order to minimize the number of programming errors, a code review was conducted by a project independent software engineer and software tests were run to check whether the results of the tests remained unchanged to the time before the rewriting. Details about the verification can be found in Section 5.7.

The detailed description of the quantitative evaluation results is then presented in the following chapter (see Chapter 6).

5.1 Evaluation Project

Requirements for the selection of the project are that the application software is written in the object-oriented programming language Java, that a data access layer must exist and that no monadic structures have been used in this data access layer yet.

Due to the author's in-depth experience with software engineering for the Android platform, the search for an appropriate project focused on state-of-the-art application software for the Android operating system. At first, the search turned out to be

difficult because Android developers often use reactive libraries like RxJava to ease the development process. However, the provided data structures from RxJava can be considered as monadic. Therefore, all Android applications which were using RxJava in the data access layer could not be used for the practical part of this thesis. The search was then extended to Android applications, which have been under development for a longer time.

The open-source Android application OpenKeychain[1] met all of these requirements and was therefore selected. The following description text originates from the official website: "Modern encryption is based on digital "keys". OpenKeychain stores and manages your keys, and those of the people you communicate with, on your Android smartphone. It also helps you find others' keys online, and exchange keys. But its most frequent use is in using those keys to encrypt and decrypt messages." [Schnd].

The source code was cloned from the corresponding GitHub page with the Git *HEAD* pointing to the commit with the *short* SHA-1 checksum *61892a657* in the *master* branch.

At this point, the *java* package of the main module *OpenKeychain* consisted of 56394 *NCLOC* (measured with MetricsReloaded; see Section 5.2.3) and all changes that were made relate to this state.

5.2 Relevant Tools

5.2.1 Git

The project was cloned from the GitHub website to a local computer using Git[2] with version 2.21.0 and the changes to the source code were then applied in a separate branch.

5.2.2 Integrated Development Environment (IDE)

Android Studio 4.0 with the build version #AI-193.6911.18.40.6514223 was used as an IDE. After opening the cloned project in the IDE, the project was then set up according to the *README.md* file in the root directory of the project. Android Studio is based on the IDE IntelliJ IDEA from the company JetBrains.

5.2.3 MetricsReloaded

This work used the IntelliJ IDEA plugin *MetricsReloaded* (version 1.9) to measure the complexity metrics used in this work. The choice for this tool is based on public availability of the source code[3], the high number of downloads (127165 downloads on August 7, 2020 [Jetnd]) and that it was also used for other academic publications (e.g. [TBMP18]). Furthermore, this tool is able to measure all the required metrics.

[1] https://www.openkeychain.org/
[2] https://git-scm.com/
[3] https://github.com/BasLeijdekkers/MetricsReloaded/

5.2.4 Vavr

The software library *Vavr*[4] "is a functional library for Java 8+ that provides persistent data types and functional control structures" [DW20]. During the rewriting process, the monadic container types *Option* and *Try* were used from the library. The version 0.10.3 was used and integrated via Gradle[5].

5.2.5 JaCoCo

JaCoCo[6] is a well-known and free tool for measuring code coverage of Java programs. Good personal experience with this tool and its widespread use were reasons for using this tool as part of this work. The tool was integrated into the project with version 0.8.5.

5.2.6 palantir-java-format

To ensure a uniform code format style when measuring the *NCLOC* metric, the source code was reformatted before each of the measurements (before and after rewriting) of the *NCLOC* metric. In particular, the open-source code formatter *palantir-java-format*[7] was used.

5.3 Identifying Query and Command Methods

Based on the author's experience, a monadic programming style in query and command methods of the data access layer can positively impact the quality of the source code. Therefore, this work analyzes the impact of a monadic programming style of query and command methods in the data access layer on software complexity. To get the best possible and meaningful results, and to stay within the scope of this work, the starting point for the rewriting process is chosen depending on how often a class is used in other classes. Based on the count of dependent classes, the class *KeyRepository* of the package *org.sufficientlysecure.keychain.daos* was selected as a starting point for the evaluation. According to the MetricsReloaded plugin (see Section 5.2.3), the class has 58 dependents and thus the majority of all classes in this package. This class was then rewritten in compliance with the rules defined in Section 5.5.

Reasons why only a specific part and not the entire source code is rewritten from scratch, include that otherwise, the results would no longer derive from the monadic programming style of query and command methods from the data access layer. However, based on the results of this work, a rewrite of further parts and the whole project would be desirable for future work.

[4] https://www.vavr.io/
[5] https://gradle.org/
[6] https://www.eclemma.org/jacoco/
[7] https://github.com/palantir/palantir-java-format

5.4 Complexity Measurement

In Chapter 4, different methods for measuring software complexity were presented. In particular, five different metrics or metric sets were explained, which are considered as well known in the field of software engineering. Two of the five metrics were developed specifically for object-oriented programming languages, namely the *C&K metrics method* (see Section 4.2.4) and the *MOOD* metrics (see Section 4.2.5).

The quantitative evaluation of this work uses the *NCLOC* metric (see Section 4.2.1), *Cyclomatic Complexity* (see Section 4.2.2) and *Halstead Complexity Measures* (see Section 4.2.3). However, out of all the *Halstead Complexity Measures*, only the *Halstead Difficulty* (see Section 4.2.3) is measured since this metric directly relates to the complexity of a program (e.g. the difficulty of understanding a program).

The reason for excluding the object-oriented related complexity metrics is that rewriting does not impact any factors related to object-oriented aspects. Only method signatures and the usage of specific methods are affected by the rewriting process.

All three metrics were measured at the method level, but only the *Cyclomatic Complexity* and the *Halstead Difficulty* were additionally evaluated at the method level. The *NCLOC* metric is evaluated over the sum of all results measured at the method level. One reason for this is, for example, that an increase of the *NCLOC* metric does not necessarily indicate a deterioration in complexity.

5.5 Rewrite Rules

When source code is written, the resulting software design and quality of the source code depends, for instance, on the author's previous experience and knowledge. Moreover, the experience of the author of this work has shown that problems can usually be solved in many different ways, where some may be considered unacceptable by static code checkers and/or code reviewers.

The idea for the set of rules for the rewriting process came up at an early stage of this work while the author made his first attempts at rewriting the source code. During this attempts, it was not always clear how to use the monadic methods and how to proceed in different situations. Furthermore, software complexity must not be influenced by factors other than the monadic programming style. Guidance and instructions were required on how to proceed consistently in those situations.

Antony Tang, who wrote the paper *Software Designers, Are You Biased?*, also claims that "there are enough examples to demonstrate that biases do exist and they may be more common than we think" [Tan11, p. 7]. In order to make the quantitative work as replicable as possible and to minimize the own bias for the rewriting part, an extensive set of rules is defined. These rules determine how to proceed in certain situations, restrict the freedom of choice during the rewriting process and therefore reduce the bias and improve

the reproducibility of the results. The rules also ensure that the software complexity is mainly influenced by the monadic programming style.

5.5.1 *Source* methods

Before the actual rewriting can start, query and command methods in the data access layer must be selected. Those methods are classified and in this context referred to as *source* methods, because they provide the basis for the rewriting of the source code.

For the selection, only non-abstract query and command methods with arbitrary visibility modifiers in the data access layer are considered. The actual change to a monadic version of the method depends on the method's signature. Listing 5.1, 5.2 and 5.3 define the three relevant method types. In those listings, T stands for an arbitrary type where instances of this type never hold the value *null*. The question mark indicates that instances of type T are allowed to be *null*.

Listing 5.1: *Source* method type 1

```
T? m(..) {
    //omitted
}
```

Listing 5.2: *Source* method type 2

```
T m(..) throws E {
  //omitted
}
```

Listing 5.3: *Source* method type 3

```
T? m(..) throws E {
  //omitted
}
```

Under these conditions, Listing 5.1 represents a method that takes an arbitrary amount of parameters and returns an object of type T, which can be *null* under certain circumstances. Methods that match this type get rewritten to methods that return the type *Option*<T>. The monadic type *Option* is provided by the Vavr library (see Section 5.2.4) and corresponds to the *Maybe* monad (see Section 3.3.3). After changing the return type, at each exit point of method, the returned value is wrapped in an instance of the class *Option*. If *null* gets returned, *null* gets exchanged with *Option.none()*, which represents the *Nothing* case (see Section 3.3.3). Non-*null* values can be brought into the *Option* context by passing the value to the static method *some*. If it is not certain whether a method at an exit point returns *null* or the value to be returned is *null*, the static method *of* from the class *Option* should be used. For the practical part of this work, the analysis of whether a method returns a nullable value or not is done manually. However, this task can also be supported by static analysis tools.

Methods classified as the second *source* method type (see Listing 5.2) are methods that take an arbitrary amount of parameters and return an object of type T where instances are never equal to *null*. Furthermore, methods of this type also throw one or more exceptions E, which are subtypes of the Java class *Throwable*. These methods are rewritten as follows: First, the return type T is changed to the type *Try*<*T*>, which is also provided by the Vavr library (see Section 5.2.4) and can be equated with the *Exception* monad (see Section 3.3.2). In case the method has the return type *void*, *Try*<*Void*> is used as the new return type. Second, the throws and the associated exceptions are removed from the method signature. Third, the exit points of the methods have to be adjusted accordingly. The *Try* class also provides the static method *of*, however, this method takes a lambda expression as an argument. Code that either returns a value or throws one or multiple exceptions can be wrapped inside this lambda expression. The possibly thrown exceptions or the returned value are then automatically and correctly wrapped inside the *Try* datatype. Suppose an exception is explicitly thrown inside the method. In that case, it can be put into *Try* by passing the exception object to the the static method *failure* instead of throwing it using the *throw* statement. Successfully computed values can be wrapped by using the static method *success*. All three techniques are valid approaches to adapt the method body to the new method signature.

The third type (see Listing 5.3) is a mixture of type one and type two, whereby the method either returns a nullable value or throws one or multiple exceptions. To handle both cases, the absence of a value and the case of an exception, both cases are represented in the new return type *Try*<*Option*<*T*>>. For keeping a consistent style, the *Try* type is always used as the outer type.

Other methods that cannot be associated with any of the explained method types are not rewritten to *source* method in the initial phase. In addition, list types are not dealt with specifically (e.g. in using the *List* monad).

5.5.2 *Service* methods

The rewriting of *source* methods produces, in most cases, errors in methods that use those rewritten methods. Common errors are, for example, that the return value of a rewritten method is stored in a variable that has the wrong type or that a *try-catch* block is used, although no exception is thrown anymore. However, cases in which the new method return type is *Try*<*Void*> have to be treated specially since there may not be any errors after the rewriting from *void* to *Try*<*Void*> (e.g. the thrown exception is forwarded in methods where the rewritten method is used).

A method that uses at least one modified method is, in this context, referred to as a *service* method. In order to get an idea of the impact of monadic methods in the data access layer on the rest of the code, the method body must be adjusted accordingly. Furthermore, the method signatures of *service* methods are also rewritten under certain circumstances:

Signature Rewrites

A *service* method with a rewritten signature is in this context referred to as an *intermediate* method. On the other hand, service methods with unchanged signatures are referred to as *sink* methods. Rewritten methods can therefore be classified as either *source*, *intermediate* or *sink* method. The classification *service* method defines an umbrella term for *intermediate* and *sink* methods. Furthermore, it is possible that *source* methods may get changed to *intermediate* methods during the rewriting process (e.g. *source* methods using other rewritten methods).

In the following, the rules that indicate under which circumstances the method signatures are rewritten, as well as other rules regarding the rewrite process, will be presented:

***null* Dependence** If the *service* method can be classified as type one (see Listing 5.1) and the possible returned *null* value directly depends on the result of one of the rewritten methods used, the return type of the signature of the *service* method is also changed to return *Option<T>*.

Exception Dependence If the *service* method can be classified as type two (see Listing 5.2) and the type of the thrown exception equals the type of exception that was thrown by one of the rewritten methods used, or the exception thrown directly depends on the result of one of the rewritten methods used, the return type of the signature of the *service* method is also changed to return *Try<T>* and the dependent exception gets removed from the method signature. If there was no dependence on one of the rewritten methods, the impact on other service methods would no longer originate from the initially changed query and command methods of the data access layer.

Exception and *null* Dependence A combination of the two cases mentioned (see *null* Dependence and Exception Dependence) is also a valid case which results in a method signature rewrite of the *service* method. The new return type is *Try<Option<T>>*.

Dependence Limitation If one of the above cases (see Exception Dependence, *null* Dependence or Exception and *null* Dependence) applies to a *service* method, but the *service* method is overwritten, the method signature and the related base method is only changed if the declaration of the method is contained in the project and the method is overwritten exactly once in the project. Without this rule, it might be possible that the impact on other *service* methods does not result from the originally changed *source* methods.

Body Rewrites

Whenever possible, the *bind* operator (see Section 3.2) should be used as often as possible in *service* methods. The *flatMap* method defined for the *Try* and *Option* type can be considered equivalent to *bind*.

However, the use of *bind* in Java is restricted in specific situations. For example, "a lambda expression can only access local variables and parameters of the enclosing block that are *final* or effectively *final*" [Orand, p. 243]. As a result, *flatMap* can only be used if no local variables and parameters that are not *final* or not effectively *final* are used between the rewritten method and a second monadic expression. Furthermore, statements like *break* or *continue* can also not be used in lambda expressions. Refactoring or restructuring the code can help to overcome these limitations in certain situations. However, refactoring and code restructurings could falsify the results because it could not be clear whether improvements or deterioration in complexity result from the code restructuring, refactoring or the originally rewritten *source* methods. Hence, the traceability of the results is not ensured.

The following rules are intended to restrict freedom of action while rewriting, reduce bias and make results traceable. Furthermore, the set of rules can be viewed as a guideline for how to proceed in certain situations.

Refactoring or Restructuring Refactoring and restructuring the source code is not permitted. Furthermore, the declaration of additional Java classes or methods and transmission of code to other Java classes or methods is also prohibited. The reason for the existence of those rules is that refactoring and restructuring can result in changes in complexity that are not related to the starting point of this study. Also, if new methods are created, no comparison values from a previous version of the method exist.

The behavior of the existing source code must always be reproduced directly. If it is not possible to use a monadic programming style due to limitations, the monadic structure must then be checked manually for success or failure and the corresponding (unpacked) value must get used the same as before.

Additional Data Access Object Method Calls In case other query or command methods from the data access layer are used on the same nesting level as one of the rewritten methods and the methods are either of type one (see Listing 5.1), type two (see Listing 5.2) or type three (see Listing 5.3), then those methods get rewritten to *source* or *intermediate* methods aswell, using the same rules as presented in this section.

Service Method Returning *Try* If a service method is rewritten to return the type *Try* (because of Exception Dependence or Exception and *null* Dependence) and the method still throws other exceptions, then all other exceptions get wrapped into a *Try* object in the method body as well. Also, the throws statement gets removed from the method signature. The reason why non-dependent exceptions are also wrapped is that returning *Try* and throwing exceptions is considered redundant.

***Catch* Blocks Without Executable Code** When a *try-catch* is used to catch exceptions from a method in Java, the *try* block must always have a corresponding *catch* block, even if no code is executed in the *catch* block. This does not apply to the monadic

version of those methods. Therefore, *catch* blocks do not have to be reconstructed in the monadic version, even if the *catch* block contains comments.

No *null* Check If a method of type one (see Listing 5.1) is called in the original version of a *service* method and the returned value is not checked for containing *null*, then the method *getOrElse* from the *Option* class, which either unpacks the wrapped value or returns the passed default value, must be used with the parameter *null* in the monadic version. *flatMap* or a check for a successful computation must not be used as this would change the original behavior of the method.

All Exceptions Caught If one of the *source* or *service* methods used is either of type two or three (see Listing 5.2) or 5.3) and in the corresponding (closest) *try-catch* block of the original *service* method all thrown exceptions are caught and no *finally* block exists, then all other thrown exceptions can be wrapped in a *Try* object (by using *Try.of*) as well and the *try-catch* can be removed. However, if non-*final* or non-effectively *final* variables are used in the exception throwing line or block, wrapping is not allowed.

Non-*Final* Relaxation If *flatMap* cannot be used because of non-*final* and non-effectively *final* variables and all non-*final* variable assignments are made only before the first use of a rewritten method, then a *final* or effectively *final* template variable, which holds a copy of the original variable, can be created and used in lambda expressions.

Helper Methods of Monadic Types It is allowed to use helper methods that are provided by the monadic structures *Option* and *Try*. These helpers include, for instance, methods for filtering and unpacking stored values or methods for executing code, depending on the state of the monadic structure. The methods for unpacking wrapped values also offer parameters that can be used to define what should happen if the monadic structure has an error state. Furthermore, converter methods for converting the type *Option* to *Try* and vice versa exist and may be used.

5.6 Rewrite Approach

The following step-by-step guide describes the entire rewriting process:

1. Decide in which data access object class which query or command methods should be rewritten in a monadic version of the same method
2. Rewrite the selected methods accordingly to the above-defined rules for *source* methods (see Section 5.5.1)
3. Successively rewrite *service* methods in a monadic way and thus fix errors in *service* methods. This step must also be implemented in compliance with the rules (see Section 5.5.2)

4. If software tests exist, they must also be adjusted accordingly

5.7 Verification

5.7.1 Software Tests

The used evaluation project consists of 217 software tests, where one test is intentionally skipped. Before the start of the practical part of this work, the 216 tests were run to obtain the result of the test suite at the initial stage of the study. At this point, all 216 run tests passed. During the rewrite, the tests were executed regularly to find programming mistakes fast. After rewriting, all 216 tests still passed. However, while rewriting the project, five mistakes were found by using the software tests.

Additionally, the tool JaCoCo was used to get an approximate estimate of the code coverage. The overall coverage of the project after the rewrite was 24%. However, most of the project's untested classes were related to the user interface, which is usually not covered by unit tests. Most of the classes containing business logic have a test coverage of around 70%. Table 5.1 provides a detailed breakdown of code coverage for all classes that contain changed methods. Each classpath in the table is relative to the path *src/main/java/org/sufficientlysecure/keychain* of the main module.

Table 5.1: Code coverage for changed classes

Class	**Coverage** [%]
daos.KeyRepository	60
daos.KeyWritableRepository	74
operations.BackupOperation	76
operations.BaseOperation	82
operations.CertifyOperation	62
operations.ChangeUnlockOperation	0
operations.EditKeyOperation	0
operations.PromoteKeyOperation	81
operations.RevokeOperation	0
operations.UploadOperation	0
pgp.PgpDecryptVerifyOperation	75
pgp.PgpSignatureChecker	83
pgp.PgpSignEncryptOperation	72
remote.ui.RequestKeyPermissionPresenter	0
remote.OpenPgpService	0
remote.SshAuthenticationService	0
service.PassphraseCacheService	10
ssh.AuthenticationOperation	49
ui.adapter.ImportKeysAdapter	0
ui.keyview.KeyFragmentViewModel	0
ui.keyview.UnifiedKeyInfoViewModel	0

Class	Coverage [%]
ui.keyview.ViewKeyActivity	0
ui.keyview.ViewKeyFragment	0
ui.transfer.presenter.TransferPresenter	0
ui.token.PublicKeyRetriever	0
ui.BackupRestoreFragment	0
ui.CertifyFingerprintFragment	0
ui.CertifyKeyFragment	0
ui.CreateKeyFinalFragment	0
ui.DecryptFragment	0
ui.DeleteKeyDialogActivity	0
ui.PassphraseDialogActivity	0
ui.QrCodeViewActivity	0
ui.SecurityTokenOperationActivity	0
ui.ViewKeyAdvActivity.ViewKeyAdvViewModel	0
ui.ViewKeyAdvShareFragment	0
ui.ViewKeyAdvSubkeysFragment	0
ui.ViewKeyAdvUserIdsFragment	0
util.ShareKeyHelper	0

5.7.2 Code Review

On July 9, 2020, a code review for the code that was changed as part of this work was conducted. The review was performed by a project independent software engineer with more than four years of practical experience in a related field.

Before the start of the code review, a computer showing the code changes made was prepared in a quiet environment, a brief introduction to the topic was given and the defined rule set for the rewriting process was explained. Following the introduction, the review started at 5:55 PM CET.

Four minutes after the start, the reviewer found an implementation mistake in the method *KeyRepository.getCanonicalizedSecretKeyRing*. After a short discussion, the problem was fixed and reviewed again. After correcting the problem, the review was continued. At 6:13 CET the reviewer pointed out an *else* block without executable code. The block was deleted after the notice. At 6:35 PM CET, the reviewer found a rewritten method signature in the Java *interface PassphraseCacheInterface*, which was still throwing an exception. This finding led to the deletion of the *throws* statement and the corresponding exception. The deletion had no impact on the rest of the source code. At 7:18 PM CET the review ended.

CHAPTER 6

Results

The previous chapter (see Chapter 5) explained the relevant information about the practical part of this work and the basic process of how and under what conditions the results of this work were generated. This chapter, in turn, presents and discusses the obtained results. First, the overall results are viewed from a general perspective. Following the general perspective, the results are summarized in logical groups and analyzed in detail. All displayed complexity results are based on the calculations of the MetricsReloaded tool (see Section 5.2.3) and classpaths in tables are all relative to the path *src/main/java/org/sufficientlysecure/keychain* of the main module.

6.1 General Results

Performing the rewrite of the methods in accordance with the established set of rules (see Section 5.5) led to a change of a total of 39 Java classes and 88 methods. Table A.1 in the appendix visualizes the type (*source*, *intermediate* or *sink*; see Section 5.5), the measured *Cyclomatic Complexity* (see Section 4.2.2) and *Halstead Difficulty* (see Section 4.2.3) for all of the changed methods before and after the rewriting into a monadic programming style. In all of the following result tables, the values of the individual results of the *Halstead Difficulty* and the arithmetic mean results have been rounded automatically to two decimal places. Furthermore, method names were replaced by IDs due to their irrelevance. A mapping from method IDs to the corresponding method names can be found in Table A.2. Also, the metric *Cyclomatic Complexity* is abbreviated as *CC* and the metric *Halstead Difficulty* as *HD*. Columns with the heading *Diff* (short for *Difference*) show the change in complexity.

Based on the 13 originally changed *source* methods, eight *source*, 16 *intermediate* and 64 *sink* methods resulted from the rewriting. Looking at the arithmetic mean of the change in complexity in all changed methods, it can be seen that the rewriting of the source code resulted in an improvement of the *Cyclomatic Complexity* and a deterioration of the

Halstead Difficulty. On closer inspection, the *Cyclomatic Complexity* changed from $7,36$ to $7,32$ ($-0,54\%$) and the *Halstead Difficulty* changed from $27,53$ to $29,80$ ($+8,25\%$). These results shows that there is a disconnect between the two complexity metrics.

The analysis of the *NCLOC* metric showed that the number of *NCLOC* for all methods that were changed during the rewrite process increased from a total of 3249 to 3361 *NCLOC* ($+3,45\%$). Although an increase of 112 *NCLOC* can be recognized, this is considered negligible.

6.2 Detailed Analysis

After analyzing the results from a higher perspective, the results were divided into logical groups and analyzed in more detail. The categorization is based on the state change of the result of the complexity metric, whereby the following three state changes exist:

- Metric unchanged - Complexity unchanged - Displayed as $=$
- Metric improvement - Complexity decreased - Displayed as $\Uparrow$
- Metric deterioration - Complexity Increased - Displayed as $\Downarrow$

After only considering the *Cyclomatic Complexity* and the *Halstead Difficulty* at method level, nine different clusters could be derived. The clusters with the corresponding state changes are shown in Table 6.1. In the rightmost column, the table also presents how many methods are assigned to a specific cluster. In the following subsections, each of the clusters is analyzed individually. The name of the subsection indicates which cluster is being addressed and how the state has changed by displaying the above described state change symbols in parentheses. The first state change symbol in the parentheses relates to the *Cyclomatic Complexity* and the second to the *Halstead Difficulty*.

6.2.1 Cluster 1 ($=$ / $=$)

Cluster 1 contains all methods in which neither the *Cyclomatic Complexity* nor the *Halstead Difficulty* has changed. A total of ten methods ($11,\overline{36}\%$) are assigned to this cluster.

Of these ten methods, five are classified as *source* methods. Since this work deals with the impact of the monadic style, these methods are not considered in more detail. However, all these *source* methods have the monadic return type *Try* in common. The *Cyclomatic Complexity* remains unchanged because no additional path is added by lifting a value into the context of the *Try* datatype. By removing the throws statement from the method signature and by introducing a new operator to the method (e.g. *Try.of*), the *Halstead Difficulty* also remained unchanged.

Table 6.1: Clustering of rewritten methods

Cluster #	Cyclomatic Complexity	Halstead Difficulty	Count
1	=	=	10
2	⇓	⇓	10
3	⇑	⇑	7
4	⇑	=	1
5	⇓	=	0
6	=	⇑	1
7	=	⇓	54
8	⇑	⇓	5
9	⇓	⇑	0
			88

"=" indicates an unchanged result; "⇑" indicates an improvement, "⇓" indicates a deterioration

One method is classified as an *intermediate* method in which neither the *Cyclomatic Complexity* nor the *Halstead Difficulty* has changed. The *intermediate* method *SshAuthenticationService.getPublicKey* (*m45*) uses two rewritten methods, whereby no explicit *null* check or *try-catch* block exists since both of the methods used return the type *Try<T>* and the method itself was forwarding the previously thrown exception. Although the used methods can be sequentially executed by using *flatMap* and the sequential execution can be returned directly (e.g. *intermediate* method), no improvement was achieved in either of the two metrics.

The remaining four methods are classified as *sink* methods. In one of those methods, one of the rewritten methods was used in an anonymous class created inside the *sink* method. For this reason, the method is not considered further. In the remaining three methods, either only the generic parameter in the return type or the type of a local variable has changed. Again, no change was found in any of the metrics.

6.2.2 Cluster 2 (⇓ / ⇓)

The second cluster also consists of 10 methods ($11,\overline{36}\%$) containing 9 *sink* and 1 *intermediate* method. Table 6.2 shows an exclusive view of those methods and Figure 6.1 visualizes the results of this cluster. Overall, the *Cyclomatic Complexity* changed from $20,90$ to $22,10$ ($+5,74\%$) and the *Halstead Complexity* from $70,26$ to $78,50$ ($+11,73\%$) in this cluster. Both metrics indicate an increase in complexity.

In the method *BackupOperation.writePublicKeyToStream* (*m18*), the *Cyclomatic Complexity* increased from 3 to 4 and the *Halstead Difficulty* increased from $14,25$ to 22. The increase can be traced back to the additional *isSuccess* check in the not removed *try-catch* block, the duplicated error case and the additionally used (duplicated) method calls.

The method *writeSecretKeyToStream* (*m19*) of the class *BackupOperation*, *CertifyOpera-*

tion.execute (*m21*) and *PgpSignEncryptOperation.executeInternal* (*m31*) have a similar behavior as the previously described method *BackupOperation.writePublicKeyToStream*.

In *handleEncryptedPacket* (*m28*) of the class *PgpDecryptVerifyOperation* and *setData* (*m50*) of the class *ImportKeysAdapter*, a sequential execution by using *flatMap* was not possible. This led to explicit checks for successful calculations, which then resulted in an increase of the *Cyclomatic Complexity* and the *Halstead Difficulty*. Reasons for not using *flatMap* were the usage of the *continue* keyword and that the monadic methods were used on different blocks of nesting.

The *try-catch* block in the method *OpenPgpService.encryptAndSignImpl* (*m37*) was removed in compliance with the established rules. However, manual success checks also increased the complexity here for both metrics.

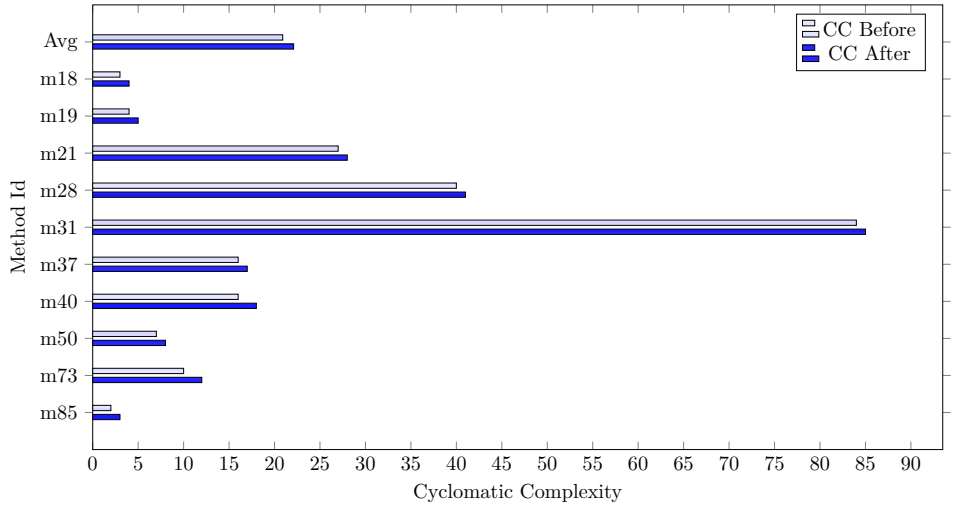
In *SshAuthenticationService.authenticate* (*m40*), the complexity also increased because of explicit *isSuccess* checks due to variable assignments between monadic methods.

Due to an anonymous class declaration, as already mentioned in cluster 1 (see Section 6.2.1), the method *checkPassphraseAndFinishCaching* (*m73*) in the inner class *PassphraseDialogFragment* of the class *PassphraseDialogActivity* was also excluded from a detailed analysis.

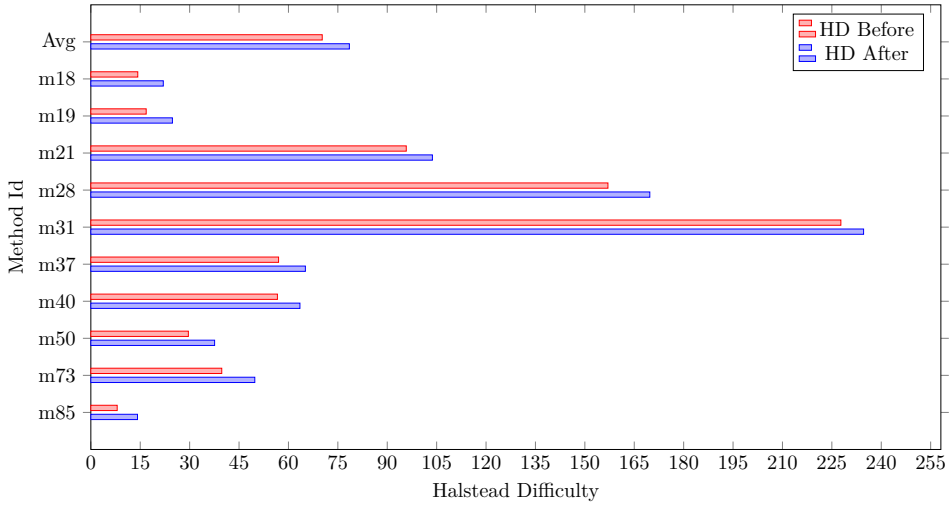
ShareKeyHelper.getSshKeyContent (*m85*), the last and only method with type *intermediate* in this cluster, also deteriorates in complexity. Here, using a monadic programming style was not possible because of additionally thrown exceptions. Theoretically, an improvement of the complexity metrics could be achieved if conditions for a monadic style existed.

Table 6.2: Cluster 2

Method Id	CC Before	CC After	CC Diff	HD Before	HD After	HD Diff	Type
m18	3	4	1	14.25	22.00	7.75	Sink
m19	4	5	1	16.81	24.77	7.96	Sink
m21	27	28	1	95.78	103.71	7.94	Sink
m28	40	41	1	157.01	169.72	12.71	Sink
m31	84	85	1	227.74	234.64	6.90	Sink
m37	16	17	1	57.00	65.14	8.14	Sink
m40	16	18	2	56.65	63.49	6.84	Sink
m50	7	8	1	29.61	37.58	7.97	Sink
m73	10	12	2	39.75	49.78	10.03	Sink
m85	2	3	1	8.00	14.17	6.17	Intermediate
Avg	20.90	22.10	1.20	70.26	78.50	8.24	



(a) *Cyclomatic Complexity* difference visualization



(b) *Halstead Difficulty* difference visualization

Figure 6.1: Cluster 2 visualization

6.2.3 Cluster 3 (⇑ / ⇑)

This cluster's results are considered to be one of the most important of this study since the *Cyclomatic Complexity* and the *Halstead Difficulty* improved for seven methods in this cluster. Table 6.3 shows only the results of these methods, while Figure 6.2 visualizes them in a bar plot. The *Cyclomatic Complexity* decreased from $3,71$ to $2,57$ ($-30,73\%$) while the *Halstead Difficulty* decreased from $17,24$ to $15,26$ ($-11,48\%$).

In the two *intermediate* methods *getCanonicalizedPublicKeyRing* (*m1*) and *getCanonical-*

izedSecretKeyRing (*m2*) of the class *KeyRepository*, an explicit *null* check was removed by using monadic methods. Furthermore, by using the *bind* operation and convenience methods of the monadic structures, a single sequential execution was generated, which is returned directly. Even though additional convenience methods are used, an improvement for both metrics could be achieved.

The impact on the method *RevokeOperation.execute* (*m25*) is similar. Here, too, a *try-catch* block was removed and a single chained term was created, using methods from the monadic structure (in this case *flatMap*, *toTry* and *getOrElse*), and returned directly.

Also, the entire bodies of the methods *SshAuthenticationService.getX509PublicKey* (*m43*) and *RequestKeyPermissionPresenter.setRequestedMasterKeyId* (*m34*) were converted to a single sequential version and a *try-catch* block was resolved as well. However, this was only possible because all thrown exceptions were allowed to be wrapped into a *Try* object.

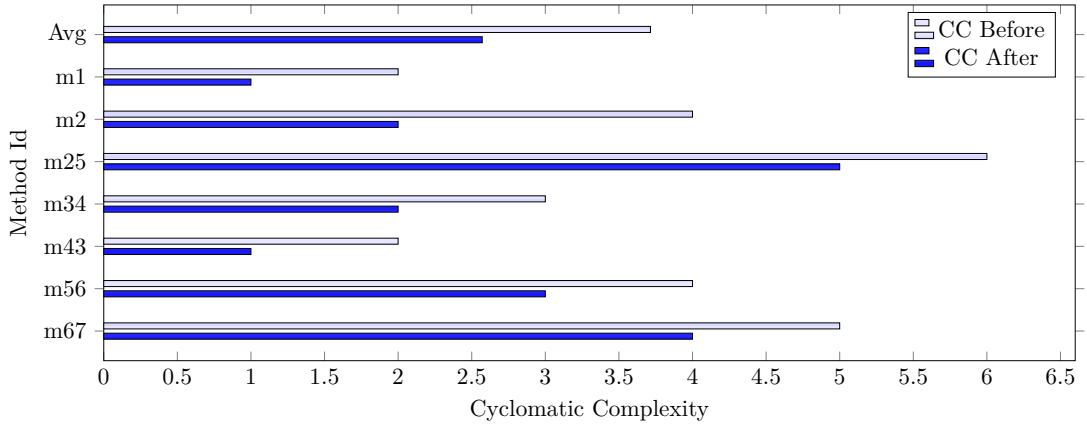
The method *DecryptFragment.loadSignerKeyData* (*m67*) is classified as a *sink* method. However, the usage of rewritten methods takes place in a lambda expression. In this lambda expression, an explicit *null* check was resolved and two monadic methods were concatenated using the *flatMap* method. Also, like above, the whole sequential execution is now directly returned.

In the last (*sink*) method of this cluster, a *try-catch* block was resolved and the *fold* method was used to receive a value depending on the state of the monad. The result of the convenience method call is now directly returned.

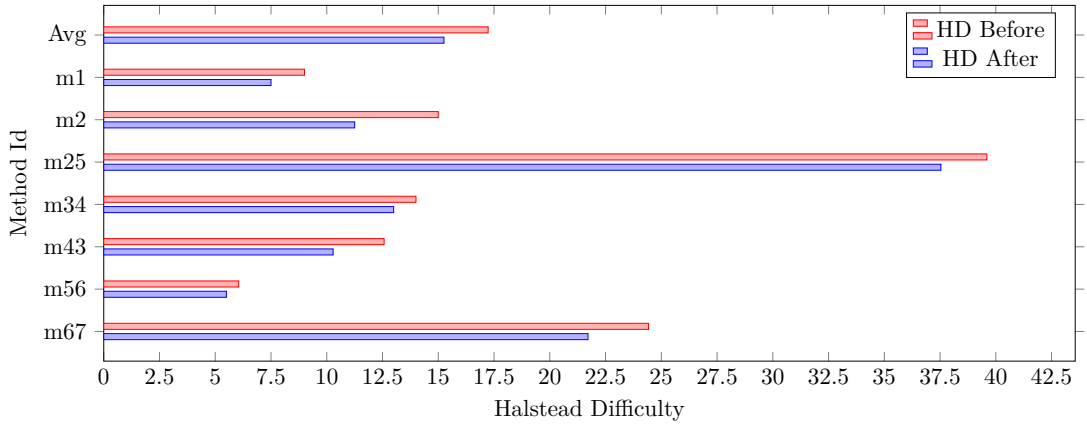
For most of the methods in this cluster, the Cyclomatic Complexity only decreased by 1. However, a decrease of 1 implies that an entire independent path could be removed (see Section 4.2.2).

Table 6.3: Cluster 3

Method Id	CC Before	CC After	CC Diff	HD Before	HD After	HD Diff	Type
m1	2	1	-1	9.00	7.50	−1.50	Intermediate
m2	4	2	-2	15.00	11.25	−3.75	Intermediate
m25	6	5	-1	39.60	37.54	−2.06	Sink
m34	3	2	-1	14.00	13.00	−1.00	Intermediate
m43	2	1	-1	12.57	10.29	−2.29	Intermediate
m56	4	3	-1	6.05	5.50	−0.55	Sink
m67	5	4	-1	24.43	21.71	−2.71	Sink
Avg	3.71	2.57	−1.14	17.24	15.26	−1.98	



(a) *Cyclomatic Complexity* difference visualization



(b) *Halstead Difficulty* difference visualization

Figure 6.2: Cluster 3 visualization

6.2.4 Cluster 4 (⇑ / =)

In exactly one method (*BaseOperation.getCachedPassphrase - m20*), an improvement of the *Cyclomatic Complexity* was achieved whereby the *Halstead Difficulty* stayed unchanged. The improvement of the *Cyclomatic Complexity* can be attributed to the use of the monadic structures. Since this method has been classified as an *intermediate* method and several exit points exist, all exit points must be treated accordingly. This treatment has balanced the complexity result of the *Halstead Difficulty*. Although the result of this cluster consists of only one entry, this cluster is visualized in Figure 6.3.

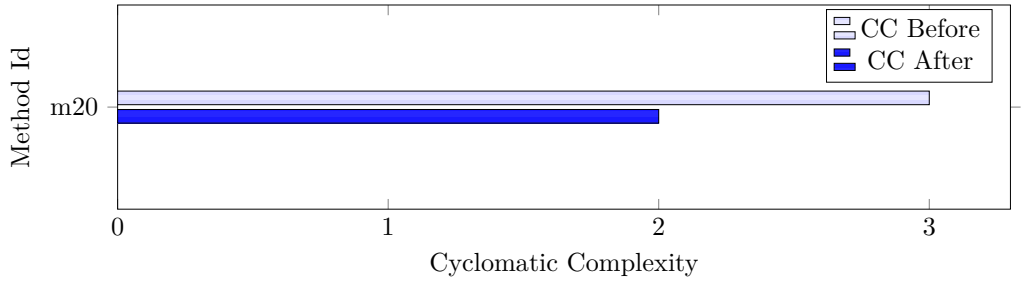


Figure 6.3: Cluster 4 visualization

6.2.5 Cluster 5 (⇓ / =)

There exist no cases where an increasing (worse) *Cyclomatic Complexity* left the *Halstead Difficulty* unchanged. This behavior can be attributed to the fact that an increasing *Cyclomatic Complexity* implies, that additional paths were added to the code. Additional paths introduce additional operators and/or operands to the method, hence the *Halstead Difficulty* changes as well.

6.2.6 Cluster 6 (= / ⇑)

For exactly one method, the *Cyclomatic Complexity* remained unchanged, while the *Halstead Difficulty* improved. This result of the *ViewKeyAdvShareFragment.onViewCreated* (*m80*) method is due to a change from an explicit *null* check (e.g. from == *null*) to the method call *isEmpty()*. In addition, the *get* method is called once to extract the value from the monad context. The single result is visualized in Figure 6.4.

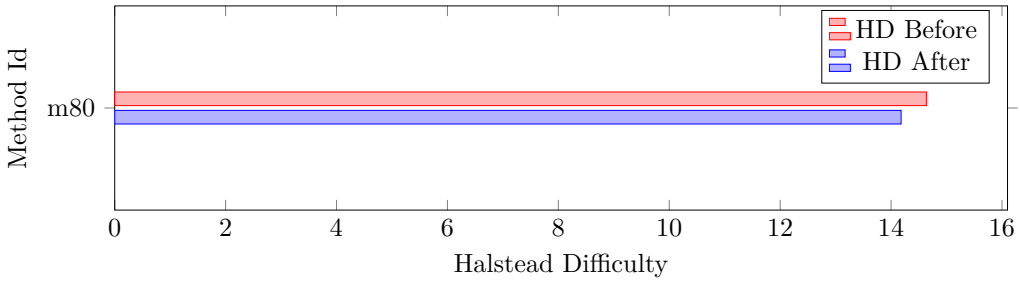


Figure 6.4: Cluster 6 visualization

6.2.7 Cluster 7 (= / ⇓)

With 54 of 88 methods ($61,\overline{36}\%$), cluster 7 is the largest of all. The methods in this cluster show an unchanged *Cyclomatic Complexity* and an increased *Halstead Difficulty*. Overall, the *Halstead Difficulty* of this cluster changed from $25,80$ to $27,97$ ($+8,41\%$). Due to a large number of methods, the behavior of each individual method is not explained in detail. For most methods, however, the pattern described in the following applies.

There are reasons why no sequential executions or lambda expression usages were possible. For example, the usage of non-*final* and non-effectively *final* variables, the usage of the *continue* keyword or the unavailability of other terms for sequential execution. In these cases, monadic structures are solely checked for success or failure. If the calculation is successful, the wrapped value is unwrapped and used as before. In the event of an error, the associated error code is executed.

Consider the code examples of an ordinary *try-catch* block (see Listing 6.1) and the corresponding monadic version (see Listing 6.2). When calculating the *Cyclomatic Complexity* and *Halstead Difficulty* for both methods, it can be seen that the *Cyclomatic Complexity* does not change from the value 2, but the *Halstead Difficulty* doubles from 4 to 8. Creating the same example with an explicit *null* check and change the code to use the monadic structure *Option*, the *Halstead Difficulty* changes from 4 to 6. In both examples, the *Cyclomatic Complexity* does not change while the *Halstead Difficulty* increases. Those code examples illustrate the underlying reason for the increasing *Halstead Difficulty* in most of the methods in this cluster.

However, six rewritten methods of the class *KeyRepository* do not match this pattern. Three of them are *source* methods, where the increase of the *Halstead Difficulty* can be attributed to the lifting of values into the monadic types. In the other three methods, a sequential execution was created and also returned directly. However, the reason why no improvement of any of the metrics could be achieved is that no explicit *null* check or *try-catch* block was resolved. The same behavior also applies to methods *SshAuthenticationService.getDescription* (*m46*) and *PassphraseCacheService.getCachedPassphraseImpl* (*m47*).

Listing 6.1: Ordinary *try-catch* example

```
void ordinaryTryCatch() {
  try {
    int i = getValue();
    System.out.println("Value: " + i);
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

Listing 6.2: Monadic *try-catch* example

```
void monadicTryCatch() {
  Try<Integer> i = getValueMonadic();
  if (i.isSuccess()) {
    System.out.println("Value: " + i.get());
  } else {
    i.getCause().printStackTrace();
  }
}
```

The results of this cluster are presented in Table 6.4 and visualized in Figure 6.5.

Table 6.4: Cluster 7

Method Id	CC Before	CC After	CC Diff	HD Before	HD After	HD Diff	Type
m3	1	1	0	4.00	4.67	0.67	Source
m4	1	1	0	2.50	3.00	0.50	Source
m8	1	1	0	4.00	5.00	1.00	Intermediate
m9	1	1	0	3.50	4.00	0.50	Intermediate
m10	4	4	0	13.64	14.32	0.68	Intermediate
m11	2	2	0	6.00	6.50	0.50	Source
m15	21	21	0	66.17	74.12	7.95	Sink
m16	19	19	0	63.48	66.45	2.97	Sink
m17	5	5	0	30.81	36.00	5.19	Sink
m22	7	7	0	29.81	31.56	1.74	Sink
m23	15	15	0	57.04	60.48	3.44	Sink
m24	8	8	0	36.88	37.41	0.53	Sink
m26	6	6	0	26.63	28.29	1.66	Sink
m29	5	5	0	19.46	24.92	5.46	Sink
m30	5	5	0	19.46	24.92	5.46	Sink
m32	6	6	0	23.68	25.41	1.74	Sink
m33	3	3	0	7.50	8.40	0.90	Sink
m35	6	6	0	13.81	17.00	3.19	Intermediate
m36	13	13	0	62.76	65.98	3.22	Sink
m38	10	10	0	36.09	39.94	3.84	Sink
m39	5	5	0	30.47	31.00	0.53	Sink
m41	3	3	0	11.79	14.67	2.88	Sink
m44	2	2	0	6.29	6.86	0.57	Intermediate
m46	1	1	0	6.00	7.50	1.50	Intermediate
m47	11	11	0	17.00	18.00	1.00	Intermediate
m48	18	18	0	67.85	70.83	2.98	Sink
m49	26	26	0	85.61	91.23	5.62	Sink
m51	3	3	0	8.00	9.17	1.17	Sink
m52	3	3	0	8.00	9.17	1.17	Sink
m53	3	3	0	9.75	11.00	1.25	Sink
m54	3	3	0	8.00	9.17	1.17	Sink
m57	22	22	0	98.22	101.14	2.92	Sink
m59	2	2	0	4.00	5.25	1.25	Sink
m60	2	2	0	7.70	14.00	6.30	Sink
m61	8	8	0	24.48	28.00	3.52	Sink
m63	2	2	0	5.63	6.67	1.04	Sink
m64	6	6	0	24.04	24.92	0.88	Sink
m65	5	5	0	32.80	34.55	1.75	Sink

Method Id	CC Before	CC After	CC Diff	HD Before	HD After	HD Diff	Type
m66	2	2	0	8.57	9.33	0.76	Sink
m68	9	9	0	32.03	34.17	2.14	Sink
m69	8	8	0	27.11	28.30	1.20	Sink
m70	5	5	0	31.33	32.37	1.04	Sink
m71	6	6	0	20.25	23.20	2.95	Sink
m72	20	20	0	99.62	107.05	7.42	Sink
m74	2	2	0	9.44	10.13	0.68	Sink
m75	17	17	0	79.83	83.68	3.85	Sink
m77	3	3	0	10.80	12.10	1.30	Sink
m78	3	3	0	10.80	12.10	1.30	Sink
m79	7	7	0	29.00	29.17	0.17	Sink
m81	2	2	0	7.70	9.00	1.30	Sink
m82	1	1	0	1.00	1.50	0.50	Sink
m83	2	2	0	4.00	5.25	1.25	Sink
m86	3	3	0	20.93	22.40	1.48	Sink
m88	5	5	0	18.00	19.00	1.00	Sink
Avg	6.65	6.65	0.00	25.80	27.97	2.17	

6.2.8 Cluster 8 (⇑ / ⇓)

All five methods with an improved *Cyclomatic Complexity* and a deterioration of the *Halstead Difficulty* have a resolution of a *try-catch* block in common. However, no sequential executions with other monadic methods were possible. Furthermore, other convenience methods of the monadic structures were used in all five methods (*onSuccess*, *onFailure*, *fold*, *recover*, *flatMapTry*). The increase of the *Halstead Difficulty* can be explained by the use of additional convenience methods (e.g. increase in count of operators and operands). The average *Cyclomatic Complexity* of this cluster changed from $4,4$ to 3 ($-31,82\%$) and *Halstead Difficulty* from $12,81$ to $15,71$ ($+22,64\%$). Table 6.5 shows a cutout of all of the results for this cluster, while Figure 6.6 displays the results of this cluster in a bar plot.

Table 6.5: Cluster 8

Method Id	CC Before	CC After	CC Diff	HD Before	HD After	HD Diff	Type
m14	4	3	-1	12.57	13.33	0.76	Sink
m27	5	4	-1	20.30	25.38	5.07	Intermediate
m42	6	3	-3	9.15	12.31	3.15	Sink
m84	2	1	-1	4.00	5.50	1.50	Intermediate
m87	5	4	-1	18.00	22.05	4.05	Sink
Avg	4.40	3.00	−1.40	12.81	15.71	2.90	

6.2.9 Cluster 9 (⇓ / ⇑)

Cluster 5 (see Section 6.2.5) already showed that there exist no cases where the *Halstead Difficulty* stayed unchanged with an increased *Cyclomatic Complexity*. Cluster 9 shows that an increase of the *Cyclomatic Complexity* with a decrease of the *Halstead Difficulty* was not found in any method. This could also be attributed to the fact that an increasing of the *Cyclomatic Complexity* implies that additional paths were added to the source code, which further implies that additional keywords, operators and/or operands were added as well. Adding additional operands or operators negatively impacts the *Halstead Difficulty* (see Section 4.2.3).

6.3 Discussion

This chapter dealt intensively with the results of this thesis. First, the results were described on a general level (see Section 6.1). It was shown that the rewriting of parts of the source code led to a change of 88 methods. Overall, the *Cyclomatic Complexity* was improved by $0,54\%$. The *Halstead Difficulty* has deteriorated by $8,25\%$ and, in addition, the *NCLOC* metric has increased by 112 NCLOC.

In the second part of this chapter, the results were divided into logical groups and analyzed in detail (see Section 6.2). Certain patterns could be derived from these groups. For example, cluster 3 showed that it is possible to improve the complexity for both metrics, the *Cyclomatic Complexity* and the *Halstead Difficulty*, by resolving explicit *null* checks and *try-catch* blocks and by creating and directly returning sequential executions.

On the other hand, cluster 7 showed that the same behavior, without resolving an explicit *null* check or *try-catch* block, can lead to a constant *Cyclomatic Complexity* and a worse (higher) *Halstead Difficulty*. Also, cluster 1 consists of one method where the same constellation resulted in no change in neither of the two metrics.

From cluster 2, it could be derived that if explicit *null* checks cannot be resolved and a monadic style also cannot be used, both metrics (*Cyclomatic Complexity* and *Halstead Difficulty*) deteriorate. Furthermore, the results showed that a deterioration of the *Cyclomatic Complexity* has always resulted in a worsening of the *Halstead Difficulty*.

Last but not least, it can be deduced that an improved *Halstead Difficulty* led in most cases (all but one) to an improvement of the *Cyclomatic Complexity*.

Now, to officially answer the research question by answering the subquestions:

Can the software complexity be reduced by using monadic query and command methods? Yes, as shown in this chapter, it is possible to reduce the software complexity under certain circumstances. If these circumstances are not met, the complexity may remain the same or even deteriorate.

How do the three different results of the complexity measurement relate to each other? Based on the detailed analysis of this study (see Section 6.2), there is no general answer to this question. There exist cases where both metrics, the *Cyclomatic Complexity* and the *Halstead Metric*, positively or negatively correlate. However, also cases exist where one metric showed an improvement while the other metric showed a deterioration or one metric remained unchanged while the other metric changed. In the case where both metrics have improved, on average, the *Cyclomatic Complexity* has decreased (improved) by 1.14 (30, 73%) and the *Halstead Difficulty* has decreased (improved) by 1.98 (11, 48%). The result of the *NCLOC* metric was negligibly small and was therefore excluded from the analysis.

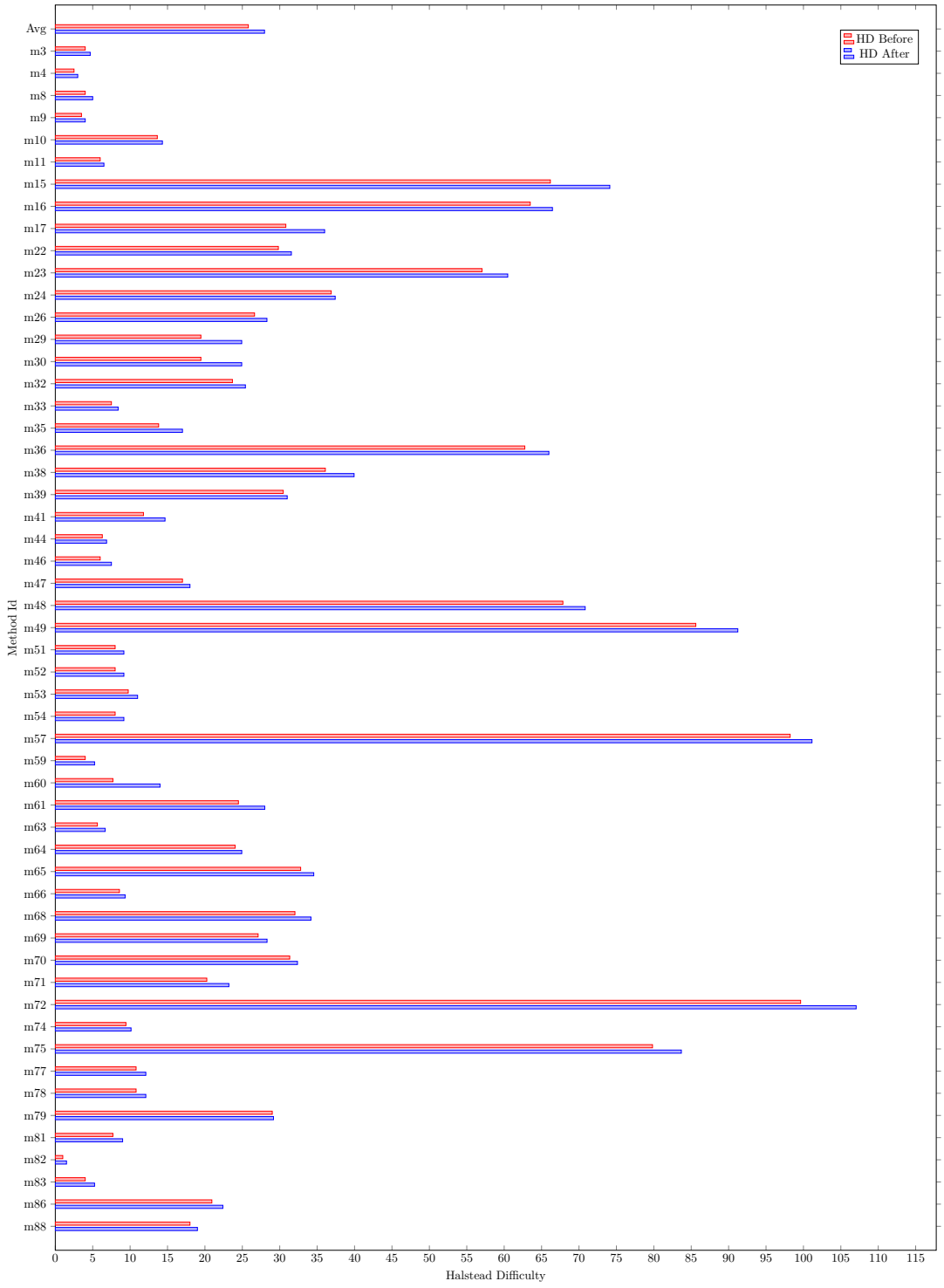
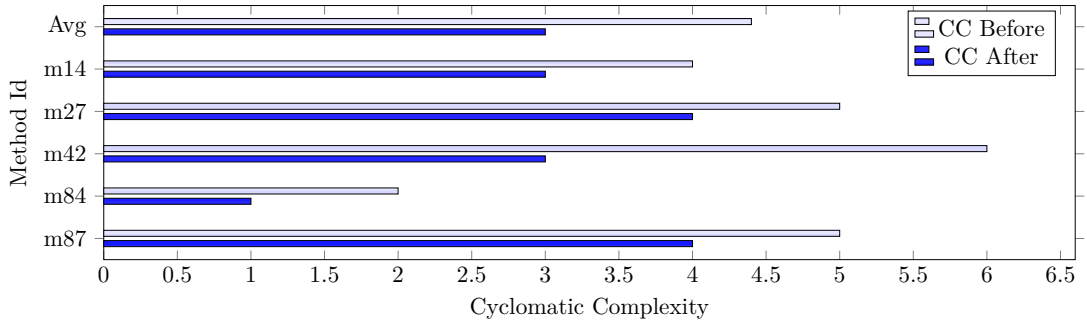
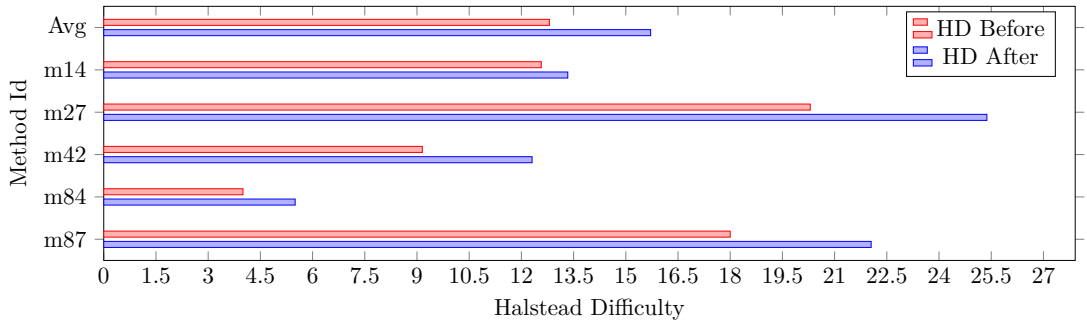


Figure 6.5: Cluster 7 visualization



(a) *Cyclomatic Complexity* difference visualization



(b) *Halstead Difficulty* difference visualization

Figure 6.6: Cluster 8 visualization

CHAPTER 7

Conclusion & Future Work

This thesis presented the impact of a monadic programming style in the data access layer on the software complexity of an Android application software, programmed in the mainstream object-oriented programming language Java. Before the results were presented, a detailed overview of the theoretical knowledge required was given. This overview presented the two programming paradigms object-oriented and functional programming in detail (see Chapter 2), explained the theoretical knowledge about monads by using practical examples (see Chapter 3) and also showed different methods for measuring software complexity (see Chapter 4).

Following the theoretical part of this thesis, the evaluation-related details were addressed in Chapter 5. This chapter covered information on the evaluation project and the tools used to support the practical part. Moreover, an extensive set of rules for the rewriting process was presented in order to make the rewriting process reproducible and transparent. After parts of the source code had been rewritten in compliance with the rules presented in Chapter 5, the results obtained were summarized and analyzed in Chapter 6.

The results showed that it is indeed possible to improve the complexity of the source code. Seven out of 88 rewritten methods showed an improved complexity using the complexity metrics *Cyclomatic Complexity* and *Halstead Difficulty*. However, ten methods show the exact opposite, namely a deterioration of both metrics. The result of the *NCLOC* metric was negligibly small.

To summarize the conducted work, the evaluation of the selected Android application software project in compliance with the established rules showed that it is possible to significantly improve the complexity under certain circumstances. However, if these circumstances do not apply, the monadic programming style can also harm the software complexity.

For future research, it would be interesting to repeat this study for further projects to determine whether the results can be used to infer the general from the specific.

In addition, based on the developed set of rules, it would be desirable to develop a computer program for predicting expected complexity changes for a potential rewrite of the data access layer to a monadic style. A prediction program can help project stakeholders with the decision on whether it is worth to rewrite a large program to a monadic version or not.

Furthermore, it would be interesting to extend the initial rewriting process to additional parts of the source code and then compare the gained results with the results of this work.

APPENDIX A

Additional Results

Due to their length and in order not to disturb the flow of reading, the following tables have been moved to the appendix. Table A.1 contains and presents the evaluation results of all changed methods and in Table A.2, the mapping of the method IDs used to the corresponding method names is shown.

Table A.1: All rewritten methods

Method Id	CC Before	CC After	CC Diff	HD Before	HD After	HD Diff	Type
m1	2	1	-1	9.00	7.50	−1.50	Intermediate
m2	4	2	-2	15.00	11.25	−3.75	Intermediate
m3	1	1	0	4.00	4.67	0.67	Source
m4	1	1	0	2.50	3.00	0.50	Source
m5	1	1	0	3.00	3.00	0.00	Source
m6	1	1	0	5.33	5.33	0.00	Source
m7	1	1	0	6.67	6.67	0.00	Source
m8	1	1	0	4.00	5.00	1.00	Intermediate
m9	1	1	0	3.50	4.00	0.50	Intermediate
m10	4	4	0	13.64	14.32	0.68	Intermediate
m11	2	2	0	6.00	6.50	0.50	Source
m12	1	1	0	4.67	4.67	0.00	Source
m13	1	1	0	4.67	4.67	0.00	Source
m14	4	3	-1	12.57	13.33	0.76	Sink
m15	21	21	0	66.17	74.12	7.95	Sink
m16	19	19	0	63.48	66.45	2.97	Sink
m17	5	5	0	30.81	36.00	5.19	Sink
m18	3	4	1	14.25	22.00	7.75	Sink
m19	4	5	1	16.81	24.77	7.96	Sink

Method Id	CC Before	CC After	CC Diff	HD Before	HD After	HD Diff	Type
m20	3	2	-1	11.00	11.00	0.00	Intermediate
m21	27	28	1	95.78	103.71	7.94	Sink
m22	7	7	0	29.81	31.56	1.74	Sink
m23	15	15	0	57.04	60.48	3.44	Sink
m24	8	8	0	36.88	37.41	0.53	Sink
m25	6	5	-1	39.60	37.54	−2.06	Sink
m26	6	6	0	26.63	28.29	1.66	Sink
m27	5	4	-1	20.30	25.38	5.07	Intermediate
m28	40	41	1	157.01	169.72	12.71	Sink
m29	5	5	0	19.46	24.92	5.46	Sink
m30	5	5	0	19.46	24.92	5.46	Sink
m31	84	85	1	227.74	234.64	6.90	Sink
m32	6	6	0	23.68	25.41	1.74	Sink
m33	3	3	0	7.50	8.40	0.90	Sink
m34	3	2	-1	14.00	13.00	−1.00	Intermediate
m35	6	6	0	13.81	17.00	3.19	Intermediate
m36	13	13	0	62.76	65.98	3.22	Sink
m37	16	17	1	57.00	65.14	8.14	Sink
m38	10	10	0	36.09	39.94	3.84	Sink
m39	5	5	0	30.47	31.00	0.53	Sink
m40	16	18	2	56.65	63.49	6.84	Sink
m41	3	3	0	11.79	14.67	2.88	Sink
m42	6	3	-3	9.15	12.31	3.15	Sink
m43	2	1	-1	12.57	10.29	−2.29	Intermediate
m44	2	2	0	6.29	6.86	0.57	Intermediate
m45	1	1	0	7.50	7.50	0.00	Intermediate
m46	1	1	0	6.00	7.50	1.50	Intermediate
m47	11	11	0	17.00	18.00	1.00	Intermediate
m48	18	18	0	67.85	70.83	2.98	Sink
m49	26	26	0	85.61	91.23	5.62	Sink
m50	7	8	1	29.61	37.58	7.97	Sink
m51	3	3	0	8.00	9.17	1.17	Sink
m52	3	3	0	8.00	9.17	1.17	Sink
m53	3	3	0	9.75	11.00	1.25	Sink
m54	3	3	0	8.00	9.17	1.17	Sink
m55	3	3	0	8.36	8.36	0.00	Sink
m56	4	3	-1	6.05	5.50	−0.55	Sink
m57	22	22	0	98.22	101.14	2.92	Sink
m58	1	1	0	25.50	25.50	0.00	Sink
m59	2	2	0	4.00	5.25	1.25	Sink
m60	2	2	0	7.70	14.00	6.30	Sink

Method Id	CC Before	CC After	CC Diff	HD Before	HD After	HD Diff	Type
m61	8	8	0	24.48	28.00	3.52	Sink
m62	14	14	0	43.75	43.75	0.00	Sink
m63	2	2	0	5.63	6.67	1.04	Sink
m64	6	6	0	24.04	24.92	0.88	Sink
m65	5	5	0	32.80	34.55	1.75	Sink
m66	2	2	0	8.57	9.33	0.76	Sink
m67	5	4	-1	24.43	21.71	−2.71	Sink
m68	9	9	0	32.03	34.17	2.14	Sink
m69	8	8	0	27.11	28.30	1.20	Sink
m70	5	5	0	31.33	32.37	1.04	Sink
m71	6	6	0	20.25	23.20	2.95	Sink
m72	20	20	0	99.62	107.05	7.42	Sink
m73	10	12	2	39.75	49.78	10.03	Sink
m74	2	2	0	9.44	10.13	0.68	Sink
m75	17	17	0	79.83	83.68	3.85	Sink
m76	3	3	0	7.00	7.00	0.00	Sink
m77	3	3	0	10.80	12.10	1.30	Sink
m78	3	3	0	10.80	12.10	1.30	Sink
m79	7	7	0	29.00	29.17	0.17	Sink
m80	2	2	0	14.64	14.18	−0.45	Sink
m81	2	2	0	7.70	9.00	1.30	Sink
m82	1	1	0	1.00	1.50	0.50	Sink
m83	2	2	0	4.00	5.25	1.25	Sink
m84	2	1	-1	4.00	5.50	1.50	Intermediate
m85	2	3	1	8.00	14.17	6.17	Intermediate
m86	3	3	0	20.93	22.40	1.48	Sink
m87	5	4	-1	18.00	22.05	4.05	Sink
m88	5	5	0	18.00	19.00	1.00	Sink
Avg	7.36	7.32	−0.04	27.53	29.80	2.27	

Table A.2: Method Id - Name Mapping

Method Id	Method Name
m1	daos.KeyRepository.getCanonicalizedPublicKeyRing
m2	daos.KeyRepository.getCanonicalizedSecretKeyRing
m3	daos.KeyRepository.getMasterKeyIdBySubkeyId
m4	daos.KeyRepository.getUnifiedKeyInfo
m5	daos.KeyRepository.getSecretKeyType
m6	daos.KeyRepository.getFingerprintByKeyId
m7	daos.KeyRepository.getKeyRingAsArmoredData

Method Id	Method Name
m8	daos.KeyRepository.getPublicKeyRingAsArmoredString
m9	daos.KeyRepository.getSecretKeyRingAsArmoredData
m10	daos.KeyRepository.loadPublicKeyRingData
m11	daos.KeyRepository.loadSecretKeyRingData
m12	daos.KeyRepository.getSecretSignId
m13	daos.KeyRepository.getEffectiveAuthenticationKeyId
m14	daos.KeyWritableRepository.getTrustedMasterKeys
m15	daos.KeyWritableRepository.savePublicKeyRing
m16	daos.KeyWritableRepository.saveSecretKeyRing
m17	daos.KeyWritableRepository.updateTrustDb
m18	operations.BackupOperation.writePublicKeyToStream
m19	operations.BackupOperation.writeSecretKeyToStream
m20	operations.BaseOperation.getCachedPassphrase
m21	operations.CertifyOperation.execute
m22	operations.ChangeUnlockOperation.execute
m23	operations.EditKeyOperation.execute
m24	operations.PromoteKeyOperation.execute
m25	operations.RevokeOperation.execute
m26	operations.UploadOperation.execute
m27	operations.UploadOperation.getPublicKeyringFromInput
m28	pgp.PgpDecryptVerifyOperation.handleEncryptedPacket
m29	pgp.PgpSignatureChecker.findAvailableSignature
m30	pgp.PgpSignatureChecker.findAvailableSignature
m31	pgp.PgpSignEncryptOperation.executeInternal
m32	pgp.PgpSignEncryptOperation.processEncryptionMasterKeyId
m33	remote.ui.RequestKeyPermissionPresenter.setupFromIntentData
m34	remote.ui.RequestKeyPermissionPresenter.setRequestedMasterKeyId
m35	remote.ui.RequestKeyPermissionPresenter.findSecretKeyRingOrPublicFallback
m36	remote.OpenPgpService.signImpl
m37	remote.OpenPgpService.encryptAndSignImpl
m38	remote.OpenPgpService.getKeyImpl
m39	remote.OpenPgpService.getSignKeyIdImpl
m40	remote.SshAuthenticationService.authenticate
m41	remote.SshAuthenticationService.getAuthenticationKey
m42	remote.SshAuthenticationService.getAuthenticationPublicKey
m43	remote.SshAuthenticationService.getX509PublicKey
m44	remote.SshAuthenticationService.getSSHPublicKey
m45	remote.SshAuthenticationService.getPublicKey
m46	remote.SshAuthenticationService.getDescription
m47	service.PassphraseCacheService.getCachedPassphraseImpl

Method Id	Method Name
m48	service.PassphraseCacheService.onStartCommand
m49	ssh.AuthenticationOperation.executeInternal
m50	ui.adapter.ImportKeysAdapter.setData
m51	ui.keyview.KeyFragmentViewModel.getIdentityInfo
m52	ui.keyview.KeyFragmentViewModel.getSubkeyStatus
m53	ui.keyview.KeyFragmentViewModel.getSystemContactInfo
m54	ui.keyview.KeyFragmentViewModel.getKeyserverStatus
m55	ui.keyview.UnifiedKeyInfoViewModel.getUnifiedKeyInfoLiveData
m56	ui.keyview.ViewKeyActivity.keyHasPassphrase
m57	ui.keyview.ViewKeyActivity.onLoadUnifiedKeyInfo
m58	ui.keyview.ViewKeyFragment.onActivityCreated
m59	ui.keyview.ViewKeyFragment.onLoadUnifiedKeyInfo
m60	ui.transfer.presenter.TransferPresenter.prepareAndSendKey
m61	ui.token.PublicKeyRetriever.retrieveLocal
m62	ui.BackupRestoreFragment.backupAllKeys
m63	ui.CertifyFingerprintFragment.onLoadUnifiedKeyInfo
m64	ui.CertifyKeyFragment.onActivityCreated
m65	ui.CreateKeyFinalFragment.moveToCard
m66	ui.DecryptFragment.showKey
m67	ui.DecryptFragment.loadSignerKeyData
m68	ui.DecryptFragment.onLoadSignerKeyData
m69	ui.DeleteKeyDialogActivity.onCreate
m70	ui.DeleteKeyDialogActivity.DeleteKeyDialogFragment.onCreateDialog
m71	ui.PassphraseDialogActivity.onCreate
m72	ui.PassphraseDialogActivity.PassphraseDialogFragment.onCreateDialog
m73	ui.PassphraseDialogActivity.PassphraseDialogFragment. checkPassphraseAndFinishCaching
m74	ui.QrCodeViewActivity.onLoadUnifiedKeyInfo
m75	ui.SecurityTokenOperationActivity.doSecurityTokenInBackground
m76	ui.ViewKeyAdvActivity.ViewKeyAdvViewModel.getUnifiedKeyInfoLiveData
m77	ui.ViewKeyAdvActivity.ViewKeyAdvViewModel.getSubkeyLiveData
m78	ui.ViewKeyAdvActivity.ViewKeyAdvViewModel.getUserIdLiveData
m79	ui.ViewKeyAdvActivity.onLoadUnifiedKeyInfo
m80	ui.ViewKeyAdvShareFragment.onViewCreated
m81	ui.ViewKeyAdvShareFragment.onLoadUnifiedKeyInfo
m82	ui.ViewKeyAdvSubkeysFragment.onLoadUnifiedKeyId
m83	ui.ViewKeyAdvUserIdsFragment.onLoadUnifiedKeyInfo
m84	util.ShareKeyHelper.getKeyContent
m85	util.ShareKeyHelper.getSshKeyContent
m86	util.ShareKeyHelper.shareKeyIntent

Method Id	Method Name
m87	util.ShareKeyHelper.shareKey
m88	util.ShareKeyHelper.shareSshKey

List of Figures

List of Tables

List of Code

Bibliography

[Abr10] A. Abran. *Halstead's Metrics: Analysis of Their Designs*, pages 145–159. 2010.

[AC94] Fernando Brito Abreu and Rogério Carapuça. Object-oriented software engineering: Measuring and controlling the development process. In *Proceedings of the 4th international conference on software quality*, volume 186, pages 1–8, 1994.

[AYK19] Mauricio Aniche, Joseph Yoder, and Fabio Kon. Current challenges in practical object-oriented software design. *Proceedings - 2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results, ICSE-NIER 2019*, pages 113–116, 2019.

[BM14] S Bhatia and J Malhotra. A survey on impact of lines of code on software complexity. In *2014 International Conference on Advances in Engineering Technology Research (ICAETR - 2014)*, pages 1–4, August 2014.

[CDS86] S D Conte, H E Dunsmore, and V Y Shen. *Software Engineering Metrics and Models*. Benjamin-Cummings Publishing Co., Inc., USA, 1986.

[CK94] S R Chidamber and C F Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[CMSR17] Sharon Christa, V Madhusudhan, V Suma, and Jawahar J Rao. Software Maintenance: From the Perspective of Effort and Cost Requirement. In Suresh Chandra Satapathy, Vikrant Bhateja, and Amit Joshi, editors, *Proceedings of the International Conference on Data Engineering and Communication Technology*, pages 759–768, Singapore, 2017. Springer Singapore.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR)*, 17(4):471–523, 1985.

[DW20] Daniel Dietrich and Robert Winkler. Vavr user guide. `https://www.vavr.io/vavr-docs/`, May 2020. (Accessed on 2020/07/21).

[GM10] Maurizio Gabbrielli and Simone Martini. *Programming Languages: Principles and Paradigms*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[GP01] Andy Gill and Ross Paterson. Control.monad.trans.reader. https://hackage.haskell.org/package/transformers-0.5.6.2/docs/Control-Monad-Trans-Reader.html, 2001. (Accessed on 2020/04/25).

[Hal77] Maurice H Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., USA, 1977.

[Has20] HaskellWiki contributors. Exception - haskellwiki. https://wiki.haskell.org/Exception, January 2020. (Accessed on 2020/04/25).

[Hin09] K Hinsen. The Promises of Functional Programming. *Computing in Science Engineering*, 11(4):86–90, July 2009.

[HMKD82] Harrison, Magel, Kluczny, and DeKock. Applying software complexity metrics to program maintenance. *Computer*, 15(9):65–79, 1982.

[HPF99] Paul Hudak, John Peterson, and Joseph Fasel. A Gentle Introduction to Haskell 98. *Functional Programming*, 1999.

[Hud89] Paul Hudak. Conception, Evolution, and Application of Functional Programming Languages. *ACM Comput. Surv.*, 21(3):359–411, September 1989.

[Hug89] J. Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, 1989.

[IEE90] IEEE. IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, pages 1–84, December 1990.

[IEE93] IEEE. IEEE Standard for Software Productivity Metrics. *IEEE Std 1045-1992*, pages 0_1–, 1993.

[IEE98] IEEE. IEEE Standard for a Software Quality Metrics Methodology. *IEEE Std 1061-1998*, pages i–, December 1998.

[Jetnd] JetBrains. MetricsReloaded - Plugins | JetBrains. https://plugins.jetbrains.com/plugin/93-metricsreloaded, n.d. (Accessed on 2020/08/07).

[Jon95] Mark P Jones. Functional Programming with Overloading and Higher-Order Polymorphism. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 97–136, Berlin, Heidelberg, 1995. Springer-Verlag.

[KF19] Shriram Krishnamurthi and Kathi Fisler. Programming Paradigms and Beyond. *The Cambridge Handbook of Computing Education Research*, pages 377–413, 2019.

[KY17] A. Khanfor and Y. Yang. An overview of practical impacts of functional programming. In *2017 24th Asia-Pacific Software Engineering Conference Workshops (APSECW)*, pages 50–54, 2017.

[LW94] Barbara H. Liskov and Jeannette M. Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841, November 1994.

[McC76] T J McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, December 1976.

[Mey88] B Meyer. *Object-oriented software construction*. Prentice-Hall international series in computer science. Prentice-Hall, 1988.

[Nag19] Gergely Nagy. Comparing software complexity of monadic error handling and using exceptions. *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2019 - Proceedings*, pages 1575–1580, 2019.

[Nar09] Ph Narbel. Functional programming at work in object-oriented programming. *Journal of Object Technology*, 8(6):181–209, 2009.

[Nob09] James Noble. The Myths of Object-Orientation. In Sophia Drossopoulou, editor, *ECOOP 2009 – Object-Oriented Programming*, pages 619–629, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[Orand] Oracle and/or its affiliates. Lambda expressions. `https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html`, n.d. (Accessed on 2020/07/14).

[Pet18] Tomas Petricek. What we talk about when we talk about monads The Art , Science , and Engineering of Programming. *The Art, Science, and Engineering of Programming*, 2(3):1–27, 2018.

[Pey07] Simon Peyton Jones. A History of Haskell: being lazy with class. In *The Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III)*, June 2007.

[Pre20] Oxford University Press. paradigm, n. `https://www.oed.com/view/Entry/137329`, March 2020. (Accessed on 2020/04/01).

[PSG12] Victor Pankratius, Felix Schmidt, and Gilda Garreton. Combining functional and imperative programming for multicore software: An empirical study evaluating Scala and Java. *Proceedings - International Conference on Software Engineering*, pages 123–133, 2012.

[Ren82] Tim Rentsch. Object Oriented Programming. *ACM SIGPLAN Notices*, 17(9):51–57, 1982.

[Rot86] Gian-Carlo Rota. Toposes, triples and theories. *Advances in Mathematics*, 61(2):184, 1986.

[Schnd] Dr.-Ing. Dominik Schürmann. Openkeychain. `https://www.openkeychain.org/`, n.d. (Accessed on 2020/07/21).

[SM17] Biswajit Saha and Debaprasad Mukherjee. Analysis of Applications of Object Orientation to Software Engineering, Data Warehousing and Teaching Methodologies. *International Journal of Computer Sciences and Engineering*, 5(9):244–248, 2017.

[Tan11] Antony Tang. Software designers, are you biased? In *Proceedings of the 6th International Workshop on SHAring and Reusing Architectural Knowledge*, SHARK '11, page 1–8, New York, NY, USA, 2011. Association for Computing Machinery.

[TBMP18] Damian A. Tamburri, Marcello M. Bersani, Raffaela Mirandola, and Giorgio Pea. Devops service observability by-design: Experimenting with model-view-controller. In Kyriakos Kritikos, Pierluigi Plebani, and Flavio de Paoli, editors, *Service-Oriented and Cloud Computing*, pages 49–64, Cham, 2018. Springer International Publishing.

[The01a] The University of Glasgow. Data.either. `https://hackage.haskell.org/package/base-4.12.0.0/docs/Data-Either.html`, 2001. (Accessed on 2020/04/27).

[The01b] The University of Glasgow. Data.maybe. `http://hackage.haskell.org/package/base-4.12.0.0/docs/Data-Maybe.html`, 2001. (Accessed on 2020/04/28).

[The01c] The University of Glasgow. Prelude. `https://hackage.haskell.org/package/base-4.12.0.0/docs/Prelude.html`, 2001. (Accessed on 2020/05/01).

[The01d] The University of Glasgow. System.random. `https://hackage.haskell.org/package/random-1.1/docs/System-Random.html`, 2001. (Accessed on 2020/05/02).

[The09] The University of Glasgow. Ghc.types. `https://hackage.haskell.org/package/ghc-prim-0.5.3/docs/src/GHC.Types.html`, 2009. (Accessed on 2020/04/29).

[TMM18] Timothy A.V. Teatro, J. Mikael Eklund, and Ruth Milman. Maybe and Either Monads in Plain C++ 17. *Canadian Conference on Electrical and Computer Engineering*, 2018-May:1–4, 2018.

[TSZ09] Honglei Tu, Wei Sun, and Yanan Zhang. The research on software metrics and software complexity metrics. *IFCSTA 2009 Proceedings - 2009 International Forum on Computer Science-Technology and Applications*, 1:131–136, 2009.

[Tur13] D A Turner. Some History of Functional Programming Languages. In Hans-Wolfgang Loidl and Ricardo Peña, editors, *Trends in Functional Programming*, pages 1–20, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[Wad90] Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, page 61–78, New York, NY, USA, 1990. Association for Computing Machinery.

[Wad93] Philip Wadler. Monads for functional programming. In Manfred Broy, editor, *Program Design Calculi*, pages 233–264, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[Weg90] Peter Wegner. Concepts and paradigms of object-oriented programming. *SIGPLAN OOPS Mess.*, 1(1):7–87, 1990.

[YC85] S S Yau and J S Collofello. Design Stability Measures for Software Maintenance. *IEEE Transactions on Software Engineering*, SE-11(9):849–856, 1985.

[Zus91] Horst Zuse. *Software Complexity*. De Gruyter, Berlin, Boston, 1991.