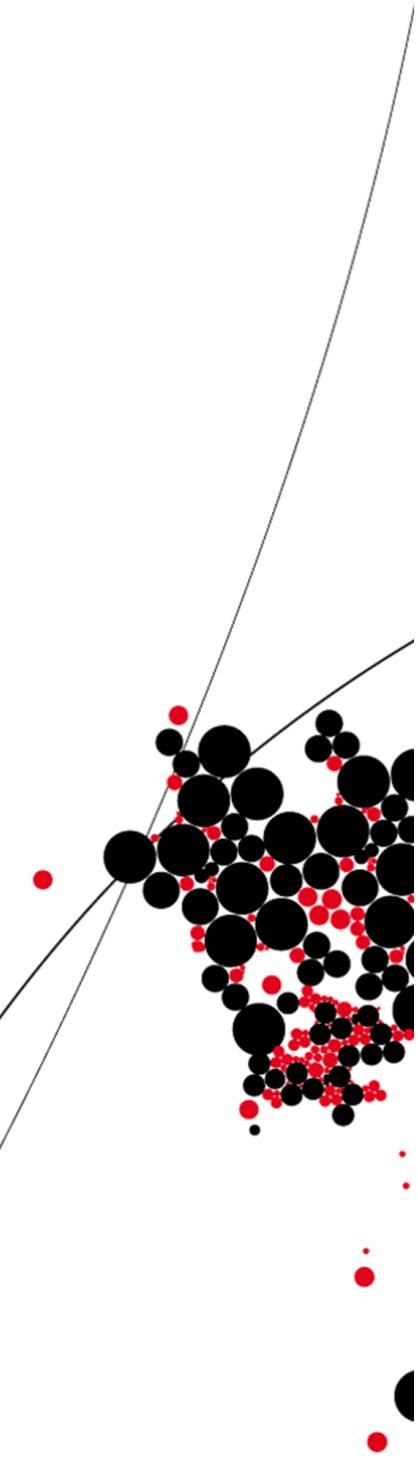
UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering, Mathematics & Computer Science

Source Code Metrics for Combined Functional and Object-Oriented Programming in Scala

Sven Konings

Master Thesis
Nov. 2020



Supervisors:
dr. A. Fehnker
dr. L. Ferreira Pires
ir. J.J. Kester (Info Support)

Formal Methods and Tools
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Abstract

Source code metrics are used to measure and evaluate the code quality of software projects. Metrics are available for both Object-Oriented Programming (OOP) and Functional Programming (FP). However, there is little research on source code metrics for the combination of OOP and FP. Furthermore, existing OOP and FP metrics are not always applicable. For example, the usage of mutable class variables (OOP) in lambda functions (FP) is a combination that does not occur in either paradigm on their own. Existing OOP and FP metrics are therefore unsuitable to give an indication of quality regarding these combined constructs.

Scala is a programming language which features an extensive combination of OOP and FP construct. The goal of this thesis is to research metrics for Scala which can detect potential faults when combining OOP and FP. We have implemented a framework for defining and analysing Scala metrics. Using this framework we have measured whether code was written using mostly OOP- or FP-style constructs and analysed whether this affected the occurrence of potential faults. Next, we implemented a baseline model of existing OOP and FP metrics. Candidate metrics were added to this baseline model to verify whether they improve the fault detection performance.

In the analysed projects, there was a higher percentage of faults when mixing OOP- and FP-style code. Furthermore, most OOP metrics perform well on FP-style Scala code. The baseline model was often able to detect when code was wrong. Therefore, the candidate metrics did not significantly improve the fault detection performance of the baseline model. However, the candidate metrics did help to indicate why code contained faults. Constructs were found for which over half of the objects using those constructs contained faults.

Acknowledgements

First of all, I would like to thank my supervisors: Jan-Jelle Kester (Info Support), Ansgar Fehnker (University of Twente) and Luís Ferreira Pires (University of Twente). Their questions, feedback and insights have helped greatly writing this thesis.

Furthermore, I would like to thank Info Support for providing the starting points and hosting this thesis. I would like to thank Rinse van Hees (Info Support) for providing interesting information and putting me into contact with Erik Landkroon, whom I would like to thank for taking the time to answer my questions about his work. I would like to thank Lodewijk Bergmans from the Software Improvement Group for taking the time to discuss the definitions of code quality and what the Software Improvement Group does to quantify it.

Finally, I would like to thank my friends and family for their support and always providing a listening ear.

- Sven

Contents

Chapter 1

Introduction

This chapter sets the stage for this Master's thesis. Section 1.1 explains the motivation for researching source code metrics aimed at the combination of object-oriented and functional programming. Section 1.2 explains the limitations of the currently available tooling and metrics. The scope of this thesis is defined in Section 1.3. Section 1.4 presents the research questions based on the introduced problems. Section 1.5 described the approach used to answer the research questions. Section 1.6 presents the contributions made by this thesis. Finally, the outline of this thesis is presented in Section 1.7.

1.1 Motivation

An important aspect of software projects is code quality, especially maintainability and reliability [25]. Poor code quality can lead to unreliable projects that are difficult to maintain. One way to increase maintainability and reliability is by using source code metrics. Source code metrics measure attributes of the code and can be used to locate code that is potentially unreliable or difficult to maintain. Metrics can be used during the development process to identify problematic code before it reaches production. Furthermore, metrics can provide pointers as to why code is unreliable. If we know certain constructs almost always cause issues we can develop and add patterns to tooling to prevent the use of these constructs.

Many source code metrics exist for Object-Oriented Programming (OOP) languages like Java [3, 15, 51] and C# [9, 19, 51]. Metrics have also been defined for Functional Programming (FP) languages like Haskell [37, 38]. However, there is little research on source code metrics for the combination of OOP and FP [30]. Furthermore, existing OOP and FP metrics are not sufficient when two paradigms are used in combination [53].

The combination of OOP and FP is becoming more and more common. Popular OOP languages, like Java and C#, have incorporated concepts from FP languages, like lambdas and higher-order functions [53]. In addition, Multi-paradigm Programming (MP) languages, like Scala and Kotlin, feature an extensive combination of OOP and FP constructs. More and more software projects are using these MP languages [8]. Therefore, source code metrics and tooling aimed at the combination of OOP and FP could be a great aid to improve the reliability and maintainability of these projects.

1.2 Problem statement

The combination of OOP and FP allows for powerful new (combinations of) programming constructs. However, these new combination can also cause new problems which existing OOP and FP metrics do not take into account. For example, the usage of mutable class variables (OOP) in lambda functions (FP) is a combination that does not occur in either paradigm on their own. With regards to these combined constructs, existing OOP and FP metrics are unsuitable to give an indication of quality.

The combination of paradigms also leads to new pitfalls. For example, FP code often assumes that lambda functions have no side-effects. However, this is not guaranteed in MP languages. An advantage of having functions without side-effects, also called pure functions, is that they can be lazily evaluated and operations using them can easily be parallelized. In MP languages functions are not guaranteed to be pure. An example where this causes issues is when using parallel collections. A small Scala program that demonstrates this can be found in Listing 1.1. In this program both calls should produce the same output. However, because the lambda has side-effects it cannot safely be parallelized and introduces concurrency issues when used in combination with parallel collections. These new pitfalls are not detected by traditional metrics and tooling. To increase the reliability and maintainability, common pitfalls should be detected before code enters production. Therefore, new metrics and patterns that can be used to warn when code is likely to contain these faults are needed.

```
var counter = 0
val list = (1 to 5).toList
val parallelList = list.par

println(list.map { i =>
  counter += 1
  i + counter
}) // Prints [2, 4, 6, 8, 10]

counter = 0

println(parallelList.map { i =>
  counter += 1
  i + counter
}) // Prints [3, 7, 4, 7, 9]
```

Listing 1.1: Scala parallel collection impure lambda example

1.3 Scope

There are many different programming languages featuring a combination of OOP and FP. Each of these languages has a unique combination of constructs. For this thesis, we have decided to focus on a single language, namely Scala. Scala has been designed as a combination of OOP and FP from the start and contains a very extensive mix of OOP and FP constructs. Scala exists since 2004 and is more mature than most other languages that have been designed as a combination of OOP and FP from the start. Scala has a good adoption rate and is ranked 16$^{\text{th}}$ most popular language as of March 2020 according to the PYPL index [8]. These properties ensure there is enough data available to analyse. Existing metrics and tooling, like SonarQube [45], have support for Scala. However, this support focuses on the OOP side of Scala and does

not cover faults that occur when mixing OOP and FP constructs [29].

1.4 Research questions

The goal of this thesis is to define metrics that can indicate potential faults when mixing OOP and FP (like existing metrics for OOP and FP). These metrics can be used to increase the code quality of Scala projects. A common method to detect faults is by predicting the fault-proneness of a piece of code [13]. The fault-proneness is the likelihood a piece of software contains faults. This leads to the main research question:

RQ *To what extent can fault-proneness prediction in Scala be improved using metrics tailored to the combination of OOP and FP?*

As a starting point for defining metrics, the usage of OOP and FP constructs has been measured. The fault-proneness prediction performance of these measurements has been analysed to identify whether some constructs are more fault-prone than others. Constructs that are significantly more fault-prone can be used as a starting point for defining metrics. This leads to the first subquestion:

RQ1 *Which OOP or FP constructs in Scala are significantly more fault-prone than others?*

The mix of OOP and FP within a piece of code has also been measured using a paradigm score. The paradigm scores attribute points to the usage of OOP and FP constructs. The resulting score is based on the ratio of OOP points to FP points. The full definition of the paradigm score can be found in Section 5.2. The fault-proneness prediction performance of the paradigm score has been measured to identify whether a mix of constructs is more fault-prone. This leads to the second subquestion:

RQ2 *How well does the paradigm score perform as a predictor for fault-proneness?*

Existing OOP and FP metrics can be used to predict the fault-proneness of Scala code [30]. One way to potentially improve the fault-proneness prediction is to select which metrics are used based on the mix of OOP and FP within a class. Before this can be done it is necessary to know how these metrics are affected by the mix of OOP and FP. This leads to the third subquestion:

RQ3 *To what extent is the fault-proneness prediction ability of existing OOP and FP metrics affected by the mix of OOP and FP within a class or method?*

To validate the new metrics a baseline model has been used. The baseline model consists of a set of commonly used OOP and FP metrics. The full definition of the baseline model can be found in Section 6.1. New metrics are considered validated when they significantly improve the fault-proneness prediction performance of the baseline model. This leads us to the final subquestion:

RQ4 *To what extent can the fault-proneness prediction performance of the baseline model be improved by adding metrics tailored to the combination of OOP and FP?*

1.5 Approach

The first step within this thesis was to select Scala projects and gather data that can be used for analysis. For the fault-proneness analysis it is important to select projects that keep track of

faults which occurred in the past and which parts of the code were changed to fix them.

Next, a framework was built for the analysis. First, the framework gathers the fault data and keeps track of which faults are related to which parts of the code. Next, the metrics of the code are measured. The fault data are combined with the metric measurements so that it is known how many faults are related to each measurement. Finally, the metric measurements are used to predict faults by using logistic regression and the resulting prediction performance is measured. The framework measures the prediction performance of individual metrics and the prediction performance of all metrics combined.

The next step was to answer the first two subquestions, to find out which constructs are promising predictors for defining new metrics and how well the paradigm score performs as a predictor for fault-proneness. Construct measurements were defined based on the analysis of Scala constructs in Section 2.2. Based on these construct measurements several paradigm score alternatives were defined. The fault-proneness prediction performance of the constructs and the paradigm score was measured to answer **RQ1** and **RQ2**.

Next, the baseline model was defined by selecting general, OOP and FP metrics based on existing literature. The prediction performance of the baseline model was measured based on the combined performance of all metrics. To determine to what extent OOP and FP metrics are affected by the mix of OOP and FP, the code was split up into four categories based on the paradigm score: OOP, FP, Mix of OOP and FP, and neutral. To answer **RQ3** the prediction performance of the individual metrics of the baseline model was measured on each category.

Finally, metrics tailored to the mix of OOP and FP were validated. These metrics were defined based on three main sources:

1. The metrics for the functional side of C# by Zuilhof [53].
2. The OOP or FP constructs that are significantly more fault-prone.
3. The existing OOP and FP metrics that are significantly affected by the mix of OOP and FP.

To select promising metrics, the prediction performance of the individual metrics was measured. The metrics that performed well were added to the baseline model to validate whether they significantly improve the combined prediction performance. This answers **RQ4** and the main research question.

1.6 Contributions

This section discusses the intended contributions of this thesis. The contributions are in fourfold.

Insights in fault-proneness when mixing OOP and FP First of all, this thesis provides insights into the fault-proneness of code when mixing OOP and FP. It provides an overview of constructs and their fault-proneness. Additionally, this thesis provides insights into how the paradigm score is related to fault-proneness. Finally, this thesis provides insights into how existing OOP and FP metrics are affected by the mix of OOP and FP.

Metrics tailored to the combination of OOP and FP This thesis provides metric definitions aimed at detecting problematic code when mixing OOP and FP have been defined and validated. The performance of each of the metrics has been measured. Furthermore, this thesis provides an overview of how the occurrences of constructs measured by the defined metrics correlates with the fault-proneness.

Metric analysis dataset All data used and produced in this thesis have been published. This includes the data needed to reproduce the results, like the data of the analysed projects and a cached version of the gathered issue data. The results produced by the metric measurements and the results produced by the fault-proneness predictions have also been published. All of these results can be found at `https://github.com/svenkonings/ScalaMetrics/tree/master/data`.

Analysis framework Finally, an analysis framework for Scala that can be used to validate new metrics or to reproduce the results presented in this thesis has been created. This framework is a further development of the work by Landkroon [30]. The analysis framework consists of four main components:

GitClient - Gathering Git project data and GitHub issue data.
CodeAnalysis - Developing and running metrics.
Validator - Gathering metric results combined with fault information across different versions.
ResultAnalysis - Measuring the prediction performance of metrics using logistic regression.

The framework can be found at `https://github.com/svenkonings/ScalaMetrics`.

1.7 Outline

This thesis is structured as follows. Chapter 2 gives the necessary background info on programming paradigms, Scala and code quality. Chapter 3 discusses the validation methodology that was used to evaluate metrics. The implementation architecture is discussed in Chapter 4. In Chapter 5 the constructs within Scala and the paradigm score are measured. Their fault-proneness prediction performance is evaluated to answer **RQ1** and **RQ2**. Chapter 6 presents the baseline prediction model and discusses how existing metrics are affected by the mix of OOP and FP to answer **RQ3**. Chapter 7 defines and validates metrics tailored to the combination of OOP and FP to answer **RQ4**. In Chapter 8 related work is discussed. Finally, the concluding remarks and future work are presented in Chapter 9.

Chapter 2

Background

This chapter gives the background information necessary to understand this thesis. Section 2.1 gives the background on OOP, FP, and the definition of multi-paradigm used in this thesis. Section 2.2 presents the Scala constructs identified in the preliminary research [29]. Section 2.3 discusses code quality characteristics and metrics.

2.1 Multi-Paradigm Programming

Multi-paradigm programming is a broad term that can be used for any combination of programming paradigms. In this thesis, multi-paradigm programming refers to the combination of object-oriented programming and functional programming.

Functional programming is part of the declarative programming paradigm. In declarative programming, the program logic is defined without describing the control flow. Functional programming originates from Lambda Calculus [12], a mathematical logic for expressing computations based on function abstraction, application and composition. In pure FP languages, state and side-effects are avoided and the type system is often based on algebraic data types.

Alan Kay originally introduced object-oriented programming in 1966 to define objects that encapsulate their internal state and communicate by passing messages [28]. Nowadays, object-oriented programming is often considered an extension of the imperative and procedural programming paradigms. This means objects represent their state using data fields and communicate using procedures (also known as methods). The state of the program can be changed by modifying fields and the control-flow is described in the procedures. Concepts like inheritance and polymorphism have also become part of the object-oriented programming paradigm.

Object-oriented programming can be seen as a paradigm on its own separate from the procedural and imperative paradigms. In this case object-oriented only refers to the type system and encapsulation, and not to the program logic implementation. Within this thesis, we generally refer to object-oriented as an extension of imperative and procedural programming, since this is the definition used within Scala, and will explicitly mention when we only refer to the type system and encapsulation properties of object-oriented programming.

When combining OOP and FP into a single multi-paradigm language, the type system and data encapsulation is often based on the OOP system. MP languages often contain constructs to make the OOP type system more suitable for functional programming, like making it easy to define objects that only act as data containers and having built-in (anonymous) function types. When combining OOP and FP, the FP language is no longer pure. The addition of OOP introduces mutable state and side-effects to the language.

2.2 Scala constructs

In this section, the overall design of Scala and the constructs identified in the preliminary research [29] are discussed to give the reader an idea of the Scala language. Afterwards, the combination of constructs in Scala is analysed and compared to their equivalents in pure OOP or FP languages. The goal is to analyze which constructs Scala contains and how they differ from OOP or FP languages. Additional info on Scala can be found in the Scala language tour [43] or the language specification [42].

2.2.1 Overall design

Scala is a statically typed JVM language that combines object-oriented and functional programming. Scala is designed to fully support both OOP and FP styles of programming [42]. Scala uses class-based object-oriented programming, where classes describe the structure of each object. Objects are used for the type system in Scala. Scala contains additional functionality to make it easier to use objects as algebraic data types for functional programming.

2.2.2 OOP constructs

In this section, the Scala constructs related to classes, objects, methods or changing the state of the program are discussed.

Variables

Variables can change the state of a program and are therefore classified as OOP construct. Variables in Scala can be mutable or immutable. Mutable variables can be reassigned different values as long as the type matches, immutable variables can only be assigned at declaration [33]. The keyword *var* is used for defining mutable variables and *val* is used for immutable variables. Immutable variables are comparable to final variables in Java. If an immutable variable contains a mutable object, for example a mutable list, this object can still be modified.

Classes

Classes are one of the basic building blocks of the OOP paradigm. Classes in Scala are similar to Java or C#. A class has one or more constructors (by default an empty constructor). Scala classes can have the same modifiers as Java classes (e.g. *abstract*, *protected*, etc.). Classes in Scala cannot have static values or methods. To use a class they have to be instantiated. An instance of a class is called an object. In Scala, every value is an object [33]. This includes values that are not objects in Java, like primitives. In this sense, Scala is a pure object-oriented language in contrast to Java.

A class can extend another class. Scala classes are single inheritance, which means that it is only possible to extend a single class at a time. The top-level class, which every class inherits, is the *Any* class. It defines certain universal methods such as *equals*, *hashCode*, and *toString*. The *Any* class has two subclasses, *AnyVal* and *AnyRef* [43].

AnyVal is used for value objects, which contain a value that corresponds to a primitive value in Java (for example, booleans, integers or doubles). Even the equivalent to a *void* keyword in Java, which indicates that a method does not return any type, is represented by an object. In Scala, this is the *Unit* object. *AnyRef* is used for reference classes, which are used for everything

that is not a value object. This makes *AnyRef* similar to *Object* in Java. See Figure 2.1 for an overview of the Scala type hierarchy.

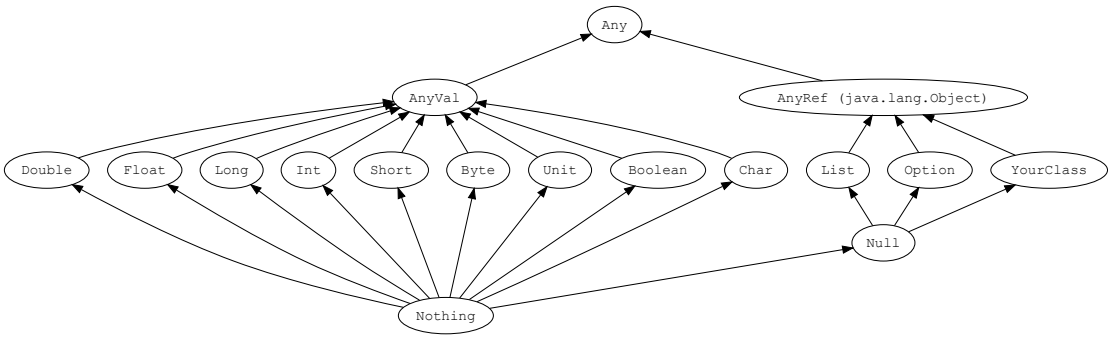


Figure 2.1: Scala type hierarchy.

Objects

In Scala, it is possible to declare an object directly. Such an object is similar to a singleton class and does not have to be instantiated. Instead, it can be accessed from anywhere in the code [43]. These singleton objects replace static values and methods. Singleton objects can share the same name with a class. In this case, they are called companion objects and contain the static members of the class [23].

Two special methods can be declared in an object, namely the *apply* and *unapply* methods. The *apply* method is similar to a factory method [14] since it takes certain arguments and returns an instantiated object. The *unapply* method does the opposite, it takes an object and tries to give back the arguments [43]. The *unapply* method is mostly used in pattern matching (see Section 2.2.3). An example of the *apply* and *unapply* methods can be found in Listing 2.1.

```
class FullName(first: String, last: String) // Class definition omitted

object FullName {
  // Creates a new instance
  def apply(first: String, last: String): FullName = new FullName(first, last)

  // Return value of unapply is always the Option class
  // Option has 2 instances: Some(value) and None
  def unapply(name: FullName): Option[(String, String)] = Some((name.first, name.last))
}

// Calls FullName.apply("Bob", "Miller")
val name = FullName("Bob", "Miller")

name match {
  // Calls FullName.unapply(name), assigns the result to first and last
  case FullName(first, last) => println("Last name is", last)
  case _ => println("Not a full name")
}
```

Listing 2.1: Scala apply-unapply example.

Case classes and case objects

Scala has case classes and case objects. Case classes are a shorthand to create a class with the given parameters as values. Getters, setters, *equals*, *hashcode*, *copy* and *toString* methods are automatically created. A companion object with *apply* and *unapply* methods is also automatically created. This makes case classes easy to use as data types. Case objects are similar to case classes, except they do not have values. There can only be a single instance of a (case) object [1].

Traits

Scala Traits are similar to interfaces or abstract classes in other OOP languages. Traits can contain variables, non-abstract and abstract methods. Traits cannot have a constructor or be instantiated and are multiple inheritance, which means they can extend multiple other traits. Traits can be mixed in with classes. When a trait is mixed in with a class, the class inherits the variables and methods of the trait, and abstract methods have to be implemented. Multiple traits can be mixed in with a single class. If two traits contain methods with the same name there is a naming collision, which has to be resolved in the class itself [52]. It is possible to specify that a type consist of multiple traits. These are called compound types [43]. An example of traits, mixins and compound types can be found in Listing 2.2.

```
// Extend Java cloneable interface
trait Cloneable extends java.lang.Cloneable{
  // Default implementation: call Java cloneable and cast result
  override def clone(): Cloneable = super.clone().asInstanceOf
}

trait Resetable {
  // Abstract method
  def reset(): Unit
}

// Class with multiple mixins
class Counter extends Cloneable with Resetable {
  var count = 0
  def inc(): Unit = count += 1

  // Implement resetable trait
  override def reset(): Unit = count = 0
}

// Method with compound type parameter
def cloneAndReset(obj: Cloneable with Resetable): Cloneable
```

Listing 2.2: Scala traits example.

Methods

Methods are another basic building block of the OOP paradigm. Classes, objects and traits in Scala can have methods, which can be public, protected or private. It is possible to override inherited methods and methods cannot be static. The return type of methods can be defined or inferred. For public methods, it is recommended to explicitly define the return type. Methods in traits or abstract classes do not need an implementation. Method parameters can have default values [43] and methods can be called with named arguments [43]. An example can be found in Listing 2.3.

```scala
// Define method with default values
def point(x: Double = 0, y: Double = 0, z: Double = 0): Point = new Point(x, y, z)

val point1 = point(1, 1, 1) // Regular call
val point2 = point() // Use default values
val point3 = point(z = 2, y = 1) // Use named parameters, x becomes the default value
```

Listing 2.3: Scala default parameters and named arguments example.

An important distinction from other OOP languages is that methods are implemented using expressions instead of *block* statements. Because of this, it is not necessary to use the *return* keyword, although this is still possible.

Nesting

Scala classes, object and trait definitions can all be nested. This means that it is possible to define classes within classes, objects within objects or traits within traits. It is also possible to mix these definitions (e.g., define a class within an object). Nested classes are called inner classes [43]. An instance of the outer definition is required to access the inner definitions. An example of this can be found in Listing 2.4.

```scala
class Outer {
  class Inner {
    def foo(x: Inner): Inner = x
  }
}

// We need an instance to access the inner class
val a = new Outer
val b = new Outer

// a.Inner and b.Inner are two different types
val aInner = new a.Inner
val bInner = new b.Inner

// Invalid, wrong type
aInner.foo(bInner)
```

Listing 2.4: Scala inner classes example.

Methods can also be nested. This means that it is possible to define a method within a method. Nested methods can only be accessed within the method they have been defined in.

Type parameterization

Type parameterization, also called generic types or polymorphism, can be added to classes, traits and methods. A type parameter is a type that can be specified later. This makes it possible to define, for example, a generic list (`List[A]`) which can be used for both integers (`List[Int]`), strings (`List[String]`) or any other object.

Type parameterization supports upper and lower type bounds. The upper type bound `T <: A` declares that type parameter `T` refers to a subtype of type `A` [33]. A lower type bound `T >: A` expresses that the type parameter `T` refers to a supertype of type `A` [33]. An example of upper and lower type bounds can be found in Listing 2.5.

```scala
// Upper type bound
// An animal container can also contain subtypes like Dogs
class AnimalContainer[T <: Animal]

class List[+T] {
  // Lower type bound
  // A List[Int] can be concatenated with supertypes like List[Number] (returns a List[Number])
  def concat[U >: T](other: List[U]): List[U]
}
```

Listing 2.5: Scala type bounds example.

By default, the subtyping of parameterized classes is invariant, meaning that `Class[A]` is only a subtype of `Class[B]` if `B = A`. This behavior can be changed with variance annotations. `Class[+A]` is a covariant class. This means `Class[B]` is a subtype of `Class[A]` if `B` is a subtype of `A`. `Class[-A]` is a contravariant class. Contravariance is the opposite of covariance. So if `Class[A]` is a subtype of `Class[B]` if `B` is a subtype of `A` [43]. An example of variance can be found in Listing 2.6.

```scala
// --- Example classes ---
abstract class Animal {
  def name: String
}
case class Cat(name: String) extends Animal
case class Dog(name: String) extends Animal

// --- Parameterized classes ---
class Container[A] {...} // Invariant
class List[+A] {...}     // Covariant
class Printer[-A] {...}  // Contravariant

// --- Covariance example ---
val cats: List[Cat] = List(Cat("Whiskers"), Cat("Tom"))
def printAnimalNames(animals: List[Animal]) ...
// Covariance, List[Cat] is an instance of List[Animal] because Cat extends Animal
printAnimalNames(cats)

// --- Contravariance example ---
def printMyCat(printer: Printer[Cat]): Unit = printer.print(myCat)
val animalPrinter: Printer[Animal] = (animal: Animal) => println("Animal name: " + animal.name)
// Contravariance, Printer[Animal] is an instance of Printer[Cat] because Cat extends Animal
printMyCat(animalPrinter)

// --- Invariance example ---
val catContainer: Container[Cat] = Container(Cat("Felix"))
// Not allowed, Container is invariant so Container[Cat] is not an instance of Container[Animal]
val animalContainer: Container[Animal] = catContainer
// Oops, we'd end up with a Dog assigned to a Cat
animalContainer.setValue(Dog("Spot"))
val cat: Cat = catContainer.getValue
```

Listing 2.6: Scala variance example.

In the invariance example, we see what could go wrong if `Container` would be covariant. To ensure type safety it is not possible to declare mutable covariant and contravariant types [34]. In the example case, that means it is not possible to define a mutable covariant container, so we cannot end up with a `Dog` assigned to a `Cat`.

2.2.3 FP constructs

In this section, the Scala constructs related to functions, pattern matching and list comprehension are discussed.

Functions

Functions are the basic building block of the FP paradigm. In Scala, function definitions return a *Function* object [35]. This object can be called, executing the function and returning the result. Like methods, the return type of a function can be inferred. Function objects can be assigned to a variable or passed to another function or method like any other object. Functions can be specified as a method parameter. This means it is possible to define higher-order functions [34]. Functions that are directly passed to another function are often called lambdas or anonymous functions.

Unlike methods, functions have no support for default parameters or named arguments [39]. Methods can automatically be converted to functions. However, converting functions to methods is not automatic and requires defining a new method. When a method is converted to a function, the ability to use named arguments or default values is lost. For an example of functions see Listing 2.7.

```
def myMethod(x: Int): Int = x * 2       // Define a method
val myFunction: Int => Int = x => x * 2 // Define a function and assign to variable
myMethod(2)   // Call a method
myFunction(2) // Call a function

def higherOrder(x: Int => Int): Int = x(2) // Takes a function object as parameter
higherOrder(x => x * 4) // Call with function directly
higherOrder(myFunction) // Call with previously defined function
higherOrder(myMethod)   // Methods can automatically be converted to functions
```

Listing 2.7: Scala functions example.

Currying

In Scala, it is possible to use currying. Currying is the process of transforming a function that takes multiple arguments into a function that takes a single argument and returns another function that accepts further arguments. By default, this requires the function or method to define multiple parameter lists. When calling this method or function with the first parameter list, it will return a function that can be called with the second parameter list [33]. Functions, even those with a single parameter list, can also be converted to curried variants that have a parameter list for each individual argument. They can also be converted (back) to tupled variants, which have a single parameter list [52]. To apply these conversions to methods they have to be converted to functions first, losing the ability to use named arguments and default values. For an example of currying see Listing 2.8.

```
val add1: (Int, Int) => Int = (x, y) => x + y // Regular function (single parameter list)
val add2: Int => Int => Int = x => y => x + y // Curried function (multiple parameter lists)
def add3(x: Int, y: Int): Int = x + y // Regular method (single parameter list, similar to add1)
def add4(x: Int)(y: Int): Int = x + y // Curried method (multiple parameter lists, similar to add2)

add1(2)    // Invalid, requires 2 arguments
add1(2,2)  // Valid, call uncurried function with both arguments
add2(2,2)  // Invalid, requires 1 argument at a time
add2(2)    // Valid, returns function that takes second argument
```

```scala
add2(2)(2) // Valid, call curried function with both arguments

val x: Int => Int = add2(2) // Call with first argument and assign resulting function
val result: Int = x(2)      // Call with second argument and get result

val add5: Int => Int => Int = add1.curried // Transform add1 into curried function (similar to add2)
```

Listing 2.8: Scala currying example.

Pattern matching

Scala supports pattern matching, which can be done based on value or type. During pattern matching, it is also possible to match or extract values for any class that has a companion object with an unapply method. It is also possible to add guards to patterns using if statements [43]. For an example of pattern matching see Listing 2.9.

```scala
case class FullName(first: String, last: String)

val x: Any = ???
x match {
  case 42 => println("Match by value")
  case FullName => println("Match by type")
  case FullName(_, x) => println("Match and unpack type, last name is", x)
  case FullName("Bob", last) => println("Match by type and value, last name is", last)
  case FullName(first, last) if first.length < last.length => println("Match with guard")
  case _ => println("Match everything else")
}
```

Listing 2.9: Scala pattern matching example.

Everything is an expression

In Scala, everything is an expression, like in FP languages. Scala does not require the use of the *return* keyword, since it uses the result of the expression will be used instead. In a *block* expression, the result is the result of the last expression in the block. In an *if* expression, the result is the result of the expression in the evaluated branch. In a *while* and a *do-while* expression, the result is always the *Unit* object, which is an empty singleton object [35]. An example can be found in Listing 2.10

```scala
def doSomething(): Boolean

def myMethod(): Int = { // Block expression, result is the last expression in the block
  val result = doSomething()
  if (result) { // Last expression in the method block, result is the evaluated branch
    42 // Last expression of the block expression in the top branch
  } else {
    -1 // Last expression of the block expression in the bottom branch
  }
}

// Same as above, but without blocks
def myMethod2(): Int = if (doSomething()) 42 else -1

// Valid, but assignment is useless since while always returns Unit
val x: Unit = while(doSomething()) myMethod()
```

Listing 2.10: Scala “everything is an expression” example.

Lazy evaluation

There are two main methods of evaluation in a programming language: strict evaluation and lazy evaluation. The former is often used in OOP languages, while the latter is often used in FP languages. With strict evaluation, variables and expressions are evaluated immediately. With lazy evaluation, variables and expressions are only evaluated when they are needed. This reduces unnecessary computations, makes it easier to use infinite data structures and makes it easier to parallelize code. However, the programmer can no longer rely on the execution order or if and when side-effects are triggered. This makes lazy evaluation unsuited for imperative programming, which explicitly specifies the control flow and relies on side-effects. By default, Scala is strictly evaluated. Scala does have support for lazy variable evaluation by prepending the *lazy* keyword to the variable definition and there is a lazily evaluated list available in the standard library.

2.2.4 MP constructs

In this section, the Scala constructs that are classified as both OOP and FP (MP) are discussed.

For-comprehensions

Scala for-comprehensions are a combination of for-loops in OOP languages and list-comprehensions in FP languages. For-comprehensions can be used to traverse collections. It is possible to traverse multiple collections or collections within collections with a single for-comprehension. It is also possible to assign variables and add guards with a for-comprehension. After defining the for-comprehension, either an expression follows or the *yield* keyword. The expression gets called every iteration, making it similar to a for-loop. The *yield* keyword returns a list, making it similar to a list comprehension [52]. An example can be found in Listing 2.11.

```
val matrix = List(
  List(0, 1),
  List(2, 3)
)

// Similar to a for-loop
val foo: Unit = for (
  x <- matrix; // x becomes a value from matrix
  y <- x; // y becomes a value from x
  if y > 0 // only call iteration if y is larger than 0
) {
  print(y) // prints y
  y // result of block becomes y, useless in for loop
}
// Prints 123, returns Unit

// Similar to list-comprehension
val bar: List[Int] = for (
  x <- matrix; // x becomes a value from matrix
  y <- x; // y becomes a value from x
  if y < 3 // only call iteration if y is smaller than 3
) yield {
  print(y) // prints y
  y // result of block becomes y, is added to resulting list
}
// prints 012, returns [0, 1, 2]
```

Listing 2.11: Scala for-comprehension example.

2.2.5 Other constructs

In this section, the Scala constructs that remain unclassified are discussed.

Operator overloading

In Scala, any arity 1 method can be used in infix notation. This means that methods with a single argument like `list.add(1)` can be written as `list add 1`. Operators themselves are also defined as methods within Scala. Because of this, operators can also be called in dot notation. For example, `1 + 1` can be written as `1.+(1)`. It is also allowed to use characters like + and - in method names, making it possible to override/overload operators. The precedence of infix operators is based on the first character[33].

Tuples

Scala has native support for tuples. They can be defined by putting multiple values between round brackets. They can be specified as a type for variables, arguments, functions and methods [43].

Annotations

Scala has annotations that can be used for meta-programming [33]. You can annotate classes, methods, fields, local variables and parameters [23]. One way annotations can be used is to ensure correctness. For example, *@tailrec* ensures the method is tail-recursive. Annotations can also be used to affect code generation. For example, *@inline* will attempt to inline methods. Furthermore, some constructs that are less commonly used keywords in Java have been implemented in Scala using annotations instead (e.g., *@volatile*, *@transient* and *@native*).

Implicit parameters

A method can define implicit parameters. The method can be called with or without these implicit parameters. If the method is called without them, the compiler attempts to get an implicit value for the parameter that matches the type. Implicit values are any variable or method that has been defined with the implicit keyword within the current scope. This can be combined with type parameterization to define different implicit values for different types [52]. For an example of implicit parameters see Listing 2.12.

```
abstract class Monoid[A] {
  def add(x: A, y: A): A
}
// Define implicit values
implicit val intMonoid: Monoid[Int] = (x: Int, y: Int) => x + y
implicit val stringMonoid: Monoid[String] = (x: String, y: String) => x concat y

// Method with implicit parameter
def sum[A](x: A, y: A)(implicit add: Monoid[A]): A = add.add(x, y)

sum(1, 2) // Uses intMonoid implicitly
sum(1, 2)(intMonoid) // Uses intMonoid explicitly
sum("Hello", "World") // Uses stringMonoid implicitly
sum(1.5, 3.5) // Compile error: no matching implicit in scope
```

Listing 2.12: Scala implicit parameter example.

Implicit conversions

Implicit conversions are methods that convert a value from type `A` to type `B`. If an expression does not conform to the expected type, or a member of a value is accessed that does not exist for that type of value, the Scala compiler checks whether there is an implicit conversion in scope that can convert the expression to the expected type or convert the value to a type that does have the accessed member [52]. The compiler only attempts direct conversions and never chains conversions. If there are multiple valid conversions, there is an ambiguity and the compiler throws an error [23]. For an example of implicit conversions see Listing 2.13.

```
// Converts Scala Int to Java Integer (part of Scala standard library)
implicit def int2Integer(x: Int): java.lang.Integer = java.lang.Integer.valueOf(x)

val javaList = new java.util.ArrayList[String]()
javaList.add(0, "Test") // Scala Int implicitly converted to Java Integer
```

Listing 2.13: Scala implicit conversion example.

2.2.6 Constructs overview

This section contains an overview of the identified constructs. The overview can be found in Table 2.1.

OOP constructs	FP constructs	MP constructs	Other constructs
Variables	Functions	For-comprehensions	Operator overloading
Classes	Currying		Tuples
Objects	Pattern matching		Annotations
Case classes/objects	Everything is an expression		Implicit parameters
Traits	Lazy evaluation		Implicit conversions
Methods			
Nesting			
Type parameterization			

Table 2.1: Identified Scala constructs.

2.2.7 Analysis

When analysing the combination of constructs in Scala, one of the first things that stands out is the high degree of similarity between methods and functions. They both serve the same purpose and the syntax is similar. Methods support additional features, like default parameter values and named arguments. In addition, a method can automatically be converted to a function when needed. Because of this, there seems to be no reason to use functions instead of methods, except when passing an anonymous function.

Another interesting difference between methods and functions is the return statement. The return statement is not needed in Scala, yet methods still support it whereas functions do not. Return statements within methods can be non-local. This means that it is possible to define a function within a method, use a return within the function, and it returns a value for the method instead. An example of this is shown in Listing 2.14. This behaviour is probably not intended by the developer.

```scala
def myMethod(): Int = {
  val innerFunction = () => {
    return -1 // Non-local return, will return value for myMethod() instead
  }
  innerFunction() // myMethod() returns -1 when innerFunction() is called
  0 // Return 0; this expression is never reached
}
```

Listing 2.14: Scala non-local return example.

If we compare the OOP constructs in Scala to those of pure OOP languages, most of the constructs remain unchanged. The main difference is that all statements in Scala are expressions. Block statements and while-loops have return values. Another difference is that traditional for-loops are not present in Scala. Scala only has for-comprehensions, which is a more extensive version of the for-each loop. Traditional for-loops can be emulated with a for-comprehension over a range of numbers.

If we compare the FP constructs in Scala to those of pure FP languages, more differences can be found. One of the most important differences is that functions have access to variables outside their scope. This means that functions can change the state of the program. Within pure FP languages, this is only possible through the use of monoids. Many of the higher-order functions within FP are based on the idea that functions do not change the state on the program. This makes it possible to change the execution order without changing the result, something that can not be relied upon within Scala.

Another difference is the type system. Scala's type system is based on the OOP type system where every type is an object and every object is a type. Type systems in pure FP languages are often defined using algebraic data types [26]. For example, a binary tree in the FP language Haskell can be defined as follows: `data Tree = Empty | Leaf Int | Node Tree Tree`. Scala includes constructs that make the OOP type system more suitable for FP. Case classes with support for pattern matching make it easier to use classes as algebraic data types. The advanced type interference of the compiler reduces the type signatures that are needed. This makes it easier to use complex types. Finally, the combination of everything is an object (including functions and tuples) and everything is an expression allow for a functional programming style.

The last difference is the evaluation method. FP languages often use lazy evaluation. By default, Scala is strictly evaluated, but it has some support for lazy evaluation (lazy variable evaluation and a lazily evaluated list type). However, most constructs and collections in the standard library use strict evaluation.

If we look at the other constructs that are present in Scala, there are two constructs that can only be found in Scala, namely implicit parameters and implicit conversions. These are powerful constructs that, because of their implicit nature, can be difficult to debug when used incorrectly. Furthermore, importing implicit parameters or conversions into scope can cause unwanted side-effects.

The differences presented in this analysis should be taken into account when using existing OOP/FP metrics or when defining metrics for Scala. The combination of higher-order function and changing the state of the program does not occur in either paradigm on their own. As discussed, non-local returns and implicit parameters/conversions can cause unexpected be-

haviour. In addition, being able to use block statements and while-loops as values could cause mistakes that go unnoticed during compilation. Finally, the behaviour of for-comprehensions changes based on a keyword in the middle of the expression (demonstrated by Listing 2.11). This makes them more difficult to read and understand. All of these differences can affect the fault-proneness.

2.3 Code quality

The ISO/IEC 25010:2011 [25] standard defines software product quality as the degree to which the system satisfies the stated and implied needs of its various stakeholders, and thus provides value. It defines eight quality characteristics:

- **Functional Suitability** Degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions.
- **Performance efficiency** Performance relative to the amount of resources used under stated conditions.
- **Compatibility** Degree to which a product, system or component can exchange information with other products, systems or components, and/or perform its required functions while sharing the same hardware or software environment.
- **Usability** Degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use.
- **Reliability** Degree to which a system, product or component performs specified functions under specified conditions for a specified period of time.
- **Security** Degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization.
- **Maintainability** Degree of effectiveness and efficiency with which a product or system can be modified to improve it, correct it or adapt it to changes in the environment, and in requirements.
- **Portability** Degree of effectiveness and efficiency with which a system, product or component can be transferred from one hardware, software or other operational or usage environment to another.

Code quality is strongly related to performance efficiency, reliability, security and maintainability. Source code metrics often target the measurement of reliability and maintainability. Therefore, we focused on those two aspects.

2.3.1 Maintainability

Software maintainability, as defined by the ISO/IEC 25010:2011 standard, is composed of the following sub-characteristics [25]:

- **Modularity** Degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components.

- **Reusability** Degree to which an asset can be used in more than one system, or to build other assets.
- **Analysability** Degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, or to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified.
- **Modifyability** Degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality.
- **Testability** Degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met.

The Software Improvement Group (SIG) is an organization that helps other organizations measure, evaluate and improve their software quality. They have defined a maintainability model that can be used to measure the maintainability of a software product[50]. This model defines the following measurements:

- **Volume** Overall size of the source code of the software product. Size is determined from the number of lines of code per programming language normalized with industry-average productivity factors for each programming language. Volume shall be rated on a scale that is independent of the type of software product.
- **Duplication** Degree of duplication in the source code of the software product. Duplication concerns the occurrence of identical fragments of source code in more than one place in the product.
- **Unit complexity** Degree of complexity in the units of the source code. The notion of unit corresponds to the smallest executable parts of source code, such as methods or functions.
- **Unit size** Size of the source code units in terms of the number of source code lines.
- **Unit interfacing** Size of the interfaces of the units of the source code in terms of the number of interface parameter declarations.
- **Module coupling** Coupling between modules in terms of the number of incoming dependencies for the modules of the source code. The notion of module corresponds to a grouping of related units.
- **Component balance** Size distribution of top-level components. The notion of top-level components corresponds to the first subdivision of the source code modules of a system into components, where a component is a grouping of source code modules.
- **Component independence** Percentage of code in modules that have no incoming dependencies from modules in other top-level components.
- **Component Entanglement** Percentage of communication between top-level components in the system that are part of commonly recognized architecture anti-patterns.

2.3.2 Reliability

Software reliability, as defined by the ISO/IEC 25010:2011 standard, is composed of the following sub-characteristics [25]:

- **Maturity** Degree to which a system, product or component meets its needs for reliability under normal operation.
- **Availability** Degree to which a system, product or component is operational and accessible when required for use.
- **Fault tolerance** Degree to which a system, product or component operates as intended despite the presence of hardware or software faults.
- **Recoverability** Degree to which, in the event of an interruption or a failure, a product or system can recover the data directly affected and re-establish the desired state of the system.

Chapter 3

Validation methodology

This chapter discusses the methodology used to validate metrics. In this thesis, two different methodologies are used. Section 3.1 describes Briand's validation methodology. Section 3.2 describes Landkroon's validation methodology. Section 3.3 describes how the measurements from both validation methodologies are used to predict fault-proneness. Finally, Section 3.4 describes how prediction performance is evaluated in this work.

3.1 Briand's validation methodology

A common method for metric validation is Braind's validation methodology [6], which investigates the relation between the internal attribute A_1 and the external attribute A_2. Briand's methodology relies on the following assumptions:

1. The internal attribute A_1 is related to the external attribute A_2
2. Measure X_1 measures the internal attribute A_1
3. Measure X_2 measures the external attribute A_2

The suitability of a metric is validated by evaluating how well it can be used as a predictor for fault-proneness. The fault-proneness is the likelihood a piece of software contains faults. Fault-proneness is commonly used to validate metrics and to give an indication of the code quality [6, 11]. Since the fault-proneness cannot be measured directly, it is estimated by counting the number of faults a piece of code contained during its lifetime.

Measure X_1 is the metric to validate. This measure quantifies defined attributes of the code. These measured attributes of the code are the internal attribute A_1. Measure X_2 is the fault-proneness measurement of the code. The fault-proneness is the external attribute A_2.

Measure X_1 measures the attributes of the latest version of the code. Measure X_2 measures the fault-proneness of the code, which is done over the entire lifecycle of the code. The relationship between A_1 and A_2 is investigated using a prediction model. This is done using regression analysis. The resulting performance of the prediction model, which indicates how well a metric can be used as indicator for fault-proneness, is evaluated to determine the suitability of the metric.

3.2 Landkroon's validation methodology

Landkroon has modified Briand's validation methodology [30], so that instead of measuring the metrics values of the latest version, metric values are measured each time the code contains a fault. Only the metric values of faulty pieces of code are measured. The metric values of all non-faulty pieces of code are still measured based on the latest version of the code. Landkroon's reasoning behind this modification is that in projects with longer lifecycles, classes can be refactored or rewritten and the metric values of the final version of the code are not indicative of the version of the code when it contained faults.

The main difference between Briand's and Landkroon's validation methodology, is what one wants to measure. For example, assume we want to validate a metric for fault-proneness. In this case, Briand's methodology measures whether the metric gives an indication for the fault-proneness of the final product. This is especially useful when assessing the quality of the final products. In contrast, Landkroon's methodology measures whether the metric gives an indication of the fault-proneness of the intermediate product. This is especially useful for detecting faults during the development process. Nowadays, software often does not have a final version anymore and is continuously being developed, i.e., continuous deployment is becoming more and more common. To ensure faults do not end up in production it is important that the checks before deployment detect potential faults. Therefore, we would argue that Landkroon's methodology is more suitable for modern-day software development.

3.3 Relating measurements to fault-proneness

Investigating the relationship between measurements and fault-proneness is done using a prediction model that predicts the fault-proneness based on the metric measurements [18]. This prediction model uses regression analysis. Generally logistic regression is used as regression model [4, 6]

Logistic regression is used to describe the relation between a dependent variable (response or outcome) and one or more independent variables (predictors) [24]. The dependent variable is dichotomous (e.g., true or false) [36]. The logistic model estimates the possibility of one of the values of the dependent variable using the independent variables. In our case, logistic regression can be used to predict the chance of code being faulty or not based on the metric values. Two different types of logistic regression are used:

Univariate logistic regression Univariate logistic regression uses only one independent variable and describes the relation between the independent and the dependent variable. Univariate regression is performed for each independent variable and is helpful to determine whether there is a relation between the independent and the dependent variable [4]. The prediction model is constructed using Equation 3.1. In this equation β_0 is a constant, X_1 is the independent variable and β_1 is the coefficient of the independent variable.

$$P(faulty = 1) = \frac{e^{\beta_0 + \beta_1 X_1}}{1 + e^{\beta_0 + \beta_1 X_1}} \tag{3.1}$$

Multivariate logistic regression Multivariate logistic regression predicts the dependent variable using multiple independent variables. Multivariate regression is helpful to evaluate the predictive capability of the metrics that have been assessed sufficiently significant in the univariate analysis [4]. It can be used to validate which metrics are the best predictors and whether a set

of metrics can be improved by adding other metrics. The prediction model is constructed using Equation 3.2.

$$P(faulty = 1) = \frac{e^{\beta_0+\beta_1 X_1+\beta_2 X_2+\cdots+\beta_n X_n}}{1 + e^{\beta_0+\beta_1 X_1+\beta_2 X_2+\cdots+\beta_n X_n}} \tag{3.2}$$

	Predicted negative	Predicted positive
Actual negative	TN (true negative)	FP (false positive)
Actual positive	FN (false negative)	TP (true positive)

Table 3.1: Example confusion matrix.

3.4 Prediction performance evaluation

Prediction models are commonly validated using cross-validation [47]. In our validation, stratified k-fold cross-validation with 10 folds is used. k-fold cross-validation randomly partitions the data into k equally sized subsamples. A single subsample is used for the test set and the remaining $k-1$ subsamples are used for the training set. This is repeated k times with each of the subsamples used exactly once as test set. The results of all the runs are aggregated to produce a single estimation. Stratified k-fold ensures the balance between faulty and non-faulty measurements is the same for each fold, because the ratio between faulty and non-faulty measurements is likely to be unbalanced. Weights are assigned to both classes while training the logistic regression. The weights are inversely proportional to the class frequencies. The resulting prediction model has several measures that can be used to evaluate the performance, goodness of fit and significance of the model:

Precision The precision is the ratio of true positives to the total amount of predicted positives [5]. The precision is 100% when a piece of code predicted as faulty is always faulty. The lower the precision the higher the odds that a piece of code predicted as faulty is actually non-faulty. With the values of Table 3.1, the equation is as follows:

$$Precision = \frac{TP}{TP + FP} \tag{3.3}$$

Recall The recall is the ratio of true positives to the total amount of actual positives [5]. The recall is 100% when a piece of faulty code will always be predicted as faulty. The lower the recall the higher the odds that a piece of faulty code is predicted as non-faulty. With the values of Table 3.1, the equation is as follows:

$$Recall = \frac{TP}{TP + FN} \tag{3.4}$$

Matthews correlation coefficient (MCC) There are several measures that can be used to summarise the performance of a binary classification model in a single number, such as accuracy, the F_1-score and Matthews correlation coefficient. MCC is a statistical rate that produces a high score only if the prediction obtained proper results in all of the four confusion matrix categories (see Table 3.1), proportionally to the size of those categories [10]. This measure returns a value between -1 and $+1$. A coefficient of $+1$ represents a perfect prediction, 0 no better than random prediction and -1 indicates total disagreement

between prediction and observation. When the dataset is unbalanced (the number of samples in one class is much larger than the number of samples in the other classes), accuracy and F_1-score cannot be considered reliable measures anymore, because they provide an over-optimistic estimation of the classifier ability on the majority class [17]. Since our data sets are likely to be unbalanced (more non-faulty than faulty code), we have opted to use MCC. The formula for MCC is as follows:

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{3.5}$$

Chapter 4

Implementation

This chapter discusses the implementation of our analysis framework. Section 4.1 describes the criteria for the data and which projects have been selected. Section 4.2 presents an overview of the framework design. Section 4.3 presents how the fault analysis is done. The code analysis is presented in Section 4.4. Section 4.5 discusses the validation workflow. Section 4.6 presents how the results are analysed. Finally, the benefits and drawbacks of the chosen approach are discussed in Section 4.7.

4.1 Data collection

Data are collected from open-source projects, which need to satisfy the following criteria:

Scala Since the metrics and code analysis are based around Scala, the projects need to be written (mainly) in Scala.

Version Control For the validation of metrics, previous versions of the code need to be accessible. The changes made between each version of the code should also be accessible. Therefore, version control is a prerequisite.

Issue tracker To gather faults we need to know which faults have been found and what changes fixed these faults. Therefore, an issue tracker that keeps track of the faults is a prerequisite. Since issue trackers are often also used to track feature requests and other info, it is a prerequisite that faults are labelled separately from the other issues. Additionally, there needs to be a link between the issue and the changes that resolved the issue.

Maturity There should be enough data available to analyse. Longer running projects are more likely to have found faults and have accumulated more fixes over time. This makes them more suitable for analysis, suitable projects should have at least 100 issues labelled as faulty.

Data-set balance The resulting data set should contain enough faulty code compared to the non-faulty code. If less than 1% of the code is faulty the results could be unreliable, since single incidents have a large influence on the results.

There are multiple version control systems and issue trackers available. Our implementation uses Git as version control system in combination with the GitHub Issue Tracker, since these are the most commonly used for open-source Scala projects.

4.1.1 Selected projects

For the validation of metrics data are needed. Projects have been selected using the official Scala Library Index [41], by filtering on the latest Scala version (2.13 at the time of writing) and then sorting by stars. The top 40 projects have been filtered based on the criteria described above. Finally, projects with less than 100 faulty issues, 2.5% faulty functions or 5% faulty objects have been filtered out to make sure enough data are available. The following projects have been selected:

Akka is an actor-based framework for building concurrent, reactive and distributed applications. At the time of writing, the project had 25,164 commits, 730 contributors and 734 closed issues labelled as fault. Data for this project were collected on the 2nd of August 2020 (url: `https://github.com/akka/akka`, branch: master, commit: 142a63f600af6b8b805a10f0f401a4615237be48)

Gitbucket is a Git web platform with GitHub API compatibility that can be privately hosted. At the time of writing, the project had 5,077 commits, 153 contributors and 311 closed issues labelled as fault. Data for this project were collected on the 2nd of August 2020 (url: `https://github.com/gitbucket/gitbucket`, branch: master, commit: 3534b7172d4b8ee439349817a20d73e20e960299)

Http4s is a Scala interface for HTTP services. At the time of writing, the project had 9,505 commits, 287 contributors and 153 closed issues labelled as fault. Data for this project were collected on the 6th of August 2020 (url: `https://github.com/http4s/http4s`, branch: master, commit: 9b588daa6d371d960ba19f8eb6a76a5ae21ea3ec)

Quill provides a domain specific language to express database queries in Scala. At the time of writing, the project had 2,777 commits, 85 contributors and 181 closed issues labelled as fault. Data for this project were collected on the 6th of August 2020 (url: `https://github.com/getquill/quill`, branch: master, commit: 7e7648d7337507083acb39d5f9248e523b165b63)

Scio is a Scala API for Apache Beam and Google Cloud Dataflow inspired by Apache Spark and Scalding. It is developed by Spotify. At the time of writing, the project had 3,754 commits, 103 contributors and 355 closed issues labelled as fault. Data for this project were collected on the 6th of August 2020 (url: `https://github.com/spotify/scio`, branch: master, commit: e71a3334f64d6b550ca38715e66ac75fa1205012)

Shapeless is a type class and dependent type based generic programming library for Scala. At the time of writing, the project had 2,344 commits, 119 contributors and 87 closed issues labelled as fault. Data for this project were collected on the 6th of August 2020 (url: `https://github.com/milessabin/shapeless`, branch: master, commit: 3d6ae8f0bb6bc428fd9bc8f926a77af4ab72cce8)

ZIO is a type-safe, composable library for async and concurrent programming in Scala. At the time of writing, the project had 4,378 commits, 317 contributors and 135 closed issues labelled as fault. Data for this project were collected on the 6th of August 2020 (url: `https://github.com/zio/zio`, branch: master, commit: 7aefe52a7f43475caa7e6ac9d905655b9b17fb6a)

4.2 Framework design

An overview of the framework design can be found in Figure 4.1. Each of the square blocks represents a module within the framework. The ellipses represent resources used by the framework. This section describes each of the modules and their responsibilities.

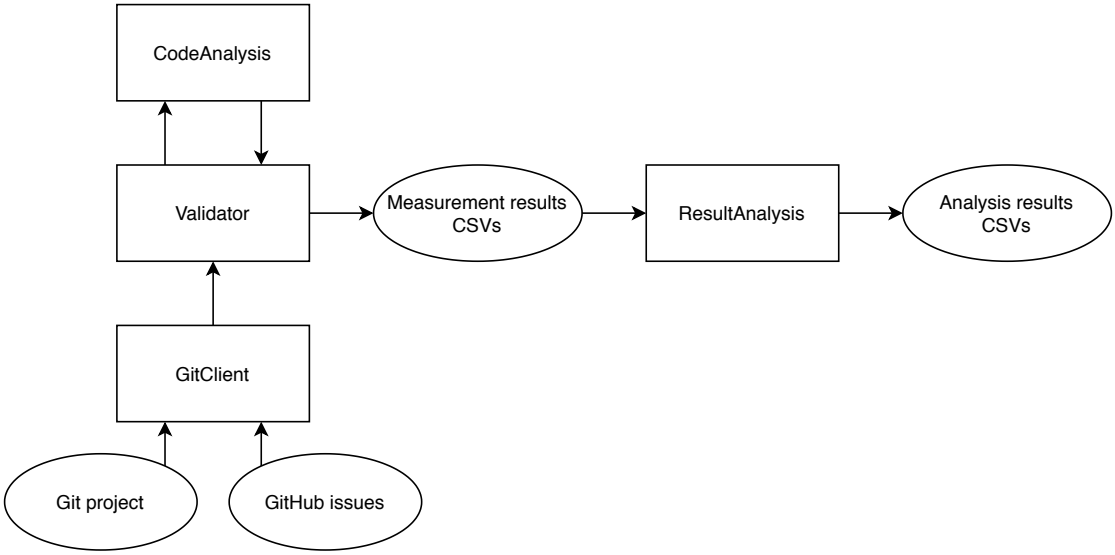


Figure 4.1: Framework design overview

GitClient The GitClient module is responsible for managing the Git project and its issues. The module can retrieve all commits that refer to faulty issues, it can calculate the changes between two versions and it can retrieve all files of a certain version of the code. The fault analysis to determine which commits refer to faulty issues is described in Section 4.3.

CodeAnalysis The CodeAnalysis module is responsible for analysing the code using metrics. Given a set of files, it can parse the code, run the metrics and return the results in a tree-like format based on the structure of the code. It contains all the metrics and the utilities needed to define them. The code analysis is described in Section 4.4.

Validator The Validator module is responsible for running the validation methodology workflow. It uses the GitClient module to retrieve files for analysis, getting the faulty commits and getting the changes made by those commits. Files for analysis are passed to the CodeAnalysis module, which returns the results back to the Validator module. The results are then processed and stored in CSV files. The validation workflow is described in Section 4.5.

ResultAnalysis The ResultAnalysis module is responsible for running logistic regression on the Validator results. It also includes functionality to calculate statistics of the Validator results. The statistics and logistic regression results are stored in CSV files. The result analysis is described in Section 4.6.

Algorithm 1: Validator pseudocode

```
Data:
The Git repository
List of fix commits
List of metrics
Result:
Briand's methodology results
Landkroon's methodology results
begin
    R ⟵ The Git repository;
    F ⟵ List of fix commits;
    M ⟵ List of metrics;
    latestResults ⟵ { };
    faultyResults ⟵ { };
    latestFiles ⟵ getLatestFiles(R);
    foreach file ∈ latestFiles do
        path ⟵ getPath(file);
        tree ⟵ getTree(file);
        resultTree ⟵ getMetricValues(tree, M);
        latestResults += (path, resultTree);
    end
    foreach f ∈ F do
        faults ⟵ getNumberOfFixes(f);
        changedFiles ⟵ getChangedFiles(f);
        // Only analyse files that exist in the latest version
        faultyFiles ⟵ {file ∈ changedFiles | containsPath(latestResults, file)};
        foreach file ∈ faultyFiles do
            path ⟵ getPath(file);
            tree ⟵ getTree(file);
            resultTree ⟵ getMetricValues(tree, M);
            latestResultTree ⟵ getLatestResult(path, latestResults);
            diff ⟵ getDiff(path, f);
            AddFaults(resultTree, latestResultTree, diff, faults);
        end
    end
    briandResults ⟵ lastestResults;
    landkroonResults ⟵ faultyResults ++ removeFaultyResults(latestResults);
    return (briandResults, landkroonResults)
end
Function AddFaults(tree, latestResult, diff, faults):
    if containsChanges(tree, diff) then
        tree.faults += faults;
        if existsInLatest(tree, latestResult) then
            latestTree ⟵ findInLatest(tree, latestResult);
            latestTree.faults += faults;
        end
        foreach child ∈ tree do
            AddFaults(child, lastestResult, diff, faults);
        end
    end
return
```

4.3 Fault analysis

Faults are counted by analysing the Git commits. Each issue and pull request within a project has a unique number. Commits can refer to these issues and pull requests by including the number with a # in front in their title or message. Pull requests can also refer to issues in the same way as commits.

When a commit refers to an issue and the issue is labelled as a fault, the commit is considered as a fix commit that has fixed a fault. When a commit refers to a pull request and the pull request refers to a faulty issue, the commit is also considered a fix commit. A commit or pull request can refer to multiple faulty issues and thus contain multiple fixes. The total number of fixes of a commit is the number of unique faulty issues it refers to. All analysed projects should refer to issues by referencing them in commits or pull requests. The result of the fault analysis is a list of fix commits combined with the total number of fixes of the commit.

4.4 Code analysis

The code analysis is done using the trees from the Scala compiler, which are an Abstract Syntax Tree (AST) representation of the code parsed by the compiler. First, all Java and Scala sources of the version to be analysed are parsed and loaded by the compiler. This is done to resolve imports. Next, each file to be analysed is typed and the resulting tree is retrieved. Since the library sources are not available, not everything can be typed. However, the compiler will generate the tree with all available information.

Metrics are run by traversing the tree. There are three types of metrics: file metrics, object metrics and method metrics. File metrics receive the top-level node, object metrics receive a module node (a module can be a class, trait or object) and method metrics receive a method node. The metric can then access all properties of the node or traverse the children of the node to calculate the metric values. The results have a similar structure to the trees. This means that each result contains sub-results for each object and method that existed within the tree that originated the result. The top-level results are always files.

4.5 Validator workflow

First, the latest version of the code is analysed and all the results are stored. Then, for each fix commit, the diff with the previous version within the current branch is calculated. If there are files affected by the diff that are also in the latest version, the version of the code before the commit (which still contains the faults) is analysed. Code is analysed per file, per object and per method. Each of these files, objects and methods can have faults. The changes of each fix commit are analysed. If the commit has modified a file, the numer of fixes the commit made are added to the fault count of the file. All the changes within the modified files are also checked. If the changes occur within an object or method, the number of fixes is added to their fault count. This means that if the fault occurs within a method of an object, the fault count of the method, the object and the file are all incremented with the number of fixes. The number of fixes is also added to the fault count of the results of the latest version if the file, object or method name matches.

After all fix commits have been analysed, two sets of results are created.

1. A set for Briand's methodology that contains the results of the latest version, including the fault counts that were added based on the changes of fix commits
2. A set for Landkroon's methodology that contains the results of all the faulty files from the fix commits (files with changes that were also in the latest version) combined with the results of the latest version for all the non-faulty files

The pseudocode of the validator can be found in Algorithm 1. The information for both sets of results is processed and written to CSV files. The CSV files contain information either per method, per object, or per file. In case of files and objects, all metrics that belong to child nodes (like methods) are summarised by calculating the average, sum and maximum per file or object.

4.6 Result analysis

The result analysis starts by calculating descriptive statistics, which includes information like minimum, maximum and average metric values. All statistics are then converted to a CSV and stored. Next, the results are analysed using logistic regression with stratified 10-fold cross-validation. First, the univariate logistic regression analysis is run for each individual metric. The prediction performance measures are calculated based on the resulting prediction. The measures of all metrics are combined in a single CSV and stored. Next, the multivariate logistic regression analysis is run for all the metrics combined. The resulting prediction performance measures are then stored. This process is repeated for each CSV file output by the validator, which outputs a separate file for each combination of methodology (Briand or Landkroon) and granularity (file, object or method).

4.7 Discussion

There were several options we could have chosen to represent the code. Initially, we used Landkroon's AST [30], which is created by converting compiler trees. During this process, some information is lost. While developing metrics the lost information was needed and it became easier to use the compiler trees directly instead of modifying Landkroon's implementation. Additionally, using compiler trees makes it easier to port metrics over to some of the existing Scala linting tools that also use compiler trees, like Scapegoat [40].

An advantage of using compiler trees is that a lot of information is available, like typing information. However, since not all sources (e.g., library or external sources) are always available, the typing information may be incomplete. Tools like Scapegoat circumvent this issue by being developed as compiler plugins and integrating directly into the build process. This works well for analysing the code during development. However, when analysing all fix commits of the code, this becomes difficult to integrate and time-consuming, since the entire build process has to be replicated for each version.

The use of compiler trees also means Scala code is desugared by the compiler and some information is lost. For example, for-expressions are translated to *filter*, *map* and *foreach* expressions and become indistinguishable from directly using these expressions.

It is possible that a file, object or method gets renamed. These renames are not detected by our implementation and the renamed unit is not associated with the latest version.

The fault-analysis assumes every mentioned issue in the main message of a pull request or commit has been fixed by that pull request or commit. Usually, this is the case. However, sometimes an issue gets mentioned in the main message for different reasons. Furthermore, sometimes the comments of a pull-request also mention issues that are fixed by the pull request. These are not taken into account since the comments also reference issues for different reasons and this occurs far more often than for the main message. Finally, when merging a pull request using a squash or merge commit, the commit always refers to the pull request. However, when using a fast-forward this is not the case. This means some pull request that fix faulty issues may not be taken into account because we cannot detect the associated commits. Fast-forwards are sometimes used to update the master branch when there is a separate development branch ahead of the master branch. Merging issue-fixing pull requests using fast-forwards is not common practice and does not occur in the selected projects. Therefore, this should not impact the analysis.

Chapter 5

Evaluating construct usage

This chapter evaluates the fault-proneness prediction performance of constructs in Scala and the paradigm scores defined within this chapter. The constructs measurements are defined in Section 5.1. The paradigm scores based on those measurements are defined in Section 5.2. Section 5.3 gives an overview of the projects and analyses the results of the construct measurements and paradigm scores. Finally, Section 5.4 concludes this chapter by answering **RQ1** and **RQ2**.

5.1 Construct measurement definitions

Scala combines OOP and FP by using OOP for the type system and using imperative and/or functional constructs for the program logic. When considering OOP as a paradigm separate from imperative programming, Scala becomes a combination between OOP, FP and imperative programming that uses OOP to structure the code and uses functional, imperative or a mix of both for the implementation of the program logic. In this sense, OOP is a neutral paradigm both present in functional style and imperative style code. For the constructs measurements and the paradigm score we have focused on the functional and imperative constructs, since the usage of those differs depending on the paradigm style used within the code.

The measured constructs are based on the constructs identified in Section 2.2. The constructs have been measured on a method-by-method basis. There are three categories of measurements: boolean, count and fraction measurements. Each category represents a different way of measuring the constructs and is explained in their own section. Each measurement is indicated by a number prefixed the first letter of the category they belong to followed by an F for functional constructs and an O for imperative constructs. To measure the construct usage within objects the sum, the average and the maximum of all methods has been used.

5.1.1 Boolean measurements

The boolean measurements indicate whether a construct is used or not. This results in a yes (1) or no (0) and is not affected by how often the construct is used. The overview of boolean measurements can be found in Table 5.1.

Nr.	Name	Description: Whether the method ...
BF1	IsRecursive	contains a call to itself
BF2	IsNested	is nested within another method

BF3	HasNestedMethods	has nested methods
BF4	HasFunctions	uses functions
BF4A	IsFunction	returns a function
BF4B	HasFunctionParameters	has parameters that are functions
BF4C	HasHigherOrderCalls	has calls that have a function as argument
BF4D	HasFunctionCalls	has calls to a function
BF4E	HasCurrying	has calls that return a function
BF5	HasPatternMatching	uses pattern matching
BF6	HasLazyValues	uses lazy values
BF7	HasMultipleParameterLists	has multiple parameter lists
BO1	HasVariables	uses (mutable) variables
BO1A	HasVariableDefinitions	defines variables
BO1B	HasInnerVariableAssignment	has assignments to variables defined within the method
BO1C	HasOuterVariableUsage	uses variables defined outside of the method
BO1D	HasOuterVariableAssignment	has assignments to variables defined outside the method
BO2	HasSideEffects	uses the *Unit*[1] type
BO2A	IsSideEffect	returns *Unit*
BO2B	HasSideEffectCalls	has calls that return *Unit*
BO2C	HasSideEffectFunctions	has functions that return *Unit*

Table 5.1: Boolean construct measurements.

5.1.2 Count measurements

The count measurements count how often constructs are used. This means that they are affected by method size and could have correlations with other size metrics like lines of code. The set of measures corresponds to the boolean measures, except for measures that cannot be counted more than once (e.g., whether the methods returns a function).

Nr.	Name	Description: The number of ...
CF1	CountRecursiveCalls	calls to itself this method has
CF2	CountNestedDepth	methods this method is nested within
CF3	CountNestedMethods	nested methods
CF4	CountFunctions	references to functions
CF4B	CountFunctionParameters	function parameters
CF4C	CountHigherOrderCalls	higher order calls
CF4D	CountFunctionCalls	calls to functions
CF4E	CountCurrying	calls that return a function
CF5	CountPatternMatching	pattern match expressions
CF6	CountLazyValues	references to lazy values
CF7	CountParameterLists	additional parameter lists (e.g. two parameter lists is one additional list)
CO1	CountVariables	references to (mutable) variables
CO1A	CountVariableDefinitions	variables this method defines
CO1B	CountInnerVariableAssignment	assignments to variables defined within this method
CO1C	CountOuterVariableUsage	references to variables defined outside this method (including assignments)
CO1D	CountOuterVariableAssignment	assignments to variables outside this method
CO2	CountSideEffects	references to the *Unit* type
CO2B	CountSideEffectCalls	calls that return *Unit*

[1]Since the *Unit* type represents an empty return value, the methods or functions returning *Unit* rely on side-effects instead.

C02C	CountSideEffectFunctions	functions returning *Unit*

Table 5.2: Count construct measurements.

5.1.3 Fraction measurements

Fraction measurements calculate the number of lines in which a construct occurs in a method, divided by the total number of lines of the method. Fraction measurements are not necessarily affected by method size like the count measurements. The set of measures corresponds to the count measures, except for measures that do not occur in the method body (e.g., parameter lists).

Nr.	Name	Description: The fraction of lines containing ...
CF1	FractionRecursiveCalls	recursive calls
CF3	FractionNestedMethods	nested method definitions
CF4	FractionFunctions	references to functions
CF4C	FractionHigherOrderCalls	higher order calls
CF4D	FractionFunctionCalls	calls to functions
CF4E	FractionCurrying	calls that return a function
CF5	FractionPatternMatching	pattern match expressions
CF6	FractionLazyValues	references to lazy values
C01	FractionVariables	references to (mutable) variables
C01A	FractionVariableDefinitions	variable definitions
C01B	FractionInnerVariableAssignment	assignments to variables defined within this method
C01C	FractionOuterVariableUsage	references to variables defined outside this method (including assignments)
C01D	FractionOuterVariableAssignment	assignments to variables outside this method
C02	FractionSideEffects	references to the *Unit* type
C02B	FractionSideEffectCalls	calls that return *Unit*
C02C	FractionSideEffectFunctions	functions returning *Unit*

Table 5.3: Fraction construct measurements.

5.2 Paradigm score definitions

The paradigm score indicates whether a method is written in an imperative style (e.g., using side-effects and mutable state) or a functional style (e.g. using compositions of functions). In Scala, OOP constructs like classes and inheritance are used for encapsulation and are an inherent part of the type system. This means that these constructs are used both when writing imperative and functional style code. Since these constructs are always needed in Scala, they do not belong to a certain style of code and are therefore not included in the paradigm score. Equation 5.1 shows how the paradigm score is calculated.

$$paradigm\ score = \frac{functional\ score - imperative\ score}{functional\ score + imperative\ score} \tag{5.1}$$

The functional score is the sum of all functional construct measurements. The imperative score is the sum of all imperative construct measurements. The paradigm score is the ratio between the functional score and the imperative score, with -1 being predominantly imperative and +1 being predominantly functional. The paradigm score is not balanced. This means that a score of 0 does not necessarily indicate a 50/50 mix. Nonetheless, the score does give an indication of

how mixed the coding styles are and can be used to compare projects with each other.

Three paradigm scores have been defined, one for each category: *ParadigmScoreBool*, *ParadigmScoreCount* and *ParadigmScoreFraction*. *ParadigmScoreBool* uses all boolean measurements to calculate the functional and imperative scores. The other two paradigm scores do the same for their category, except all excluded measures are covered by the boolean equivalent.

5.2.1 Comparison to Landkroon's paradigm score

Landkroon has also defined a paradigm score [30]. This paradigm score is included in the measurements under the name *ParadigmScoreLandkroon*. Landkroon's paradigm score is calculated using Equation 5.2.

$$paradigm\ score = \frac{functional\ score}{functional\ score + imperative\ score} \tag{5.2}$$

This yields a score between 0 and 1, with 0 being more imperative and 1 being more functional. The paradigm score is not balanced, so a score of 0.5 does not necessarily represent a 50/50 mix. The measurements used to calculate the functional (prefixed with F) and the imperative (prefixed with O) score can be found in Table 5.4.

Nr.	Name	Description	Equivalent
F1	Recursive	Whether the method is recursive	BF1
F2	Nested	Whether the method is nested	BF2
F3	HigherOrder	The number of higher-order parameters	CF4B
F4	FunctionalCalls	Counts the usage of *foldLeft*, *foldRight*, *fold*, *map*, *filter*, *count*, *exists*, *find* and pattern match expressions	-
O1	SideEffects	The number of references to (mutable) variables	CO1
O2	ImperativeCalls	Counts the usage of *while*, *do-while* and *foreach*	-

Table 5.4: Landkroon paradigm score measurements.

The *Equivalent* column denotes which of our measurements are equivalent to Landkroon's measurements. F4 has no exact equivalent. The combination of CF4C (*CountHigherOrderCalls*) and CF5 (*CountPatternMatching*) can be considered similar to F4. However, Landkroon counts a predefined set of higher-order methods, whereas our measurement counts higher-order methods based on their type signature.

```
1 def allPositive1(list: List[Int]): Boolean = list.forall(_ > 0)
2
3 def allPositive2(list: List[Int]): Boolean = {
4   var result = true
5   for (i <- list)
6     if (i <= 0)
7       result = false
8   result
9 }
```

Listing 5.1: Scala allPositive example.

This difference is illustrated by *allPositive1* in Listing 5.1. Because the call to *forall* is a higher-order method call, our measure includes it. Landkroon's measure does not, since *forall* is not part

of the set of predefined methods. Another difference can be seen when looking at the paradigm scores of *allPositive2*. This method earns imperative points in all paradigm score definitions. However, Landkroon does not make a distinction between a method which earns imperative points and a method without points. This means *allPostive1* and *allPositive2* both earn 0 points. We argue *allPositive2* is more imperative and, therefore, there should be a distinction between the two.

```
1 val list = ListBuffer[Int]()
2 def add(elem: Int): Unit =
3   list += elem
```

Listing 5.2: Scala side-effect example.

Measure `O2` has no exact equivalent either. It is similar to *CountSideEffects*, which counts occurrences of the *Unit* type. Since all predefined constructs in `O2` return the *Unit* type these are included in *CountSideEffects*. Ideally, everything with side-effects would be counted. However, it is difficult to determine whether a call has side-effects, especially since this information for external methods is often not available. Our approach is to assume that each method and function that returns the *Unit* type has side-effects. Since the *Unit* type represents an empty return value, the methods or functions returning *Unit* would be useless otherwise. Listing 5.2 contains the method *add* with a side-effect, where the += operation returns the *Unit* type. Therefore, this method earns imperative points in our measurements, but it earns no imperative points in Landkroon's measurements. Because of the side-effects, we argue it should.

5.3 Results

The results consist of three parts. First, an overview of the available data per project is presented and the paradigm scores are visualized. This gives an indication of the data available per project and a rough overview of the style they are written in. Next, the fault-proneness prediction performance of all measurements described in Section 5.1 have been analysed using regression analysis. This gives an indication of how fault-prone each construct is. Finally, the fault-proneness prediction performance of the paradigm score is evaluated.

5.3.1 Projects overview

Table 5.5 shows the amount of non-faulty and faulty methods available for each project. Briand's methodology gives an indication of the current project size, whereas Landkroon's methodology also takes faulty files of past version into account. As can be seen, Akka is by far the largest project and has the most data available in terms of absolute numbers. Gitbucket is the smallest project when using Briand's methodology, but the amount of analyzed methods is closer to the other projects when using Landkroon's methodology. The amount of faults available for each project differs significantly. When using Briand's methodology Quill has the lowest percentage of faulty methods (3.7%). However, when using Landkroon's methodology, Quill has the highest percentage of faulty methods (12.32%). When taking faulty files of past versions into account, Quill's non-faulty methods show the smallest relative increase, whereas the faulty methods show the largest relative increase. This causes the large difference in the percentage of faulty methods between methodologies. For Briand's methodology, ZIO has the highest percentage of faulty methods (11.87%). A large amount of data available or a high percentage of faulty code could improve the fault-proneness prediction performance of the measured constructs.

	Briand's methodology			Landkroon's methodology		
Project	Non-faulty	Faulty	% faulty	Non-faulty	Faulty	% faulty
Akka	17437	1228	6.58%	58755	2746	4.46%
Gitbucket	1006	97	8.79%	3093	181	5.53%
Http4s	3096	239	7.17%	4602	563	10.90%
Quill	2212	85	3.70%	2655	373	12.32%
Scio	2838	252	8.16%	7202	668	8.49%
Shapeless	3106	217	6.53%	7373	441	5.64%
ZIO	4341	585	11.87%	17803	1347	7.03%

Table 5.5: Available method data of analysed projects.

The paradigm scores of the analysed projects have been visualized to give an indication of the style of code of each project. The paradigm score obtained with fraction measurements is used for the visualizations, since it is the most fine-grained paradigm score. In this section, some examples will be discussed. The full overview of all visualizations can be found in Appendix A.

In Figure 5.1, the paradigm scores of Akka are visualized using histograms. The left histogram visualizes the paradigm scores of all methods and the right histogram the average paradigm scores of all objects. Each of the bars indicates the number of occurrences of paradigm scores between the left number (inclusive) and the right number (exclusive, except for the rightmost bar). The leftmost bar indicates the number of methods or objects which earned no points at all. These methods are often basic methods, for example, getters and setters, used in both imperative and functional style code. Therefore, we consider them as neutral. The bottom colour of the histogram indicates the paradigm scores of code that contained faults according to the fault-detection described in Section 4.3. The top colour indicates the paradigm score of code that did not contain faults. The percentage above each bar indicates the percentage of scores that are non-faulty.

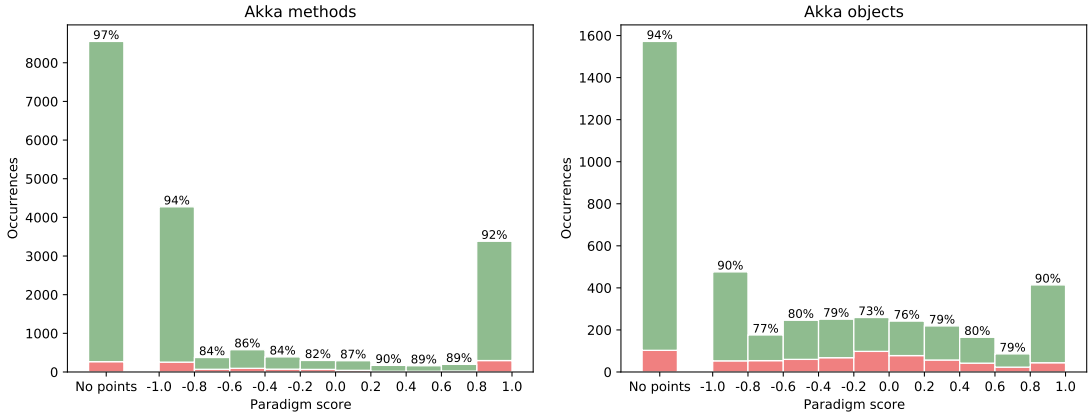


Figure 5.1: Akka paradigm score histogram.

In the case of Akka, most methods contain imperative or functional style code and methods that contain a mix of both styles are less common. However, objects do often contain a mix of both. Objects that contain a mix of both styles have a relatively higher number of faults than objects which mostly contain a single style. There are a lot of methods and objects that earned no points. Methods and objects that earned no points have a relatively low number of faults, even though the absolute number of faults is higher than the other categories.

The paradigm scores of Gitbucket are visualized in Figure 5.2. As can be seen, Gitbucket mostly uses functional style code. Out of all the analysed projects, Gitbucket has the relatively largest amount of methods that contain a mix of imperative and functional style code. There are slightly more methods that contain a mix of both styles than imperative style methods. Furthermore, mixing styles within objects is common. For the Gitbucket project, there is no clear correlation between the paradigm score and the percentage of faults, except that neutral methods are less likely to contain faults.

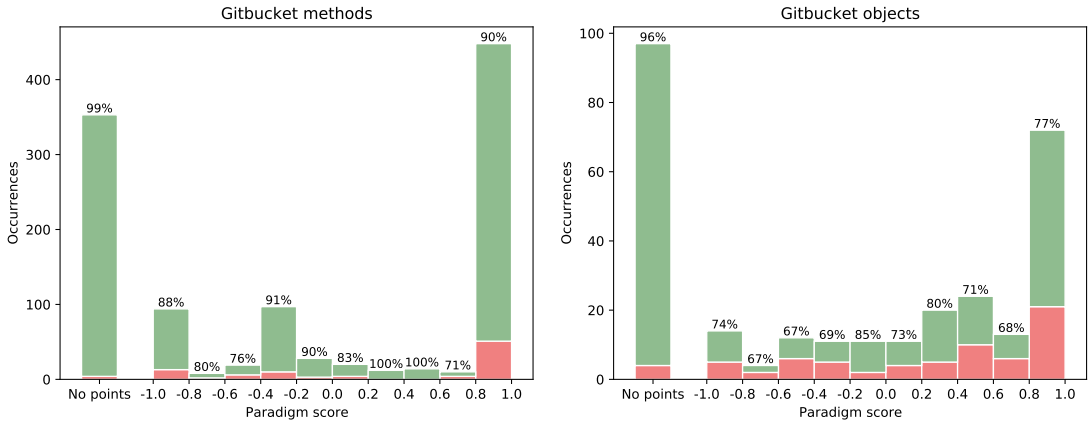


Figure 5.2: Gitbucket paradigm score histogram.

The paradigm score distribution of Http4s and Quill is very similar and mostly consist of functional style methods, with a few imperative style methods. The paradigm scores of Http4s are visualized in Figure 5.3. As can be seen, only a very small amount of methods contains mixed code. Mixed styles are slightly more common within objects, although most objects lean more towards functional style code. Shapeless also has a similar paradigm score distribution, except with an even stronger focus on functional style code and barely any imperative style code. Within Shapeless, objects containing mixed styles seem more likely to contain faults. However, there are not enough mixed style objects available to confirm this. For the other projects, there is no clear correlation between the paradigm score and the percentage of faults.

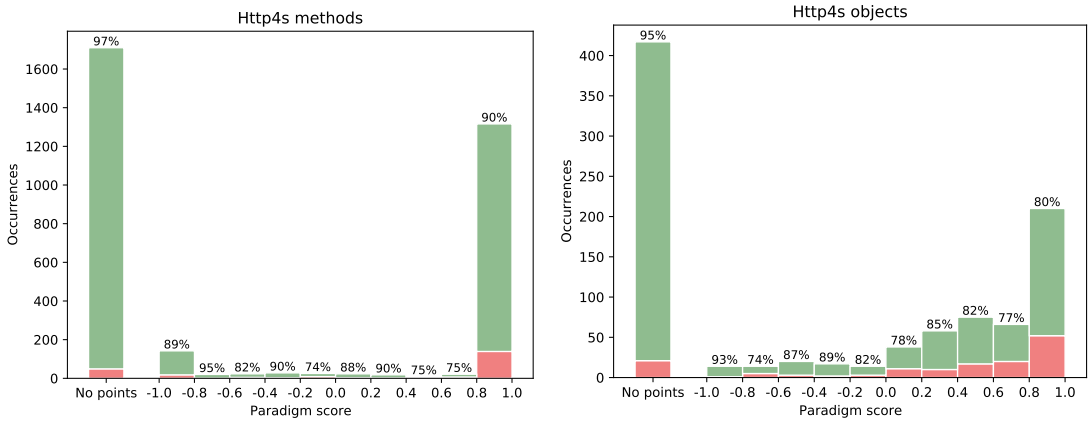


Figure 5.3: Http4s paradigm score histogram.

In Figure 5.4 the paradigm scores of Scio are visualized. Scio consists of mostly imperative and functional style methods and a mix of both styles within a method is uncommon. There are around twice as many functional style methods to imperative style methods. Although mixing styles within methods is uncommon, mixing styles within objects is common. Scio methods and objects that contain mostly functional style code with a little bit of imperative style code are slightly more likely to contain faults than other methods and objects.

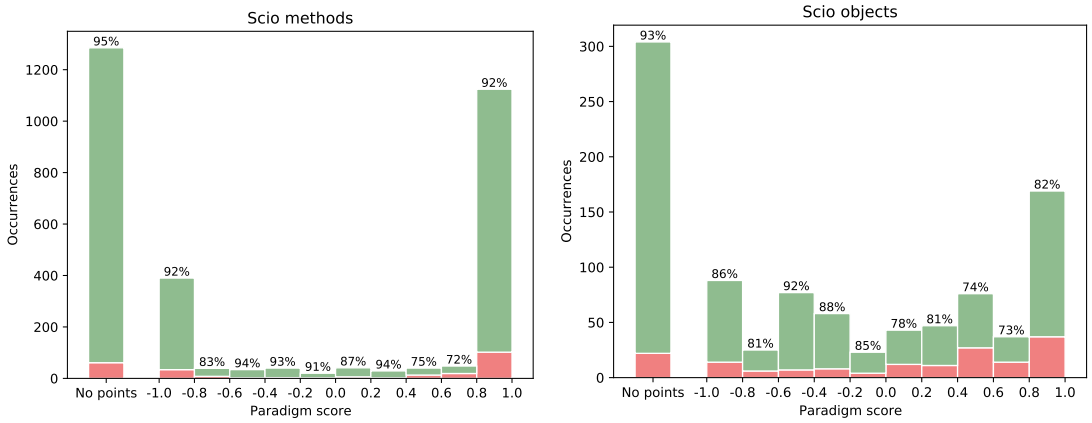


Figure 5.4: Scio paradigm score histogram.

The paradigm scores of ZIO are visualized in Figure 5.5. As can be seen, ZIO mostly has functional style methods with a small amount of imperative style methods. Mixing styles is not very common within methods, but is quite common within objects. However, there is still a large amount of objects that purely contain functional style code. Within ZIO, functional style methods are more likely to contain faults than other methods, although this does not hold for objects. Similar to Scio, objects that contain mostly functional style code with a little bit of imperative style code are slightly more likely to contain faults.

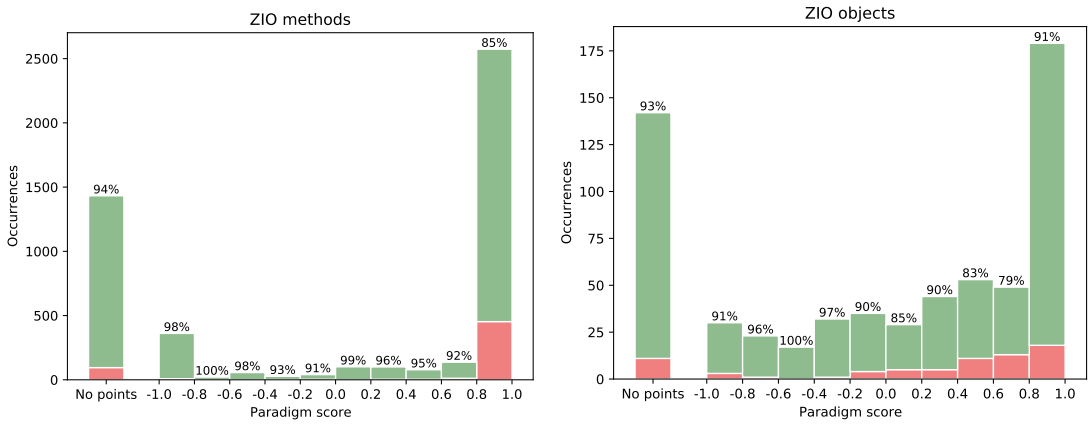


Figure 5.5: ZIO paradigm score histogram.

5.3.2 Construct measurement results

A summary of the construct measurement results can be found in Table 5.6. The results have been sorted by precision, which indicates the chances of a method being faulty if the measurement predicts it as faulty. The recall indicates the chances of a faulty method being predicted as faulty. The MCC is a measure of the overall performance. In this case, the goal is not to find a single measure that is able to detect every faulty method in the project. Instead, the goal is to find a measure that when it predicts a method is faulty, it is correct. This indicates that the measure is a reliable way to find (additional) faults and could potentially improve the baseline model. For each combination of category and methodology, the top 5 measures are shown in the table. The full overview can be found in Appendix B.

Name	Precision mean	Precision std.	Recall mean	Recall std.	MCC mean	MCC std.
Boolean construct measurements (Briand)						
HasOuterVariableUsage	18.19	15.09	48.34	48.41	0.063	0.040
HasVariables	17.95	8.20	22.86	34.43	0.070	0.042
HasVariableDefinitions	17.64	7.26	17.90	35.76	0.045	0.025
HasSideEffectFunctions	17.26	11.61	23.70	31.94	0.049	0.069
HasInnerVariableAssignment	16.64	6.85	18.34	35.55	0.047	0.028
Count construct measurements (Briand)						
CountOuterVariableUsage	18.19	15.09	48.34	48.41	0.063	0.040
CountVariables	17.95	8.20	22.86	34.43	0.070	0.042
CountVariableDefinitions	16.48	8.25	18.87	35.40	0.038	0.032
CountInnerVariableAssignment	16.46	7.00	7.06	6.03	0.039	0.037
CountSideEffectFunctions	16.20	12.18	20.81	30.86	0.045	0.071
Fractional construct measurements (Briand)						
FractionSideEffectFunctions	17.41	11.80	33.56	42.19	0.050	0.068
FractionOuterVariableUsage	16.49	15.49	60.88	47.19	0.044	0.059
FractionSideEffectCalls	14.92	2.88	31.02	31.00	0.094	0.028
FractionPatternMatching	14.84	4.44	29.01	7.60	0.112	0.044
FractionVariableDefinitions	14.64	8.40	30.79	42.55	0.027	0.040
Boolean construct measurements (Landkroon)						
HasPatternMatching	15.16	7.66	36.14	9.84	0.125	0.088
HasNestedMethods	14.88	8.49	13.80	6.40	0.065	0.045
HasVariableDefinitions	14.59	10.44	26.44	40.47	0.028	0.032
HasLazyValues	14.20	5.75	19.50	36.04	0.032	0.024
HasSideEffectFunctions	14.00	4.54	20.46	35.55	0.044	0.043
Count construct measurements (Landkroon)						
CountNestedMethods	14.88	8.49	13.80	6.40	0.065	0.045
CountPatternMatching	14.82	7.79	37.56	9.80	0.117	0.100
CountSideEffectFunctions	14.00	4.54	20.46	35.55	0.044	0.043
CountLazyValues	13.32	6.60	19.42	35.68	0.029	0.027
CountOuterVariableUsage	13.18	6.75	38.03	43.15	0.042	0.038
Fractional construct measurements (Landkroon)						
FractionNestedMethods	14.78	8.72	25.29	29.59	0.062	0.050
FractionSideEffectFunctions	14.00	4.56	20.39	35.57	0.043	0.043
FractionPatternMatching	13.75	5.53	30.95	7.04	0.098	0.055
FractionVariableDefinitions	12.66	10.22	56.85	49.10	0.019	0.038
FractionInnerVariableAssignment	10.67	7.56	56.41	42.82	0.011	0.035

Table 5.6: Construct measurements prediction performance summary.

For Briand's methodology, one of the top-performing measures is outer variable usage, which measures the usage of variables that have been defined outside the scope of the method. This

measure has a very high standard deviation. The recall of this measure is also highly project-dependent, so how well this measure performs depends on the project. The other variable measures also perform well and have a lower standard deviation. Finally, the side-effect functions measure, especially the one that measures the fraction of lines containing functions with side-effects, also performs relatively well.

For Landkroon's methodology, the results are quite different. In al three measurement categories, pattern matching is the best performing measure when considering both precision and recall. Nested methods and side-effect functions also belong to the top predictors.

The overall performance of the measurements as predictors for fault-proneness is not that good with the highest MCC score being 0.155 for Briand (FunctionalScoreCount) and 0.125 for Landkroon (HasPatternMatching). However, if a language construct would consistently cause faults, it would likely be seen as a flaw of the language and potentially be removed. Still, some of the constructs are somewhat indicative for faults whereas others are much less indicative. The differences between the categories of measurements are small and there are no measurements that perform significantly better in a single category.

5.3.3 Paradigm score results

The results for the paradigm scores can be found in Table 5.7. The differences between the paradigm scores are small. Our paradigm scores have a precision around 10% and a recall around 60%. The imperative and functional score better with a higher MCC. The imperative score has the highest precision, but the lowest recall. Landkroon's paradigm score has a higher precision overall, but a lower recall. None of the scores are a good predictor for fault-proneness.

5.4 Conclusion

The results presented in Section 5.3 brings us to the following answers to he research questions:

RQ1 *Which, if any, OOP or FP constructs in Scala are significantly more fault-prone than others?*

> Within Scala, OOP constructs that are not imperative can be considered neutral. The program logic in Scala is implemented using imperative and/or functional constructs. Within these imperative and functional constructs, there is not a single set of constructs that is significantly more fault-prone than others. However, there is still a difference between constructs. The most fault-prone constructs are (outer) variable usage, variable definitions, inner variable assignments, functions with side-effects, nested methods and pattern matching.

RQ2 *How well does the paradigm score perform as a predictor for fault-proneness?*

> The paradigm score on its own does not perform well as a predictor for fault-proneness. The precision of the different paradigm scores was around 10% and the recall around 60%.

Name	Precision mean	Precision std.	Recall mean	Recall std.	MCC mean	MCC std.
Briand's methodology						
ParadigmScoreBool	9.70	4.23	59.01	10.32	0.067	0.076
ImperativeScoreBool	14.36	3.72	39.58	26.79	0.106	0.012
FunctionalScoreBool	13.44	3.18	52.97	7.71	0.143	0.023
ParadigmScoreCount	10.13	3.77	58.83	13.52	0.087	0.053
ImperativeScoreCount	15.94	3.20	35.90	27.17	0.121	0.024
FunctionalScoreCount	14.76	3.87	47.53	6.84	0.155	0.032
ParadigmScoreFraction	10.09	4.09	64.10	11.42	0.084	0.079
ImperativeScoreFraction	11.70	3.04	36.82	25.92	0.077	0.040
FunctionalScoreFraction	11.69	3.35	55.36	5.25	0.115	0.028
ParadigmScoreLandkroon	13.11	3.37	50.82	12.94	0.132	0.025
ImperativeScoreLandkroon	14.65	4.71	23.32	31.60	0.062	0.052
FunctionalScoreLandkroon	13.93	3.03	44.86	8.11	0.137	0.021
Landkroon's methodology						
ParadigmScoreBool	9.79	4.74	59.45	9.28	0.062	0.065
ImperativeScoreBool	11.06	4.59	45.89	28.77	0.052	0.051
FunctionalScoreBool	11.71	4.69	54.23	10.19	0.106	0.049
ParadigmScoreCount	10.05	4.35	59.88	9.75	0.075	0.047
ImperativeScoreCount	12.13	4.26	34.17	28.82	0.064	0.053
FunctionalScoreCount	12.25	4.82	40.17	8.78	0.097	0.071
ParadigmScoreFraction	10.11	4.99	62.60	10.15	0.075	0.072
ImperativeScoreFraction	9.57	4.34	66.26	28.96	0.026	0.063
FunctionalScoreFraction	10.97	4.25	52.64	6.53	0.091	0.041
ParadigmScoreLandkroon	11.59	4.36	48.34	10.46	0.093	0.064
ImperativeScoreLandkroon	11.92	5.79	29.63	32.02	0.048	0.052
FunctionalScoreLandkroon	12.29	3.71	44.14	7.56	0.103	0.064

Table 5.7: Paradigm scores prediction performance.

Chapter 6

Baseline model

This chapter introduces the model that serves as a baseline for fault-proneness prediction performance. Section 6.1 defines the baseline model and its metrics. Section 6.2 evaluates the fault-proneness prediction performance of the baseline model. Section 6.3 assesses whether the performance of the baseline metrics is affected by the paradigm score. Finally, Section 6.4 concludes this chapter by answering **RQ3**.

6.1 Baseline model definition

The baseline model has been defined to ensure the researched MP metrics offer an improvement over the existing metrics. The baseline model consists of commonly used OOP and FP metrics. To ensure the researched MP metrics are an improvement over existing metrics, the fault-proneness prediction performance of the baseline model combined with one or more MP metrics should be significantly better than fault-proneness prediction performance of the baseline model on its own. This section defines the baseline model and its metrics.

6.1.1 General metrics

The general metrics are commonly used metrics related to the lines of code and not tailored to a specific paradigm. This section describes the general metrics used within the baseline model.

Lines of Code (LOC)

The Lines of Code is the total number of lines of a class, including empty lines and comments. The assumption is the larger the class, the more complex and the more likely it is to contain faults. This does not take into account the complexity of the code itself, but does yield a general indication.

Source Lines of Code (SLOC)

The Source Lines of Code is the total number of lines of a class, excluding blank and comment lines. The complexity of a class is usually related to the code and not to the white space and comments. Therefore, this metric usually gives a more accurate representation of the complexity of the class.

Comment Lines of Code (CLOC)

The Comment Lines of Code is the total number of comment lines of a class. The assumption is the more comments are needed to explain a class (relative to the other classes), the more complex the code is.

Comment Density (CD)

The Comment Density is the ratio of comment lines to source code lines. CD can be used as an indicator for software quality [2]. Using Equation 6.1, the value will be 1 if all the lines are comments and 0 if all the lines are source code.

$$CD = \frac{CLOC}{CLOC + SLOC} \tag{6.1}$$

6.1.2 OOP metrics

The OOP metric suite used for the baseline model is defined by Chidamber and Kemerer [11]. This metric suite is often validated and cited in the field of OOP metrics. Some of the metrics have been superseded by newer alternatives. This section describes which metrics are used in the baseline model.

Cyclomatic Complexity (CC)

Cyclomatic complexity, also known as McCabe complexity, indicates the complexity of a piece of code by measuring the number of linearly independent paths through the source code [31]. The cyclomatic complexity is computed by transforming the program to a control flow graph with N being the number of nodes and E being the number of edges. The cyclomatic complexity for a single program is $E - N + 2$. The idea is the more complex a piece of code is, the more fault-prone it is. In the baseline model, the cyclomatic complexity is calculated per method.

A disadvantage of the cyclomatic complexity is that it does not always translate well to programs written in a functional style. This is demonstrated by the two methods shown in Listing 6.1. Here the complexity of the second method is abstracted to the higher-order methods and the cyclomatic complexity becomes 1. However, the behaviour of the code is still just as complex as the first method.

```
// Cyclomatic complexity: 3
for (i <- list) {
  if (i > 0) {
    println(i)
  }
}

// Cyclomatic complexity: 1
list.filter(_ > 0).foreach(println)
```

Listing 6.1: Scala cyclomatic complexity example.

Weighted Methods per Class (WMC)

The Weighted Methods per Class is the number of methods of a class, weighted by the complexity of the methods. It is calculated by summing the complexity of the methods in the class [11]. If all methods are considered equally complex, the WMC is the number of methods of the class [4]. The definition by Chidamber and Kemerer does not explicitly define how the complexity should be computed, but suggests the Cyclomatic Complexity as measure. In the baseline metric suite, we used Cyclomatic Complexity. The assumption is the larger and more complex a class, the more fault-prone.

Depth of Inheritance Tree (DIT)

The Depth of Inheritance is the distance to the root class in the inheritance tree [11]. The assumption is the deeper the class is in the inheritance tree, the more definitions the class has inherited from ancestors, and thus the more fault-prone the class is [4].

Number of Children (NOC)

The Number of Children is the number of direct descendants the class has in the inheritance tree [11]. The assumption is the more direct descendants a class has, the more difficult it would be to modify, and therefore that the class would be more fault-prone [4].

Coupling Between Objects (CBO)

Objects are considered coupled when they use instances, variables or methods of another object. The Coupling Between Objects is the number of other objects to which an object is coupled [11]. The CBO can be split up in outgoing coupling (fan-out) and ingoing coupling (fan-in) [32]. This gives a more fine-grained result. In the baseline model, both CBO and fan-in/out are included. The assumption is that highly coupled objects are more fault-prone, due to inter-object activities [4]. Highly coupled objects can also indicate a weakness in module encapsulation [48].

Response for a Class (RFC)

The Response for a Class is the number of methods that can be invoked as a response to a message received by a class [11]. This includes methods that are called by methods of this class. The RFC is calculated using Equation 6.2.

$$RFC = M \cup \{\forall_x \in M \mid R_x\} \tag{6.2}$$

Where M is the set of methods of the class and R_x the set of methods called by method x.

Lack of Cohesion in Methods (LCOM)

Lack of Cohesion in Methods measures the (lack of) cohesion of a class. If a class is incohesive, it should be split up into multiple classes. Hitz and Montazeri have demonstrated that the theoretical definition of LCOM by Chidamber and Kemerer is inadequate [21]. It is possible to have non-cohesive methods and still attain an LCOM score of 0 (0 means the methods are cohesive). There are several other LCOM definitions. The most well known are the two definitions by Henderson-Sellers, Constantine, and Graham [20] and the definition by Hitz and Montazeri [22].

In the baseline model, we used the LCOM by Hitz and Montazeri, since it is the most commonly used LCOM metric. This definition consists of counting the number of connected components of

a class. A connected component consists of related class methods. Methods are related if they both use the same class-level variable or one of the methods calls the other method. If a class has multiple connected components, it can likely be split up into multiple classes.

6.1.3 FP metrics

The FP metrics for the baseline model are based on the work of Ryder and Thompson [37, 38]. Some of these metrics have been validated for Scala by Landkroon [30]. The FP metrics used in the baseline model are described below.

Pattern Size (PSIZ)

The pattern size measures the size of the pattern [38]. This is based on Haskell, where every function is a pattern. In Scala, there are two types of constructs that could be considered patterns: pattern matching and method overloading. For the baseline model, we have opted to only consider pattern matching. The pattern size is based on the number of AST nodes in the pattern.

Number of Pattern Variables (NPVS)

The number of pattern variables measures the number of variables in the pattern [38]. In our case, this becomes the number of variables bound in a match expression. The assumption is that the more variables are introduced, the more fault-prone the code is.

Depth of Nesting (DON)

The depth of nesting measures the maximum depth of the AST in a pattern [38]. The assumption is the higher the depth, the more complex the code, and therefore the more fault-prone it is.

Outdegree (OUTD)

The outdegree measures the number of methods called by the method [38]. There are two variants: the total number of methods and the number of unique methods. The assumption is that the higher the outdegree, the more likely a method is affected by changes in other methods, the more fault-prone it is.

6.2 Baseline performance

The fault-proneness prediction performance of the baseline model has been measured on the projects described in Section 5.3.1. The measurements have been done on an object-by-object basis. The method-based metrics have been summarised per object. This has been done using three different summarising strategies: the average, the sum and the maximum of all method metrics.

The results for the multivariate regression, which uses all metrics, have been collected for each summarising strategy. For the univariate regression, the results of the object-based metrics have been collected once and the results of the method-based metrics have been collected for each summarising strategy. The results for the baseline model using Briand's methodology can be found in Table 6.1 and using Landkroon's methodology in Table 6.2.

Overall, the baseline model scores fairly well with a precision around 31%, recall around 61% and MCC around 0.34 for Briand's methodology, and a precision around 42%, recall around 63% and MCC around 0.41 for Landkroon's methodology. However, there is still room for improvement. Of each summarising strategy, the sum of the method metrics performs the best. The depth of the inheritance tree, number of children, and fan-in metrics do not perform well on the selected projects.

Name	Precision mean	Precision std.	Recall mean	Recall std.	MCC mean	MCC std.
Multivariate regression						
Method average	31.32	8.01	62.71	7.63	0.347	0.060
Method sum	31.34	8.00	62.23	7.64	0.346	0.066
Method max	31.34	7.68	61.90	7.50	0.346	0.058
Object metrics						
LinesOfCode	34.71	10.56	53.37	6.25	0.342	0.077
SourceLinesOfCode	33.78	10.62	52.69	7.00	0.331	0.080
CommentLinesOfCode	33.15	11.81	33.80	11.06	0.254	0.105
CommentDensity	24.80	10.59	45.54	19.24	0.185	0.113
WeightedMethodsPerClass	30.39	8.91	49.37	6.22	0.291	0.069
DepthOfInheritanceTree	12.11	4.04	60.02	14.05	0.051	0.090
NumberOfChildren	13.43	5.33	34.61	23.87	0.042	0.045
CouplingBetweenObjects	28.62	8.57	57.99	4.83	0.303	0.074
FanIn	14.50	4.07	34.18	18.24	0.069	0.036
FanOut	31.29	8.52	60.57	5.46	0.339	0.073
ResponseForClass	30.85	7.92	56.36	3.31	0.320	0.051
LackOfCohesionInMethods	20.13	8.77	43.08	16.28	0.160	0.110
Method metrics average						
CyclomaticComplexity	25.28	8.94	50.47	10.47	0.244	0.100
PatternSize	27.31	13.37	32.51	9.60	0.201	0.122
NumberOfPatternVariables	28.42	13.31	32.51	7.99	0.208	0.112
OutDegree	24.26	9.90	50.62	6.31	0.232	0.096
OutDegreeDistinct	22.81	9.03	57.11	4.92	0.232	0.086
DepthOfNesting	28.46	12.15	37.42	7.50	0.229	0.099
Method metrics sum						
CyclomaticComplexity	30.51	8.75	49.82	5.58	0.293	0.065
PatternSize	32.38	12.58	36.36	9.17	0.259	0.107
NumberOfPatternVariables	33.36	12.35	36.19	8.07	0.265	0.099
OutDegree	32.72	11.06	47.66	7.50	0.306	0.092
OutDegreeDistinct	30.98	10.15	52.29	3.88	0.305	0.076
DepthOfNesting	32.00	11.90	40.50	9.58	0.273	0.105
Method metrics max						
CyclomaticComplexity	27.85	10.11	48.84	8.37	0.264	0.095
PatternSize	29.59	11.81	39.69	8.78	0.250	0.105
NumberOfPatternVariables	30.59	11.69	39.19	10.09	0.257	0.107
OutDegree	28.72	11.31	50.19	9.09	0.278	0.108
OutDegreeDistinct	25.22	10.48	54.72	7.15	0.253	0.100
DepthOfNesting	30.12	11.58	44.89	9.74	0.270	0.098

Table 6.1: Baseline model fault-proneness prediction performance (Briand).

Name	Precision mean	Precision std.	Recall mean	Recall std.	MCC mean	MCC std.
Multivariate regression						
Method average	41.09	15.75	64.30	10.09	0.410	0.146
Method sum	42.34	15.63	62.62	9.55	0.415	0.136
Method max	41.43	15.90	63.74	9.63	0.411	0.144
Object metrics						
LinesOfCode	47.39	13.73	53.66	13.30	0.416	0.126
SourceLinesOfCode	46.51	15.03	53.70	12.66	0.410	0.133
CommentLinesOfCode	45.90	19.85	47.35	20.55	0.321	0.194
CommentDensity	34.11	13.49	50.21	19.32	0.252	0.159
WeightedMethodsPerClass	40.49	13.74	48.96	11.33	0.341	0.128
DepthOfInheritanceTree	17.78	6.09	62.72	9.64	0.107	0.050
NumberOfChildren	21.34	8.84	42.68	28.21	0.090	0.085
CouplingBetweenObjects	38.59	12.63	54.62	11.16	0.346	0.119
FanIn	21.83	8.41	40.64	24.83	0.100	0.077
FanOut	40.69	12.41	57.77	11.07	0.378	0.101
ResponseForClass	41.37	12.55	53.79	11.51	0.371	0.124
LackOfCohesionInMethods	30.43	14.97	48.65	13.98	0.234	0.144
Method metrics average						
CyclomaticComplexity	30.26	11.33	53.71	11.85	0.269	0.115
PatternSize	31.68	11.38	34.53	10.35	0.216	0.113
NumberOfPatternVariables	33.57	12.43	35.91	8.86	0.235	0.108
OutDegree	32.41	11.98	52.77	8.78	0.286	0.113
OutDegreeDistinct	28.34	9.02	58.14	7.35	0.261	0.079
DepthOfNesting	34.46	12.81	41.07	9.66	0.263	0.115
Method metrics sum						
CyclomaticComplexity	40.49	13.66	48.90	11.25	0.341	0.128
PatternSize	40.90	11.27	39.47	11.03	0.308	0.112
NumberOfPatternVariables	39.05	12.64	40.77	11.76	0.299	0.126
OutDegree	43.62	12.97	48.83	12.05	0.368	0.124
OutDegreeDistinct	41.33	12.64	51.04	11.68	0.358	0.122
DepthOfNesting	38.13	12.63	46.71	11.70	0.316	0.127
Method metrics max						
CyclomaticComplexity	34.18	12.07	52.15	11.32	0.301	0.117
PatternSize	36.95	11.55	44.32	12.18	0.298	0.124
NumberOfPatternVariables	36.88	14.05	42.72	9.81	0.287	0.127
OutDegree	37.37	13.77	50.16	11.46	0.323	0.132
OutDegreeDistinct	32.79	11.84	55.05	7.91	0.296	0.108
DepthOfNesting	35.82	12.20	49.94	10.48	0.309	0.121

Table 6.2: Baseline model fault-proneness prediction performance (Landkroon).

6.3 Metric performance by paradigm

To determine to what extent the baseline model metrics are affected by the mix of OOP and FP, the code was split up into four categories based on the paradigm score: OOP, FP, Mix of OOP and FP, and Neutral. All objects that did not earn any paradigm score points were considered neutral. Of the objects that did earn points, objects with a paradigm score below or equal to -0.8 were considered OOP, above or equal to 0.8 were considered FP, and objects with a paradigm score in between were considered mixed. This has been done for each project. Some projects had less than 10 faulty objects in one of the categories, usually the OOP category. Due to the low number of faulty cases, individual instances would have a large effect on the prediction performance causing it to be unreliable. Additionally, when using 10-fold cross-validation it is

not possible to have a faulty object in each split. Therefore, these projects have been excluded from the results. To keep the comparison between paradigms fair, these projects have also been excluded from the categories for which they did have enough results.

For each split, the average MCC per metric has been plotted in a barchart. This measure returns a value between -1 and $+1$. A coefficient of $+1$ represents a perfect prediction, 0 no better than random prediction and -1 indicates total disagreement between prediction and observation. A barchart has been plotted for each methodology and summarising strategy. For each methodology, the barchart with the best overall scoring summarising strategy is discussed. The complete overview can be found in Appendix C.

For Briand's methodology, the sum summarising strategy has the best overall performance. All 7 projects had enough data in the Neutral, FP and mixed categories. However, only Akka and Scio had enough data in the OOP category. This means that the results used for the comparison are only based on two projects and, therefore, less reliable. A reason for the low amount of OOP objects could be that Scala is often chosen over more traditional languages if FP features are desired. Furthermore, the compiler often translates OOP constructs to functional variants and the Scala standard library promotes a functional programming style. For example, the standard collections in Scala are immutable and use higher-order methods to use or transform the data. The percentage of objects that contain faults per paradigm is shown in Table 6.3. Neutral objects are the most common and are less likely to be faulty. OOP and FP objects are the least common and are more likely to be faulty. Mixed objects are more common than OOP and FP objects and are the most likely to be faulty. This could be related to source lines of code since neutral objects are often shorter and mixed objects are often longer. Another explanation could be that it is more likely mixed objects are dealing with multiple concerns.

Paradigm	Relative size	Faulty% mean	Faulty% std.	SLOC mean	SLOC std.
Neutral	60.14%	6.12%	1.51%	7.83	1.58
OOP	6.36%	13.54%	3.34%	19.41	3.95
FP	8.81%	16.26%	7.97%	35.46	24.92
Mix	24.70%	26.16%	4.39%	51.89	13.77

Table 6.3: Average percentage of faulty objects and average SLOC per paradigm using Briand's methodology.

The average MCC per metric for Briand's methodology can be found in Figure 6.1. The *All* category represents the performance of the metric without splitting by paradigm. Most metrics perform worse on neutral objects compared to the other paradigms, with the exception of comment based metrics. The best performing metrics for neutral objects are the metrics related to the lines of code, with LOC being the most indicative metric.

Most metrics perform worse on OOP-style objects compared to FP or mixed objects. Metrics that score significantly worse include cyclomatic complexity, (source) lines of code, outdegree (distinct), response for class, and weighted method count. This is surprising since cyclomatic complexity, response for class, and weighted method count are OOP metrics. The cyclomatic complexity is measured per method and the sum of the methods is used to calculate the score per object. This makes cyclomatic complexity identical to the weighted method count. The cyclo-

matic complexity metric performs significantly better for OOP and worse for FP when using the maximum per object instead of the sum. The lack of cohesion in methods metric scores well in the OOP category. However, it does not perform well in the other categories. The most indicative metrics for OOP-style objects are the coupling between objects and the fan-out metrics.

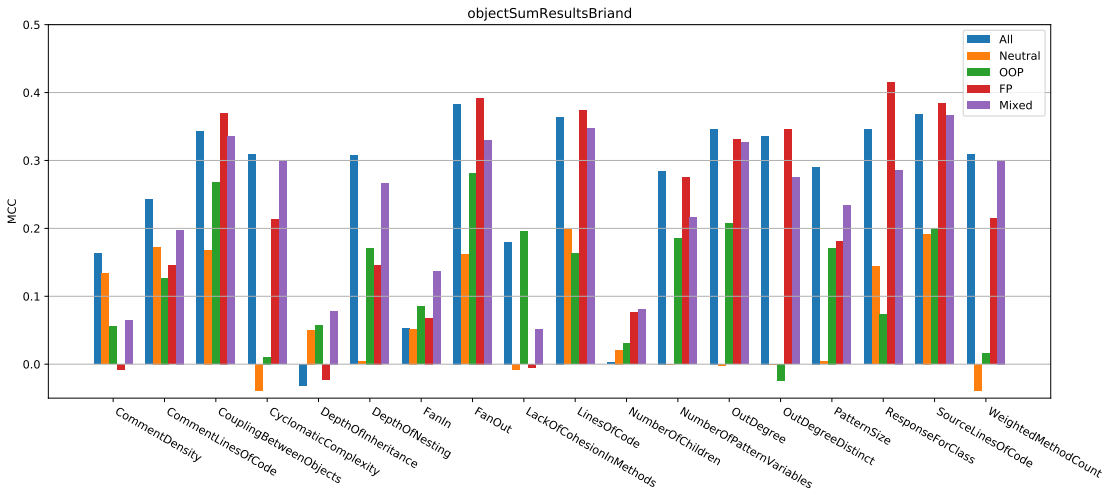


Figure 6.1: Baseline model metrics average MCC per paradigm using Briand's methodology.

Most metrics perform well on FP-style objects, including OOP metrics. Coupling-based OOP metrics like coupling between objects, fan-out, and response for class are the most indicative for finding faults. This could be because Scala uses OOP constructs for encapsulation even when writing FP-style code. The general and FP metrics also perform well on FP-style objects, with the exception of comment density. Complexity-based OOP metrics perform fairly well, whereas inheritance- or cohesion-based OOP metrics do not.

The mixed category has the most consistent performance. This could be because the baseline model contains separate metrics for both paradigms, but also because the mixed category had more faulty data available compared to the other categories. The most indicative metrics are the coupling between objects, fan-out, (source) lines of code, and outdegree. The weighted method count (and therefore cyclomatic complexity) and the depth of nesting perform significantly better for mixed objects compared to the other categories.

When comparing the performance without splitting by paradigm with the performance for each category, the response for class metric shows a significant improvement when only considering FP-style code. Most other metrics show either no or only small performance improvements. Some metrics that score poorly overall did show improvements when only considering a single paradigm, however, even with the improvements these metrics still performed poorly. Interestingly, when considering metrics with an MCC above 0.2 the only category which shows performance improvements over not splitting by paradigm is the FP category.

For Landkroon's methodology, the sum summarising strategy also yields the best overall performance. For Landkroon's methodology, only Akka, Gitbucket and Scio had enough data in the OOP category. This means the results used for the comparison are based on these three projects. The percentage of objects that contain faults per paradigm is shown in Table 6.4. Compared

to Briand's methodology, neutral objects are slightly less common and less likely to be faulty, OOP objects are slightly more common, FP objects are both more common and more likely to be faulty, and mixed objects are also more likely to be faulty. In addition, each category has a higher SLOC, but the differences between categories are similar.

Paradigm	Relative size	Faulty% mean	Faulty% std.	SLOC mean	SLOC std.
Neutral	58.01%	4.61%	1.70%	9.21	2.65
OOP	7.35%	13.52%	7.54%	27.58	14.43
FP	10.06%	21.29%	10.83%	50.49	31.44
Mix	24.58%	34.09%	13.42%	66.77	27.55

Table 6.4: Average percentage of faulty objects and average SLOC per paradigm using Landkroon's methodology.

The average MCC per metric for Landkroon's methodology can be found in Figure 6.2. The results for the neutral category are similar to using Briand's methodology, although the overall scores have slightly improved. For all other categories, the performance of the general metrics, especially the comment metrics, has significantly increased. The metric most indicative for faults in every category is the (source) lines of code. For OOP metrics, the performance of the cyclomatic complexity, fan-out, and weighted method count metrics has also improved significantly. For the FP category, the performance of the depth of inheritance and lack of cohesion in methods has significantly increased, whereas these metrics performed poorly when using Briand's methodology. For the mixed category, the performance of the cyclomatic complexity, fan-out, lack of cohesion in methods, outdegree (distinct), response for class, and weighted method metrics count have all significantly improved. When using Landkroon's methodology none of the metrics with an MCC above 0.2 show significant performance improvements when splitting by paradigm.

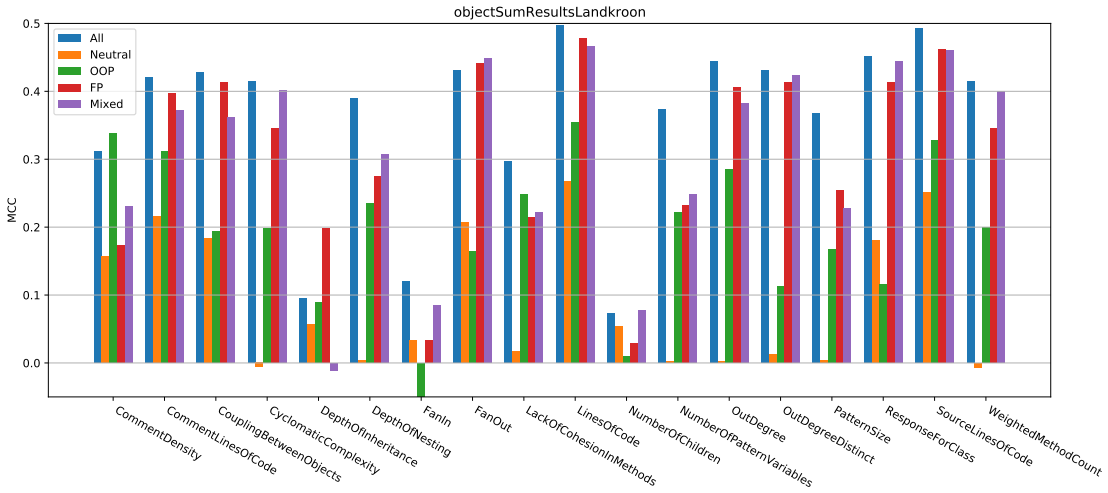


Figure 6.2: Baseline model metrics average MCC per paradigm using Landkroon's methodology.

6.4 Conclusion

The results presented in Section 6.3 brings us to the following answers to the research questions:

RQ3 *To what extent is the fault-proneness prediction ability of existing OOP and FP metrics affected by the mix of OOP and FP within a class or method?*

The results are slightly less reliable because most projects did not have enough OOP data available. The general metrics perform well on every category, especially when using Landkroon's methodology. Surprisingly enough, most OOP metrics perform better on the FP and mixed categories than in the OOP category. Especially the response for class and, when using Briand's methodology, the weighted method count perform significantly worse for the OOP category compared to the FP and mixed categories. When using Briand's methodology the lack of cohesion of methods metric only performs well in the OOP category. However, this effect does not occur when using Landkroon's methodology. The FP metrics perform well in the FP and mixed categories. Most FP metrics also perform fairly well in the OOP category, with the exception of outdegree distinct. Overall, there are not many metrics which only perform well on a certain category or show a significant improvement compared to their performance without splitting by paradigm. An interesting effect that was observed when splitting the code by paradigm was that mixed code had a significantly higher percentage of faults in the analysed projects.

Chapter 7

Metrics tailored to OOP and FP

This Chapter introduces and analyses candidate metrics for Scala tailored to the combination of OOP and FP. Section 7.1 presents the candidate metrics. The results are presented in Section 7.2. Finally, Section 7.3 concludes this chapter by answering **RQ4**.

7.1 Candidate metrics

This section defines the candidate metrics that will be analysed. The candidate metrics were defined based on the following sources:

1. The metrics for the functional side of C# by Zuilhof [53].
2. The OOP or FP constructs that are significantly more fault-prone.
3. The existing OOP and FP metrics that are significantly affected by the mix of OOP and FP.

The first source has resulted in several metric definitions aimed at the use of lambdas. These metrics are defined in Section 7.1.1. The second source has resulted in several metrics definitions measuring specific Scala constructs. These metrics are defined in Section 7.1.2. The last source has not resulted in any metrics, because none of the analysed existing OOP and FP metrics showed an MCC increase high enough to improve the baseline model when only considering a single paradigm.

7.1.1 Zuilhof's metrics

Zuilhof has defined MP metrics for the functional side of C# [53]. Most of these metrics can also be applied to Scala, with the exception of the unterminated collection queries metric since Scala collection queries are always terminated. This section describes the metrics defined by Zuilhof which are applicable to Scala.

Number of Lambda Functions Used in a Class

Counts the number of lambda functions used in a class. Lambda functions could introduce constructs which are harder to understand, therefore make the code more fault-prone.

Source Lines of Lambda

Counts the number of lines containing lambda functions. More lines spent on lambda functions could indicate more complicated lambdas, making them more fault-prone.

Lambda Score

The lambda score is the ratio of lines containing lambda functions to the number of source code lines. This metric indicates the relative amount of code spent on lambda functions, which in turn tells us something about the style the class is written in.

Number of Lambda Functions Using Outer Variables

Counts the number of lambda functions using variables defined outside the method scope. The usage of these variables could cause the behaviour of the lambda to change, which would increase the fault-proneness.

Number of Lambda Functions Using Local Variables

Counts the number of lambda functions using variables defined inside the method scope (but outside the lambda itself). The usage of these variables could cause the behaviour of the lambda to change, which could increase the fault-proneness.

Number of Lambda Functions With Side-Effects

Counts the number of lambda functions using side-effects. The usage of side-effects means the lambdas are not pure. This could cause issues if higher-order methods assume the functions are pure and thus increase the fault-proneness. In this case, side-effects can also include print statements, which are less likely to cause problems since they are generally not related to the behaviour of the code.

Number of Lambda Functions With Assignments

Counts the number of lambda functions using assignments. The usage of assignments means the lambda directly modifies variables. This could cause issues if higher-order methods assume the functions are pure and thus increase the fault-proneness. Assigning a variable is also an indication the lambda relies on the execution order. The execution order of lambda functions is not necessarily guaranteed, for example, parallel collections assume lambdas are pure and can be applied to the collection elements in any order.

7.1.2 Construct metrics

Based on the construct analysis (Section 2.2.7 and the construct measurements (Section 5.1 candidate metrics can be defined. This sections describes the candidate metrics that have been defined.

Number of implicits

Implicits in Scala are confusing to (new) programmers and hard to debug due to their implicit nature. Implicits are used by the compiler based on whether they are in scope. This means a single import could (unexpectedly) change the behaviour of the program. The measurements of implicits are split-up into three categories: definitions, conversions and parameters. Definitions are implicits defined within the code, conversions are when the type of a value is automatically converted to another type by the compiler, and parameters are when using a method with an implicit parameter that has to be filled in.

Number of Unit variables

Everything in Scala is an expression. Therefore, even statements, methods and functions that do not have a return value, should return a valid object. For this purpose, Scala uses the *Unit* type, which always is an empty object. Variables that have the *Unit* type were likely supposed to have another type, but got assigned an expression which returns the *Unit* type instead. This metric counts the number of unit variables.

Number of Function variables

Functions in Scala are often used anonymously as lambda functions. Since methods can automatically be converted to functions, they are often used instead of a function when a named function is desired. However, it is possible to assign a function to a variable and use the variable instead of a method. This metric counts the number of function variables.

Number of nulls

The official Scala book recommends to avoid *null* and use the *Option* type instead to avoid null pointers [1]. However, to stay compatible with Java, *null* is allowed. Since Scala code often does not expect *null* values, the usage of *null* could cause issues, even more so than it does in Java. This metric counts the number of times *null* is used in the code.

Number of returns

In Scala, returns are unnecessary since methods are defined using expressions, which already have a return value by definition. Additionally, when using return within a nested function it is implemented using a special exception that is caught. This means that non-local returns are possible, although they often cause unexpected results. Furthermore, the exception can be caught when catching runtime exceptions, which could also lead to unexpected behaviour.

Pattern variable name shadowing

In pattern-matching it is easy to shadow a variable name and create a new variable, when the intention was to match based on the value of the old variable. An example of this is shown in Listing 7.1. The shadowing of a variable could be accidental and, therefore, cause code to be faulty. This metric counts the number of pattern variables that shadow the names of outer variables.

```
1  def matchString(string: String) = {
2    val First = "first"
3    val second = "second"
4    string match {
5      // Matches if string has the value "first", only works for capitalized variables
6      case First => println("I'm first")
7      // Matches any string and assigns the value to a variable called second, shadowing the
       ↪ outer variable
8      case second => println("I'm second")
9    }
10 }
```

Listing 7.1: Scala match variable shadowing example.

7.2 Results

The fault-proneness prediction performance of the candidate metrics has been measured using univariate and multivariate regression. The univariate regression shows the performance of the metric on its own. The multivariate regression is done by adding the metric to the baseline model and measuring the performance of all metric combined. The multivariate regression shows whether the metric improves the baseline model.

The univariate regression results can be found in Table 7.1. Based on the MCC when using Briand's methodology, the number of lambda functions, lambda score, and implicit parameters metrics are the best predictors on their own. When using Landkroon's methodology, the number of lambda functions and source lines of lambda metrics are the best predictors on their own. However, since the metrics are used to improve the prediction performance of the baseline model, the precision is also very important. The precision indicates that the measure is a reliable way to find (additional) faults and could potentially improve the baseline model. Based on the precision the overriding pattern variables and lambda functions using assignments metrics would be the best candidates when using Briand's methodology. Nevertheless, the low recall of both metrics does indicate that the improvements are likely minor since it does not find many (additional) cases. When using Landkroon's methodology, the source lines of lambda, lambda functions with side effects, and overriding pattern variables metrics are the most promising candidates.

Name	Precision mean	Precision std.	Recall mean	Recall std.	MCC mean	MCC std.
Briand's methodology						
NumberOfLambdaFunctions	34.18	8.20	44.55	4.91	0.305	0.052
SourceLinesOfLambda	36.66	8.35	41.49	5.25	0.311	0.052
LambdaScore	26.54	8.59	43.52	8.72	0.231	0.082
LambdaFunctionsUsingOuterVariables	35.72	42.69	16.73	32.37	0.072	0.099
LambdaFunctionsUsingLocalVariables	36.76	30.21	6.00	3.73	0.089	0.100
LambdaFunctionsWithSideEffects	33.43	17.88	16.93	10.51	0.167	0.122
LambdaFunctionsWithAssignment	39.06	32.69	8.06	4.63	0.119	0.127
ImplicitConversions	25.44	10.57	27.10	10.81	0.170	0.098
ImplicitDefinitions	32.76	10.60	32.49	10.99	0.237	0.097
ImplicitParameters	36.00	9.04	35.48	6.97	0.282	0.073
UnitVariables	22.79	17.98	17.30	32.98	0.042	0.044
FunctionVariables	34.84	10.85	27.96	8.51	0.235	0.092
UsageOfNull	36.89	12.77	15.32	5.39	0.172	0.061
NumberOfReturns	15.20	26.89	28.99	45.34	0.020	0.054
OverridingPatternVariables	44.79	15.35	14.24	6.27	0.194	0.055
Landkroon's methodology						
NumberOfLambdaFunctions	45.56	13.76	46.07	10.62	0.368	0.121
SourceLinesOfLambda	50.68	11.20	43.47	10.48	0.392	0.105
LambdaScore	33.08	9.49	46.90	12.49	0.273	0.108
LambdaFunctionsUsingOuterVariables	34.28	34.59	16.12	33.04	0.070	0.080
LambdaFunctionsUsingLocalVariables	47.80	31.41	4.35	3.71	0.096	0.091
LambdaFunctionsWithSideEffects	53.48	21.58	19.97	12.65	0.232	0.144
LambdaFunctionsWithAssignment	48.63	30.76	19.36	28.96	0.146	0.165
ImplicitConversions	35.94	13.02	33.80	13.54	0.240	0.131
ImplicitDefinitions	42.11	18.60	34.94	14.05	0.286	0.151
ImplicitParameters	42.91	13.30	34.72	10.49	0.293	0.096
UnitVariables	23.68	16.88	18.18	32.67	0.041	0.066
FunctionVariables	40.15	13.95	29.49	6.52	0.246	0.077
UsageOfNull	41.17	16.06	17.73	10.86	0.190	0.094
NumberOfReturns	29.17	40.75	41.10	48.99	0.046	0.088
OverridingPatternVariables	49.68	16.37	17.11	8.66	0.228	0.110

Table 7.1: Univariate regression fault-proneness prediction performance.

The results of the multivariate regression can be found in Table 7.2. The table shows the performance of the baseline model followed by the differences in score when adding one of the candidate metrics. As can be seen, the candidate metrics only have a small impact on the fault-proneness prediction performance. For Briand's methodology, the overriding pattern variables and lambda functions using outer variables metrics show the largest improvements with an MCC increase around 0.008. However, this is not significant enough to be meaningful. For Landkroon's methodology, the differences are even smaller. Overall, the candidate metrics do not yield large improvements. However, the metrics with the highest precision in the univariate regression often had a low recall. This is mainly because the constructs they measured did not occur often and, therefore, those metrics can only give us information about small parts of the code base. These metrics are still interesting to investigate, because they could indicate constructs that, although they are not used often, commonly cause issues when used. These constructs should then be investigated in detail to find the root causes.

Name	Precision mean	Precision std.	Recall mean	Recall std.	MCC mean	MCC std.
Briand's methodology						
Baseline	31.34	8.00	62.23	7.64	0.346	0.066
NumberOfLambdaFunctions	0.21	−0.17	−0.71	−0.40	0.000	−0.003
SourceLinesOfLambda	0.43	0.21	−1.10	0.05	0.001	0.002
LambdaScore	−0.23	−0.36	−0.61	0.13	−0.004	−0.004
LambdaFunctionsUsingOuterVariables	0.60	1.05	0.23	0.09	0.007	0.010
LambdaFunctionsUsingLocalVariables	0.04	0.25	0.44	0.35	0.002	0.006
LambdaFunctionsWithSideEffects	0.06	0.54	0.34	1.11	0.003	0.013
LambdaFunctionsWithAssignment	0.39	0.71	0.34	0.17	0.005	0.009
ImplicitConversions	0.41	−0.50	0.23	0.14	0.005	−0.004
ImplicitDefinitions	0.64	0.46	−0.69	−0.05	0.004	0.014
ImplicitParameters	0.37	−0.33	−1.07	0.02	0.000	−0.002
UnitVariables	0.27	0.16	0.08	−0.35	0.003	−0.003
FunctionVariables	0.04	0.93	−0.75	−0.67	−0.002	0.011
UsageOfNull	0.51	0.20	0.14	−0.44	0.006	0.000
NumberOfReturns	0.30	0.60	0.44	0.30	0.005	0.007
OverridingPatternVariables	0.52	−0.04	0.70	−0.01	0.008	−0.001
Landkroon's methodology						
Baseline	42.34	15.63	62.62	9.55	0.415	0.136
NumberOfLambdaFunctions	−0.15	−0.10	0.37	−0.04	0.000	0.000
SourceLinesOfLambda	−0.01	0.38	0.69	−0.13	0.002	0.004
LambdaScore	−0.04	−0.01	0.14	−0.13	0.000	−0.001
LambdaFunctionsUsingOuterVariables	−0.28	−0.26	0.27	0.46	−0.001	0.000
LambdaFunctionsUsingLocalVariables	0.07	−0.02	0.49	0.37	0.002	0.002
LambdaFunctionsWithSideEffects	0.00	0.12	0.37	−0.08	0.001	0.001
LambdaFunctionsWithAssignment	−0.08	0.11	0.05	0.33	−0.001	0.003
ImplicitConversions	0.25	−0.25	0.09	−0.14	0.002	−0.002
ImplicitDefinitions	0.03	0.43	−0.15	0.41	−0.001	0.004
ImplicitParameters	0.08	0.38	0.53	0.39	0.002	0.006
UnitVariables	0.10	0.02	0.14	−0.06	0.001	0.000
UsageOfNull	−0.02	0.02	0.25	0.08	0.000	0.000
FunctionVariables	0.21	0.03	0.14	−0.25	0.002	−0.001
NumberOfReturns	0.08	0.02	0.06	0.27	0.001	0.002
OverridingPatternVariables	−0.17	0.05	0.37	0.20	0.000	0.002

Table 7.2: Multivariate regression fault-proneness prediction performance.

To investigate these metrics we have analysed two statistics:

S1. The percentage of objects containing faults for all the objects for which the metric has measured results

S2. The percentage of objects for which the metric has measured results for all the objects containing faults

Name	Count	S1 mean	S1 std.	S2 mean	S2 std.
Briand's methodology					
All objects	7	10.66%	2.97%	100.00%	0.00%
NumberOfLambdaFunctions	7	26.10%	7.58%	55.38%	13.00%
SourceLinesOfLambda	7	26.10%	7.58%	55.38%	13.00%
LambdaScore	7	26.10%	7.58%	55.38%	13.00%
LambdaFunctionsUsingOuterVariables	5	50.58%	42.68%	3.97%	3.73%
LambdaFunctionsUsingLocalVariables	6	53.68%	14.54%	4.29%	3.48%
LambdaFunctionsWithSideEffects	7	39.55%	14.39%	15.68%	11.98%
LambdaFunctionsWithAssignment	6	52.21%	21.35%	6.69%	5.48%
ImplicitConversions	7	24.51%	10.31%	28.79%	10.60%
ImplicitDefinitions	7	25.36%	6.95%	43.91%	18.36%
ImplicitParameters	7	32.68%	8.96%	40.58%	8.64%
UnitVariables	6	26.18%	17.58%	3.55%	5.33%
FunctionVariables	7	30.07%	11.09%	33.04%	15.45%
UsageOfNull	7	36.89%	12.77%	15.32%	5.39%
NumberOfReturns	5	45.00%	44.72%	0.93%	1.46%
OverridingPatternVariables	7	44.79%	15.35%	14.24%	6.27%
Landkroon's methodology					
All objects	7	14.05%	4.47%	100.00%	0.00%
NumberOfLambdaFunctions	7	32.27%	7.53%	58.80%	15.43%
SourceLinesOfLambda	7	32.27%	7.53%	58.80%	15.43%
LambdaScore	7	32.27%	7.53%	58.80%	15.43%
LambdaFunctionsUsingOuterVariables	5	44.10%	36.41%	4.54%	5.98%
LambdaFunctionsUsingLocalVariables	6	61.55%	17.04%	3.49%	3.03%
LambdaFunctionsWithSideEffects	7	53.15%	21.35%	19.26%	14.42%
LambdaFunctionsWithAssignment	6	60.17%	18.63%	9.34%	11.78%
ImplicitConversions	7	35.68%	12.99%	35.49%	11.77%
ImplicitDefinitions	7	34.48%	12.32%	42.77%	15.29%
ImplicitParameters	7	39.60%	12.66%	40.66%	10.81%
UnitVariables	6	27.38%	16.36%	4.69%	9.39%
FunctionVariables	7	33.63%	14.24%	31.49%	8.69%
UsageOfNull	7	41.17%	16.06%	17.73%	10.86%
NumberOfReturns	5	35.00%	48.73%	1.56%	2.73%
OverridingPatternVariables	7	49.68%	16.37%	17.11%	8.66%

Table 7.3: Fault statistics per metric.

These statistics have been collected for each project and the average results can be found in Table 7.3. For some projects, a few metrics had no results. These projects have been excluded for those metrics. The amount of projects over which the average is calculated is shown in the Count column. For Briand's methodology, we can see that the lambda function using outer/local variables and lambda function using assignment metrics only have results for less than 7% of the faulty objects (S2). This means they are only related to a minority of the faults. However, over 50% of the objects for which those metrics have measured results contain faults (S1). This means that when an object within the selected projects uses, for example, a lambda function with an assignment, there is over 50% chance this object contains a fault. This is significantly more

than when an object uses lambdas (around 26%) or the overall chance of an object containing faults (around 10%). When using Landkroon's methodology, over 60% of the objects for which the lambda functions using local variables and lambda functions with assignment metrics have measured results contain faults. These differences indicate that, although these constructs do occur less often, they might be more fault-prone.

To verify to what extend the baseline model is affected by one of the candidate metrics when only considering objects for which the metric has data, we have measured the multivariate regression results on those objects with and without the candidate metric. The average MCC for Briand's methodology is visualized in Figure 7.1 and the average MCC for Landkroon's methodology is visualized in Figure 7.2. The full results can be found in Appendix D. Results have only been collected for projects that had at least 10 objects containing faults left after filtering the objects based on metric data. The number of projects over which the results have been averaged is displayed above each set of bars. Overall, the differences with and without the candidate metric are very small. This indicates most faulty code is already detected by the baseline model without the candidate metric. In some cases, there are significant improvements to the MCC when adding the candidate metric. However, these significant improvements occur when only a single project had enough data for analysis. This makes these results unreliable. Another interesting result is that no projects had enough data for the number of returns metric. To verify this we have searched for uses of the return keyword in the analysed projects and it was seldom used. This is somewhat surprising since the use of return is very common in a lot of other programming languages.

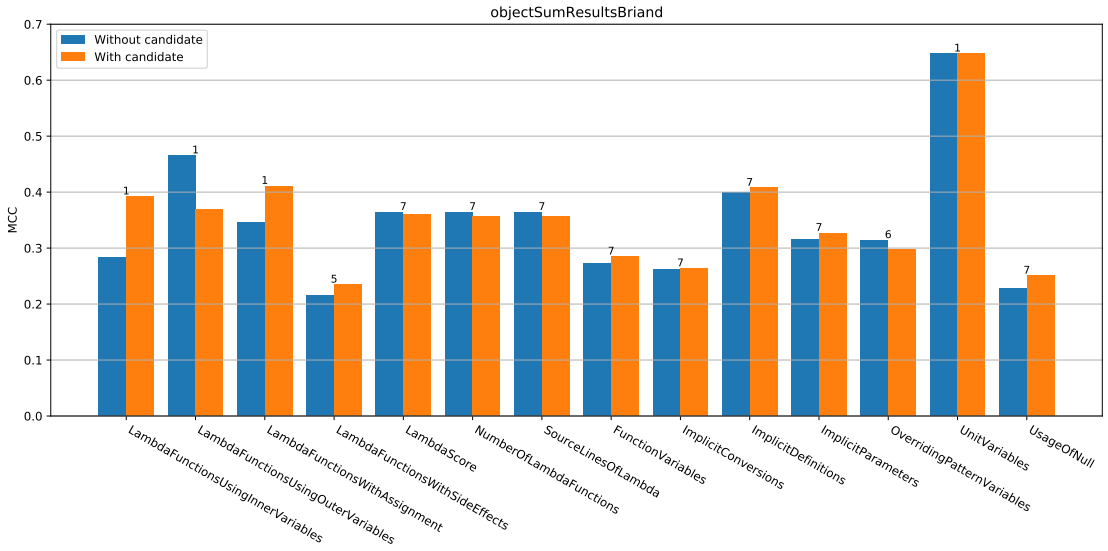


Figure 7.1: Multivariate regression fault-proneness prediction performance for objects with metric results (Briand).

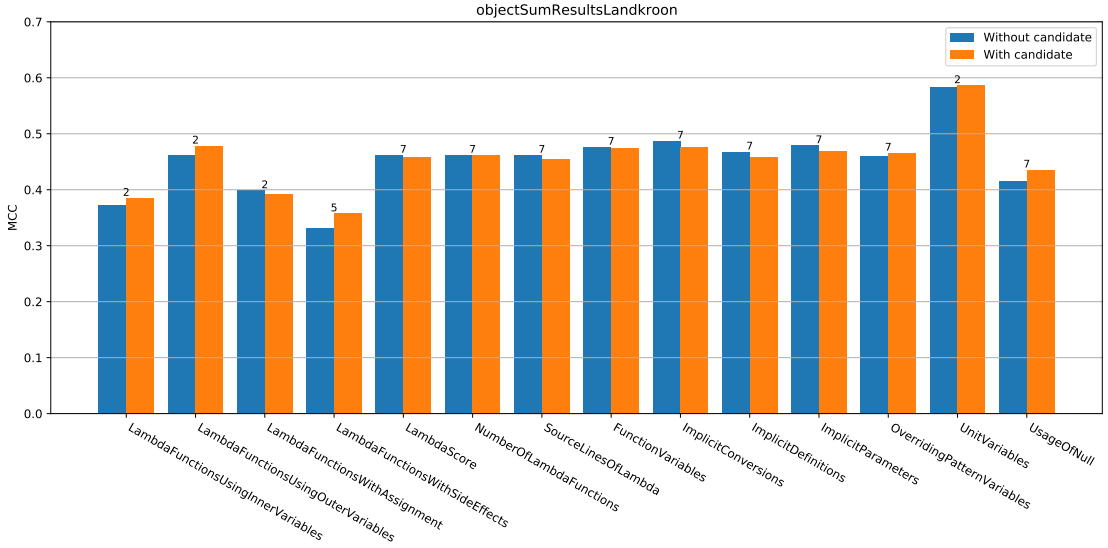


Figure 7.2: Multivariate regression fault-proneness prediction performance for objects with metric results (Landkroon).

7.3 Conclusion

The results presented in Section 6.3 brings us to the following answers to the research questions:

RQ4 *To what extent can the fault-proneness prediction performance of the baseline model be improved by adding metrics tailored to the combination of OOP and FP?*

None of the candidate metrics show a significant enough improvement to the baseline model to establish with sufficient certainty that they are an improvement. Most of the candidate metrics only measure specific constructs which do not occur often enough in the large code-bases to make a difference. The results indicate that even when the candidate metrics have results, the baseline model already manages to detect the faulty objects and adding the candidate metric does not make a significant difference. However, the candidate metrics have still resulted in interesting pointers to determine why the fault occurred instead of just determining that it occurred. Generally, the majority of objects contained faults when lambda functions using variables, side-effects and/or assignment were detected. This indicates that although these metrics might not be a significant improvement over the baseline model, they could potentially pinpoint why the code contained faults.

Chapter 8

Related work

Landkroon researched the applicability of existing OOP and FP metrics on Scala [30]. He has shown that we can use existing OOP and FP metrics as an indication of fault-proneness. However, he indicates that the existing OOP and FP metrics do not cover everything and that the results can be improved by adding metrics tailored to the MP paradigm. He also presents his own validation methodology, based on Briand's validation methodology [6]. Within Landkroon's research, his validation methodology has an overall higher performance (up to more than a two-fold increase in completeness), especially for projects with longer life-cycles, compared to Briand's methodology.

C# started as an OOP language and has become an MP language by adding FP constructs. Zuilhof defined and validated MP metrics for the functional side of C# [53]. These metrics focus on lambda functions within C# and improve the fault-proneness prediction of most analyzed projects.

Sipos, Pataki, and Porkoláb defined paradigm agnostic software complexity metrics for MP languages [44]. These metrics are calculated by converting the program to a control-flowgraph and applying the calculations to this graph. These have been validated on Aspect-Oriented Programming and OOP projects.

Jordan and Collier researched multi-paradigm metrics for the combination of Agent-Oriented Programming in combination with OOP [27]. They propose generalized paradigm-independent versions of the Coupling Between Object Classes and Lack of Cohesion Of Methods metrics. The generalized variants are called Coupling Between Elements and Lack of Cohesion of Elements. These generalized metrics reason about abstractions and elements instead of classes, methods and fields.

Next to the research that has been done in the field of MP metrics, the industry has also developed static code analysis tools. Some of these tools include support for MP languages such as Scala. One of the most well-known tools in this area is SonarQube [45]. SonarQube focuses on detecting patterns that are likely to be bugs, vulnerabilities or code smells. These patterns are detected using rules. Each language has its own set of rules. SonarQube includes rules for Scala [46]. Most of the rules for Scala are simple patterns, e.g. no empty methods or no duplicate implementations. There is also a rule that is based on the cognitive complexity metric. Cognitive Complexity is SonarQube's alternative for Cyclomatic Complexity [7]. It is aimed at

accurately reflecting the relative difficulty of understanding, and therefore of maintaining methods, classes, and applications. Code is considered more complex when it has breaks in the linear flow (like loops or recursion) or has nested structures. Apart from the Cognitive Complexity metric, SonarQube does not implement any other metrics for Scala (only patterns).

The Software Improvement Group (SIG) has a tool called BetterCodeHub [16]. This tool implements the measurements of the maintainability model by SIG[50]. It has support for measuring Scala programs. One of the goals of the maintainability model is to be language-independent. As a result of this, the measurements of BetterCodeHub are mostly language-independent. There is one exception however, the unit complexity. The complexity is measured using McCabe's complexity measure [31]. Van den Berg has carried out an experiment in which experts ranked the readability of programs written in the imperative language Pascal and the FP language Miranda [49]. Metrics, including McCabe's complexity measure, were then applied to the programs. The correlation between the ordering of the metric results and the experts' opinions was measured. This case study showed that while there was a reasonably high correlation between the metrics and the experts' view for imperative programs (Pascal), the correlation was significantly lower for functional programs (Miranda).

Chapter 9

Conclusion

This chapter concludes this Masters' thesis. Section 9.1 summarises the conclusions to the research question. Section 9.2 evaluates and discusses the design choices. Finally, Section 9.3 presents several ideas and directions for future investigation.

9.1 Findings

The answers to the research questions are as follows:

RQ1 *Which, if any, OOP or FP constructs in Scala are significantly more fault-prone than others?*

Within Scala, OOP constructs that are not imperative can be considered neutral and the program logic uses imperative and/or functional constructs. Within these imperative and functional constructs, there is not a single set of constructs that is significantly more fault-prone than others. However, there is still a difference between constructs. The most fault-prone constructs are (outer) variable usage, variable definitions, inner variable assignments, functions with side-effects, nested methods and pattern matching.

RQ2 *How well does the paradigm score perform as a predictor for fault-proneness?*

The paradigm score on its own does not perform well as a predictor for fault-proneness. The precision of the different paradigm scores was around 10% and the recall around 60%.

RQ3 *To what extent is the fault-proneness prediction ability of existing OOP and FP metrics affected by the mix of OOP and FP within a class or method?*

The results are slightly less reliable because most projects did not have enough OOP data available. The general metrics perform well on every category, especially when using Landkroon's methodology. Surprisingly enough, most OOP metrics perform better on the FP and mixed categories than in the OOP category. Especially the response for class and, when using Briand's methodology, the weighted method count perform significantly worse for the OOP category compared to the FP and mixed categories. When using Briand's methodology the lack of cohesion of methods metric only performs well in the OOP category. However, this effect does not occur when

using Landkroon's methodology. The FP metrics perform well in the FP and mixed categories. Most FP metrics also perform fairly well in the OOP category, with the exception of outdegree distinct. Overall, there are not many metrics which only perform well on a certain category or show a significant improvement compared to their performance without splitting by paradigm. An interesting effect that was observed when splitting the code by paradigm was that mixed code had a significantly higher percentage of faults in the analysed projects.

RQ4 *To what extent can the fault-proneness prediction performance of the baseline model be improved by adding metrics tailored to the combination of OOP and FP?*

None of the candidate metrics show a significant enough improvement to the baseline model to establish with sufficient certainty that they are an improvement. Most of the candidate metrics only measure specific constructs which do not occur often enough in the large code-bases to make a difference. The results indicate that even when the candidate metrics have results, the baseline model already manages to detect the faulty objects and adding the candidate metric does not make a significant difference. However, the candidate metrics have still resulted in interesting pointers to determine why the fault occurred instead of just determining that it occurred. Generally, the majority of objects contained faults when lambda functions using variables, side-effects and/or assignments were detected. This indicates that although these metrics might not be an improvement over the baseline model, they could potentially pinpoint why the code contained faults.

Overall, the baseline model performed decently and the performance was not significantly improved by adding one of the candidate metrics tailored to the combination of OOP and FP. Even when only considering the cases in which the candidate metric was relevant, there was no significant improvement over the baseline model. However, the fault-proneness prediction of the baseline model can only tell us when code is likely to contain faults. It does not provide us with information on why the code contains faults. If we know why the code contains faults, we can detect patterns that cause these faults and use those detections to prevent the faults from reaching production code. The candidate metrics can provide insights into why code contains faults. One of the areas that warrants further investigation to find such patterns, is when using lambda functions with variables, side-effects and/or assignments. The majority of objects using these lambda functions contained faults, therefore this is a promising area to investigate.

9.2 Discussion

Initially, we assumed that the paradigm score would be a measure comparing the number of OOP constructs to the number of FP constructs used in a piece of code. However, while defining the paradigm score we realized that OOP within Scala could be considered as a separate paradigm from imperative programming. The type system and encapsulation within Scala is based on OOP and the related constructs are not only used when writing OOP-style code, but also when writing FP-style code. Therefore, these constructs are used independent of style and can be considered neutral. For the paradigm score to accurately reflect which style the code is written in, we had to exclude these neutral OOP constructs and instead only measure the imperative constructs.

The metrics were measured using the AST of the Scala compiler. This AST has already been preprocessed by the compiler. During this preprocessing, for-expressions are translated to *filter*, *map* and *foreach* calls. This affects the results of some of the metrics, including the paradigm

score, since these calls are higher-order methods. The advantage of using the Scala compiler AST is that it is possible to measure inferred types and the application of implicit conversions/parameters. This information is not always complete, because not all external libraries are available. However, most typing information is available and having access to this information helped developing metrics.

The fault-analysis assumes every issue is correctly labelled and the issue is mentioned in the commit or pull-request that fixed it. This is, of course, not always the case. First of all, issues can be mislabelled. Issues could be incorrectly labelled as a bug, or, more commonly, issues which should have been labelled as bug remain unlabeled. Furthermore, commits or pull-request do not always mention the issues they fix, especially if this is only discovered afterwards that the issue is fixed. This could also be positive, because if a commit explicitly mentions an issue it is more likely to be a fix commit, instead of a different type of commit that happens to fix the issue as a side-effect. However, commits that change more code than needed to fix the bug are still likely to be included in the analysis. This results in some code being marked incorrectly as containing faults, which decreases the reliability of the analysis. A small set of fix commits of each analysed projects has been investigated to make sure the projects are reliable. There are likely still inconsistencies, but the bulk analysis across several projects should minimize the impact.

Finally, most of the selected projects did not contain enough OOP-style code for analysis. Because of this, the metric performance by paradigm comparison could only use 2 out of the 7 projects for Briand's methodology and 3 out of 7 for Landkroon's methodology. This makes the results of the comparison less reliable and representative.

9.3 Future work

There are always more metrics to validate and different variations on existing metrics. Some of these could improve the baseline model or provide new insights on why code contains faults. This research provides an implementation for defining and analysing metrics, including a set of projects to analyse and performance data of the metrics analysed in this research. Upon this basis, new metrics can be defined and explored.

Another approach is to investigate how the analysed metrics perform on different projects. Currently, seven projects have been analysed, most of which did not contain enough OOP code for the analysis split by paradigm score. It would be interesting to see how the addition of other projects affects the results. Generally, only larger projects have enough issues labelled as faults to provide reliable results. The Apache organisation manages some of the largest Scala projects. These projects use Jira as issue tracker and meticulously label their faults. Adding support for the Jira issue tracker to the framework would allow the analysis of these projects.

Some of the analysed metrics could be further investigated to find out why the code contains faults. Constructs related to these metrics could be investigated in detail to find common mistakes that cause faults. If we know which common mistakes cause these faults, we can implement patterns that detect these faults and use those to prevent the faults from reaching production code. The resulting patterns could be added to code analysis tools like SonarQube [45]. Based on the fault-statistics of the investigated metrics, the lambda functions using variables, side-effects and assignments are the most promising metrics.

Finally, it would be interesting to see how the metrics perform on other multi-paradigm languages. The design of both the Scala language itself and the Scala standard library affects how the language is used. This is different for other multi-paradigm languages and could affect the results. The popularity of relatively new multi-paradigm languages like Kotlin is rapidly increasing [8]. These languages would be interesting to investigate.

References

[1] Alvin Alexander. *Scala Book*. 2020. URL: https://docs.scala-lang.org/overviews/scala-book/introduction.html (visited on 02/19/2020).

[2] Oliver Arafati and Dirk Riehle. "The comment density of open source software code". In: *2009 31st International Conference on Software Engineering-Companion Volume*. IEEE. 2009, pp. 195–198.

[3] Tibor Bakota et al. "A probabilistic software quality model". In: *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE. 2011, pp. 243–252.

[4] Victor R Basili, Lionel C. Briand, and Walcélio L Melo. "A validation of object-oriented design metrics as quality indicators". In: *IEEE Transactions on software engineering* 22.10 (1996), pp. 751–761.

[5] Lionel C Briand, Walcelio L. Melo, and Jurgen Wust. "Assessing the applicability of fault-proneness models across object-oriented software projects". In: *IEEE transactions on Software Engineering* 28.7 (2002), pp. 706–720.

[6] Lionel Briand, Khaled El Emam, and Sandro Morasca. "Theoretical and empirical validation of software product measures". In: *International Software Engineering Research Network, Technical Report ISERN-95-03* (1995).

[7] G. Ann Campbell. *Cognitive Complexity: A new way of measuring understandability*. Tech. rep. SonarSource, 2018.

[8] Pierre Carbonnelle. *PYPL PopularitY of Programming Language*. 2020. URL: https://pypl.github.io (visited on 03/27/2020).

[9] Arti Chhikara, RS Chhillar, and Sujata Khatri. "Applying object oriented metrics to C# (C Sharp) programs". In: *International Journal of Computer Technology and Applications* 2.3 (2011), pp. 666–674.

[10] Davide Chicco and Giuseppe Jurman. "The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation". In: *BMC genomics* 21.1 (2020), p. 6.

[11] Shyam R Chidamber and Chris F Kemerer. "A metrics suite for object oriented design". In: *IEEE Transactions on software engineering* 20.6 (1994), pp. 476–493.

[12] Alonzo Church. "A set of postulates for the foundation of logic". In: *Annals of mathematics* (1932), pp. 346–366.

[13] Fabrizio Fioravanti and Paolo Nesi. "A study on fault-proneness detection of object-oriented systems". In: *Proceedings Fifth European Conference on Software Maintenance and Reengineering*. IEEE. 2001, pp. 121–130.

[14] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[15] Daniela Glasberg et al. *Validating object-oriented design metrics on a commercial java application*. Citeseer, 2000.

[16] Software Improvement Group. *Better Code Hub*. 2020. URL: https://bettercodehub.com/ (visited on 04/16/2020).

[17] Qiong Gu, Li Zhu, and Zhihua Cai. "Evaluation measures of the classification performance of imbalanced data sets". In: *International symposium on intelligence computation and applications*. Springer. 2009, pp. 461–471.

[18] Tibor Gyimothy, Rudolf Ferenc, and Istvan Siket. "Empirical validation of object-oriented metrics on open source software for fault prediction". In: *IEEE Transactions on Software engineering* 31.10 (2005), pp. 897–910.

[19] Péter Hegedűs. "A probabilistic quality model for c#-an industrial case study". In: *Acta Cybernetica* 21.1 (2013), pp. 135–147.

[20] Brian Henderson-Sellers, Larry L Constantine, and Ian M Graham. "Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design)". In: *Object oriented systems* 3.3 (1996), pp. 143–158.

[21] Martin Hitz and Behzad Montazeri. "Chidamber and Kemerer's metrics suite: a measurement theory perspective". In: *IEEE Transactions on software Engineering* 22.4 (1996), pp. 267–271.

[22] Martin Hitz and Behzad Montazeri. *Measuring coupling and cohesion in object-oriented systems*. Citeseer, 1995.

[23] Cay S Horstmann. *Scala for the Impatient*. Pearson Education, 2012.

[24] David W Hosmer Jr, Stanley Lemeshow, and Rodney X Sturdivant. *Applied logistic regression*. Vol. 398. John Wiley & Sons, 2013.

[25] *Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models*. Standard. Geneva, CH: International Organization for Standardization, Mar. 2011.

[26] Simon Peyton Jones, Geoffrey Washburn, and Stephanie Weirich. *Wobbly types: type inference for generalised algebraic data types*. Tech. rep. Technical Report MS-CIS-05-26, Univ. of Pennsylvania, 2004.

[27] Howell R Jordan and Rem Collier. "Evaluating agent-oriented programs: Towards multi-paradigm metrics". In: *International Workshop on Programming Multi-Agent Systems*. Springer. 2010, pp. 63–78.

[28] Alan Kay and Stefan Ram. *Dr. Alan Kay on the Meaning of "Object-Oriented Programming"*. 2003. URL: http://www.purl.org/stefan_ram/pub/doc_kay_oop_en (visited on 09/05/2020).

[29] Sven Konings. *Code Quality Analysis when Mixing Functional and Object-Oriented Programming in Scala*. Tech. rep. Info Support, 2020.

[30] Erik Landkroon. "Code Quality Evaluation for the Multi-Paradigm Programming Language Scala". MA thesis. Universiteit van Amsterdam, 2017.

[31] Thomas J McCabe. "A complexity measure". In: *IEEE Transactions on software Engineering* 4 (1976), pp. 308–320.

[32] Asma Mubarak, Steve Counsell, and Robert M Hierons. "An evolutionary study of fan-in and fan-out metrics in OSS". In: *2010 Fourth International Conference on Research Challenges in Information Science (RCIS)*. IEEE. 2010, pp. 473–482.

[33] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in scala*. Artima Inc, 2019.

[34] Martin Odersky et al. *An Overview of the Scala Programming Language*. Tech. rep. EPFL, 2006.

[35] Martin Odersky et al. "The Scala language specification version 2.13". In: *Programming Methods Laboratory, EPFL*. Citeseer. 2019.

[36] Chao-Ying Joanne Peng, Kuk Lida Lee, and Gary M Ingersoll. "An introduction to logistic regression analysis and reporting". In: *The journal of educational research* 96.1 (2002), pp. 3–14.

[37] Chris Ryder. "Software measurement for functional programming". PhD thesis. Computing Lab, University of Kent, 2004.

[38] Chris Ryder and Simon J Thompson. "Software metrics: measuring Haskell." In: *Trends in Functional Programming*. 2005, pp. 31–46.

[39] Lukas Rytz and Martin Odersky. "Named and default arguments for polymorphic object-oriented languages: a discussion on the design implemented in the Scala language". In: *Proceedings of the 2010 ACM Symposium on Applied Computing*. 2010, pp. 2090–2095.

[40] S. k. Samuel. *Scapegoat*. 2020. URL: `https://github.com/sksamuel/scapegoat` (visited on 08/03/2020).

[41] Scala Center. *Scaladex*. 2020. URL: `https://index.scala-lang.org/` (visited on 04/17/2020).

[42] Scala Center. *The Scala programming language*. 2020. URL: `https://www.scala-lang.org/` (visited on 02/19/2020).

[43] Scala Center. *Tour of Scala*. 2020. URL: `https://docs.scala-lang.org/tour/tour-of-scala.html` (visited on 02/20/2020).

[44] Adám Sipos, Norbert Pataki, and Zoltán Porkoláb. "On multiparadigm software complexity metrics". In: *MaCS 2006 6th Joint Conference on Mathematics and Computer Science*. 2006, pp. 85–100.

[45] SonarSource. *Code Quality and Security | SonarQube*. 2020. URL: `https://www.sonarqube.org/` (visited on 04/16/2020).

[46] SonarSource. *Scala Static Code Analysis Rules | Scala Code Analyzer*. 2020. URL: `https://rules.sonarsource.com/scala/` (visited on 04/16/2020).

[47] Mervyn Stone. "Cross-validatory choice and assessment of statistical predictions". In: *Journal of the Royal Statistical Society: Series B (Methodological)* 36.2 (1974), pp. 111–133.

[48] Mei-Huei Tang, Ming-Hung Kao, and Mei-Hwa Chen. "An empirical study on object-oriented metrics". In: *Proceedings sixth international software metrics symposium (Cat. No. PR00403)*. IEEE. 1999, pp. 242–249.

[49] Klaas Van den Berg. "Software measurement and functional programming". In: *University of Twente* (1995).

[50] Joost Visser. *SIG/TÜViT Evaluation Criteria Trusted Product Maintainability Version 11.0*. Tech. rep. Software Improvement Group, 2019.

[51] Joost Visser et al. *Building Maintainable Software: Ten Guidelines for Future-Proof Code*. O'Reilly Media, 2016.

[52] Dean Wampler and Alex Payne. *Programming Scala: Scalability = Functional Programming + Objects.* " O'Reilly Media, Inc.", 2014.

[53] Bart Zuilhof. "Code Quality Metrics for the Functional Side of the Object-Oriented Language C#". MA thesis. Universiteit van Amsterdam, 2019.

Appendix A

Fractional Paradigm Score plots

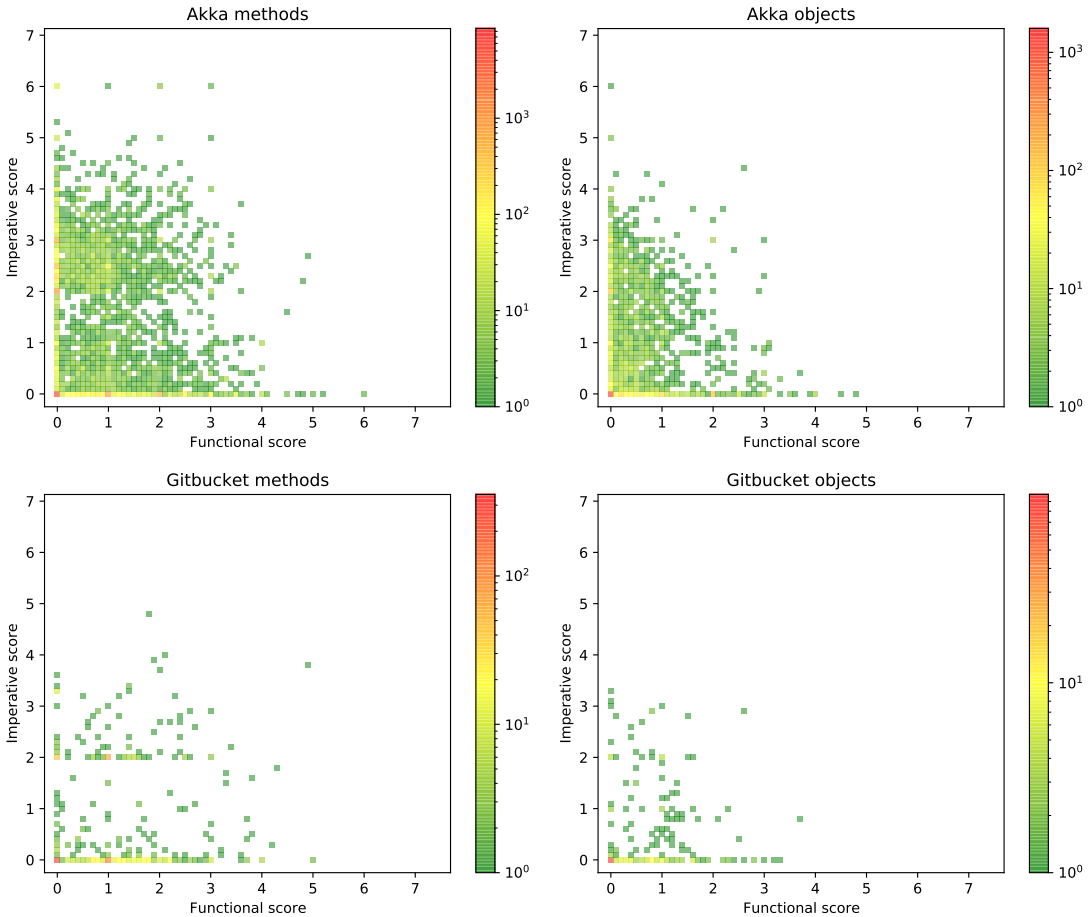


Figure A.1: Fractional paradigm score scatter plots. Color indicates number of occurrences.

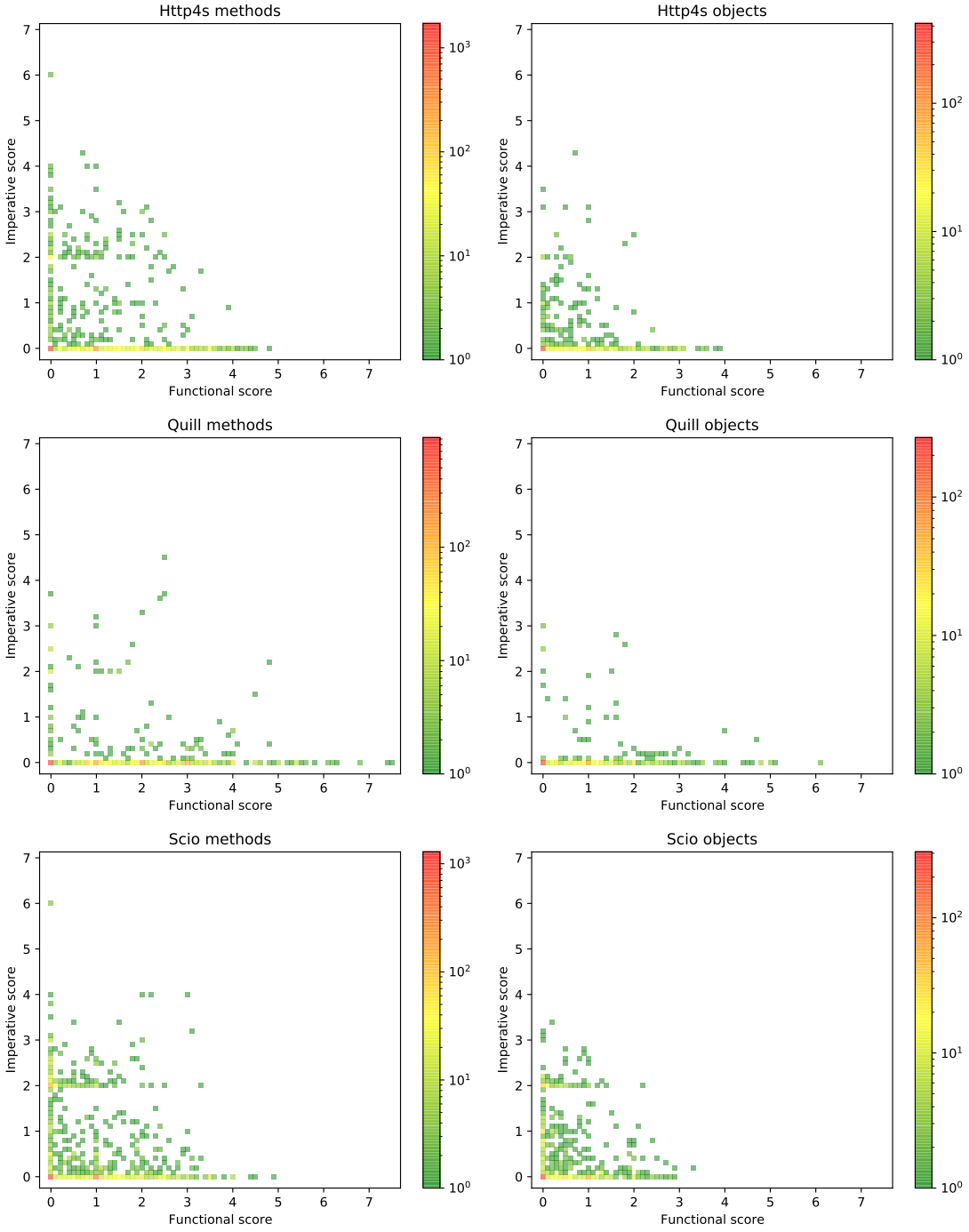


Figure A.1: Fractional paradigm score scatter plots. Color indicates number of occurrences.

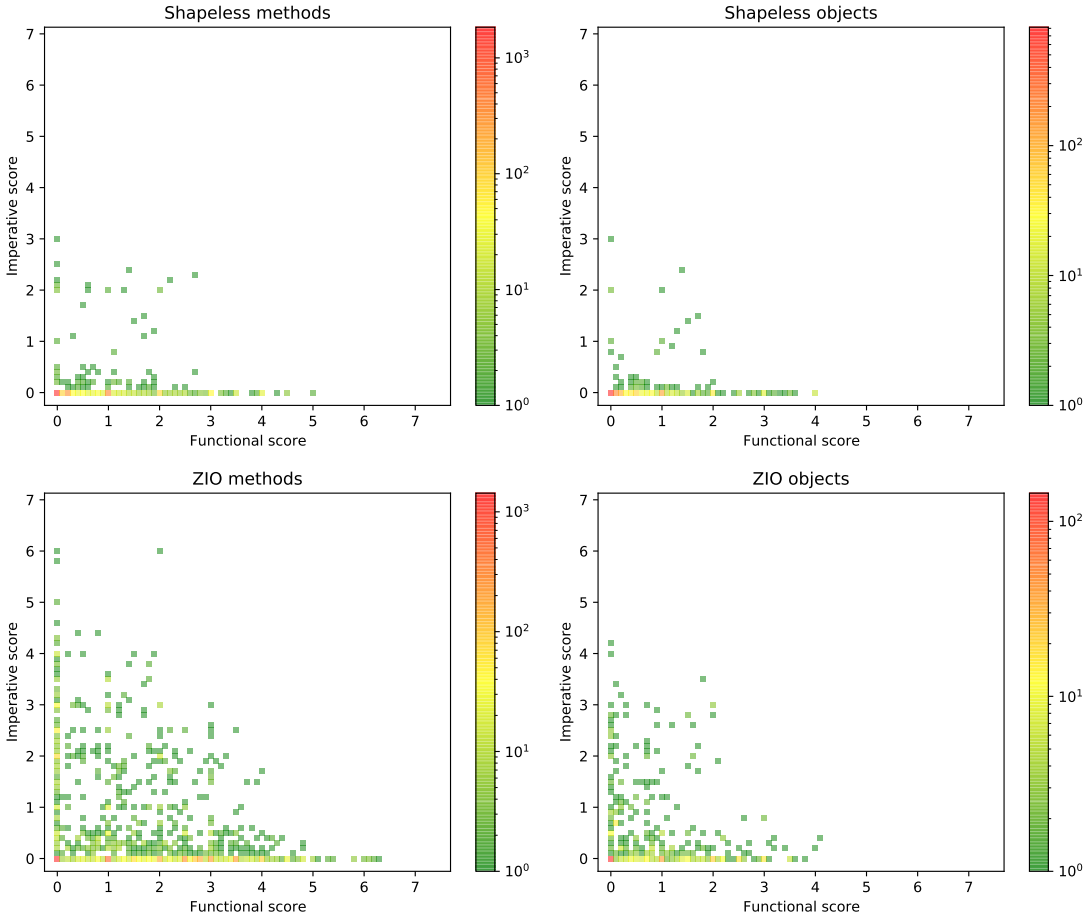


Figure A.1: Fractional paradigm score scatter plots. Color indicates number of occurrences.

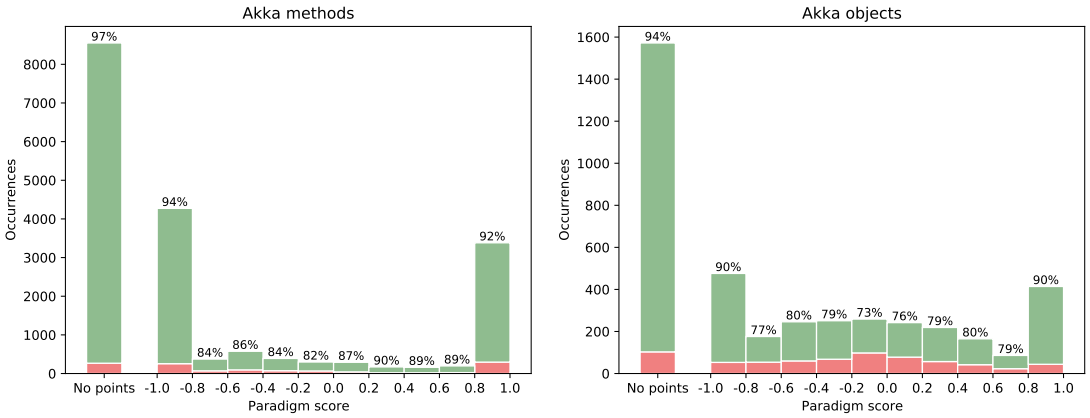


Figure A.2: Fractional paradigm score histograms. Red (bottom color) indicates faulty code, green (top color) indicates non-faulty code. Percentage indicates the percentage of non-faulty code.

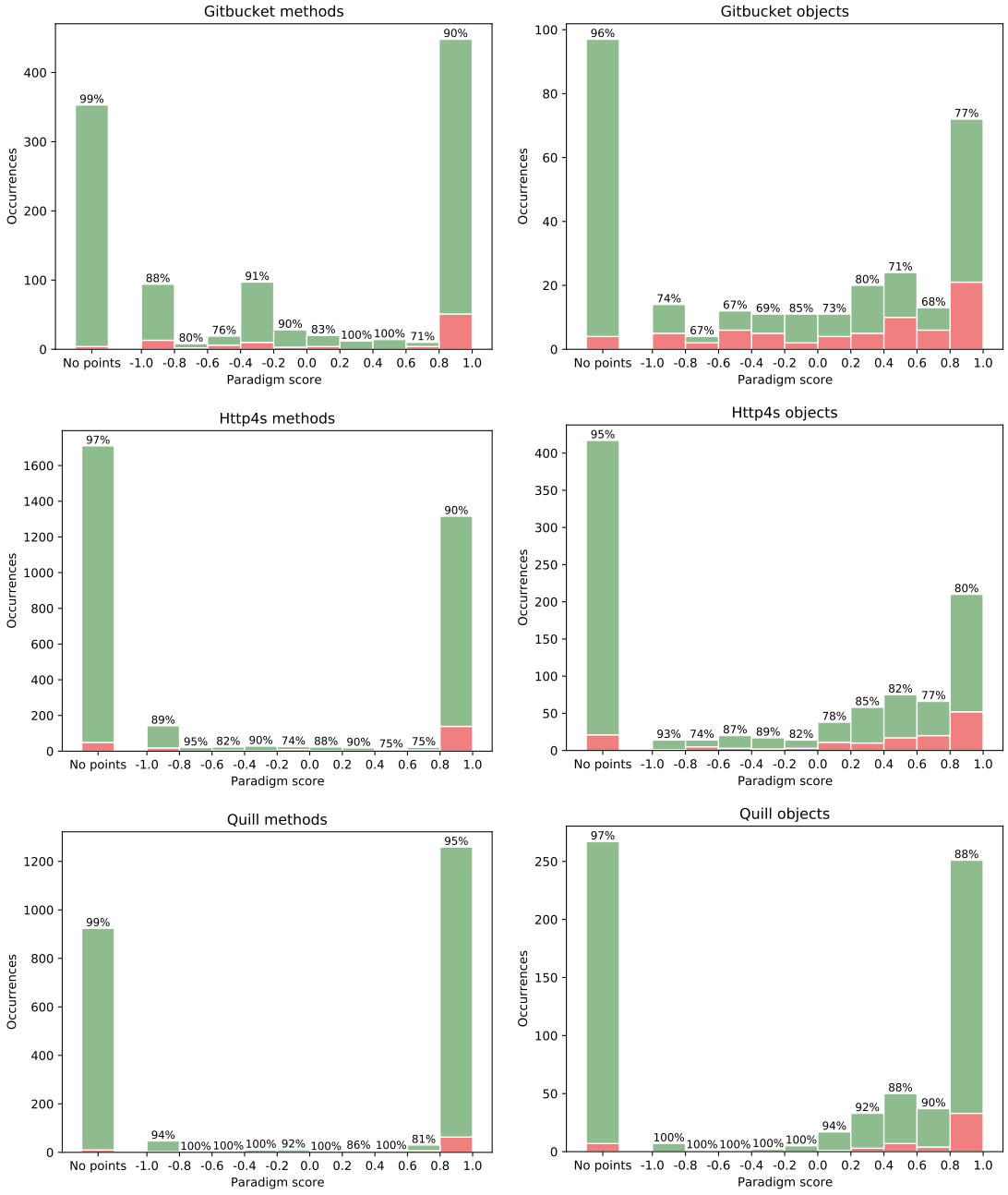


Figure A.2: Fractional paradigm score histograms. Red (bottom color) indicates faulty code, green (top color) indicates non-faulty code. Percentage indicates the percentage of non-faulty code.

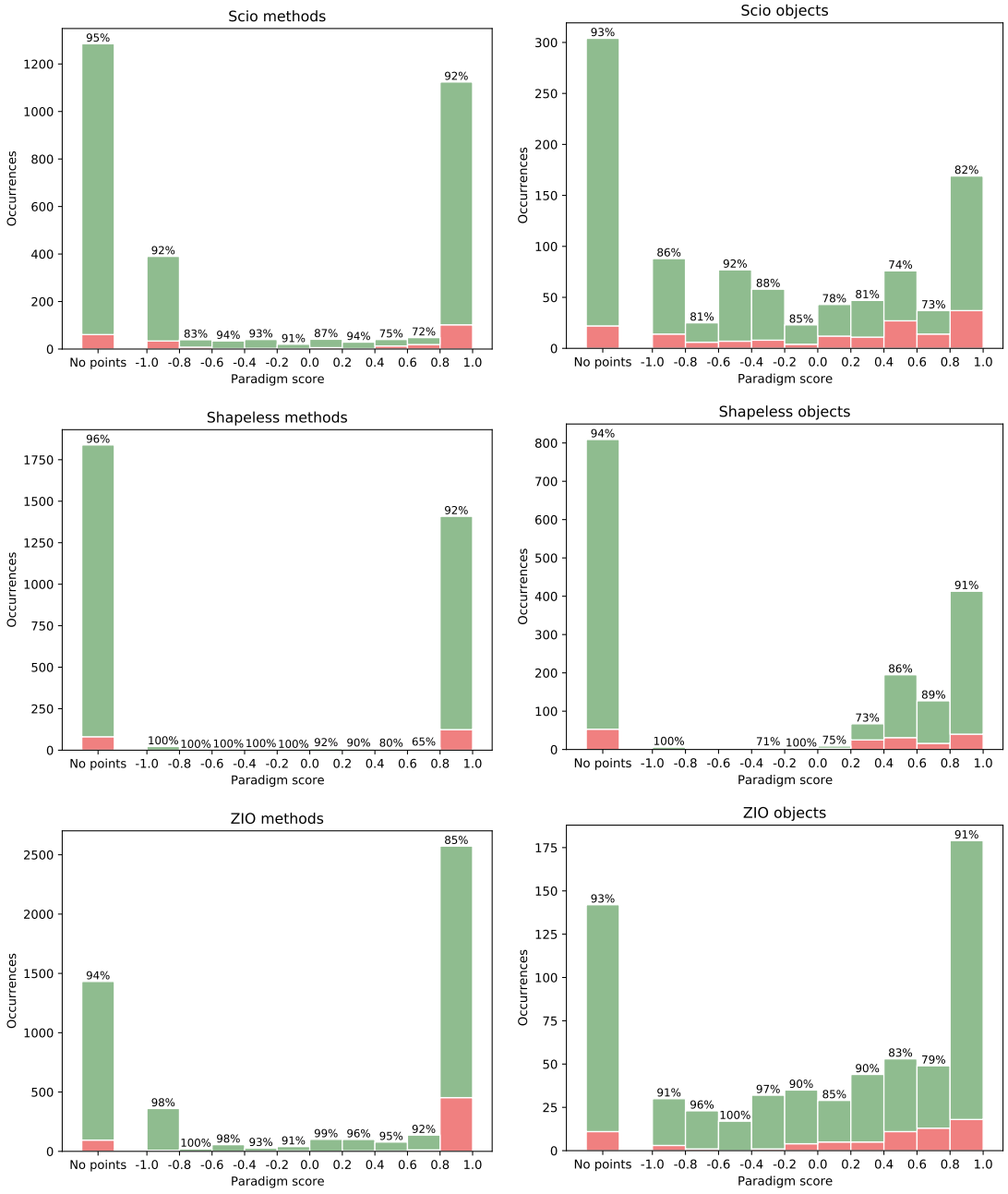


Figure A.2: Fractional paradigm score histograms. Red (bottom color) indicates faulty code, green (top color) indicates non-faulty code. Percentage indicates the percentage of non-faulty code.

Appendix B

Construct measurement results

This appendix contains the average fault-proneness prediction performance results over all the projects. The results for each individual project can be found online at `https://github.com/svenkonings/ScalaMetrics/tree/master/data/analysisResults`.

B.1 Briand's methodology

Name	Precision mean	Precision std.	Recall mean	Recall std.	MCC mean	MCC std.
Multivariate regression	15.32	4.32	61.22	10.52	0.189	0.031
HasOuterVariableUsage	18.19	15.09	48.34	48.41	0.063	0.040
HasVariables	17.95	8.20	22.86	34.43	0.070	0.042
HasVariableDefinitions	17.64	7.26	17.90	35.76	0.045	0.025
HasSideEffectFunctions	17.26	11.61	23.70	31.94	0.049	0.069
HasInnerVariableAssignment	16.64	6.85	18.34	35.55	0.047	0.028
HasSideEffectCalls	15.12	3.00	32.85	31.21	0.100	0.034
HasPatternMatching	15.08	4.12	32.15	8.87	0.122	0.040
HasSideEffects	14.56	3.85	39.64	26.95	0.105	0.023
HasLazyValues	14.45	6.65	29.15	44.79	0.030	0.027
ImperativeScoreBool	14.36	3.72	39.58	26.79	0.106	0.012
HasHigherOrderCalls	14.27	5.36	42.89	15.73	0.128	0.052
HasNestedMethods	14.21	6.15	11.60	5.72	0.054	0.049
HasFunctionCalls	13.77	4.97	38.51	41.96	0.066	0.041
FunctionalScoreBool	13.44	3.18	52.97	7.71	0.143	0.023
HasMultipleParameterLists	13.10	6.68	42.38	26.51	0.099	0.074
HasFunctions	13.01	3.76	50.32	16.33	0.126	0.036
HasFunctionParameters	12.06	4.61	24.51	19.39	0.067	0.074
IsNested	11.17	4.09	30.60	46.61	0.028	0.012
HasPointsBool	10.40	2.76	80.58	10.43	0.129	0.035
HasOuterVariableAssignment	10.37	5.24	54.20	45.22	0.033	0.047
IsSideEffect	10.37	3.95	46.34	38.96	0.045	0.041
IsFunction	10.16	3.08	61.84	47.98	0.039	0.020
ParadigmScoreBool	9.70	4.23	59.01	10.32	0.067	0.076
HasCurrying	9.36	6.67	34.55	44.32	0.030	0.045
IsRecursive	9.04	4.36	36.75	43.57	0.011	0.023

Table B.1: Boolean construct measurements prediction performance.

Name	Precision mean	Precision std.	Recall mean	Recall std.	MCC mean	MCC std.
Multivariate regression	14.94	5.47	58.32	10.25	0.177	0.047
CountOuterVariableUsage	18.19	15.09	48.34	48.41	0.063	0.040
CountVariables	17.95	8.20	22.86	34.43	0.070	0.042
CountVariableDefinitions	16.48	8.25	18.87	35.40	0.038	0.032
CountInnerVariableAssignment	16.46	7.00	7.06	6.03	0.039	0.037
CountSideEffectFunctions	16.20	12.18	20.81	30.86	0.045	0.071
CountSideEffects	16.05	3.46	32.96	20.98	0.112	0.046
ImperativeScoreCount	15.94	3.20	35.90	27.17	0.121	0.024
CountSideEffectCalls	15.89	2.75	31.05	30.37	0.106	0.044
CountPatternMatching	15.08	4.12	32.15	8.87	0.122	0.040
FunctionalScoreCount	14.76	3.87	47.53	6.84	0.155	0.032
CountHigherOrderCalls	14.63	5.63	37.66	9.51	0.124	0.050
CountNestedMethods	14.40	5.87	11.47	5.41	0.058	0.043
CountFunctions	13.91	2.96	42.32	5.27	0.130	0.041
CountFunctionCalls	13.77	4.97	38.51	41.96	0.066	0.041
CountParameterLists	13.10	6.68	42.38	26.51	0.099	0.074
CountLazyValues	12.50	5.97	31.77	46.41	0.024	0.027
CountFunctionParameters	12.06	4.61	24.51	19.39	0.067	0.074
CountNestedDepth	11.17	4.09	30.60	46.61	0.028	0.012
HasPointsCount	10.43	2.77	80.56	10.42	0.130	0.036
CountOuterVariableAssignment	10.40	5.23	61.34	48.15	0.034	0.046
ParadigmScoreCount	10.13	3.77	58.83	13.52	0.087	0.053
CountCurrying	9.36	6.67	34.55	44.32	0.030	0.045
CountRecursiveCalls	9.16	4.21	45.53	49.50	0.015	0.019

Table B.2: Count construct measurements prediction performance.

Name	Precision mean	Precision std.	Recall mean	Recall std.	MCC mean	MCC std.
Multivariate regression	13.42	4.33	64.35	9.76	0.166	0.040
FractionSideEffectFunctions	17.41	11.80	33.56	42.19	0.050	0.068
FractionOuterVariableUsage	16.49	15.49	60.88	47.19	0.044	0.059
FractionSideEffectCalls	14.92	2.88	31.02	31.00	0.094	0.028
FractionPatternMatching	14.84	4.44	29.01	7.60	0.112	0.044
FractionVariableDefinitions	14.64	8.40	30.79	42.55	0.027	0.040
FractionVariables	13.91	9.82	43.01	39.61	0.033	0.071
FractionNestedMethods	13.72	6.32	12.26	4.93	0.052	0.049
FractionHigherOrderCalls	13.41	5.73	35.67	9.05	0.100	0.066
FractionFunctions	12.20	4.07	40.62	15.96	0.100	0.050
FractionSideEffects	11.93	3.02	35.95	26.38	0.078	0.036
ImperativeScoreFraction	11.70	3.04	36.82	25.92	0.077	0.040
FunctionalScoreFraction	11.69	3.35	55.36	5.25	0.115	0.028
FractionInnerVariableAssignment	11.17	5.33	20.25	34.78	0.024	0.028
FractionFunctionCalls	10.83	3.81	51.58	42.63	0.038	0.066
HasPointsFraction	10.40	2.76	80.58	10.43	0.129	0.035
ParadigmScoreFraction	10.09	4.09	64.10	11.42	0.084	0.079
FractionCurrying	9.49	6.64	47.48	48.68	0.031	0.046
FractionOuterVariableAssignment	9.32	4.86	62.60	46.42	0.030	0.046
FractionRecursiveCalls	7.50	2.63	71.63	42.20	0.007	0.021
FractionLazyValues	7.49	2.47	73.29	39.49	−0.007	0.017

Table B.3: Fractional construct measurements prediction performance.

Name	Precision mean	Precision std.	Recall mean	Recall std.	MCC mean	MCC std.
Multivariate regression	13.91	2.84	52.14	7.40	0.152	0.021
SideEffects	17.95	8.20	22.86	34.43	0.070	0.042
ImperativeScoreLandkroon	14.65	4.71	23.32	31.60	0.062	0.052
FunctionalCalls	14.23	3.77	40.38	7.55	0.128	0.038
FunctionalScoreLandkroon	13.93	3.03	44.86	8.11	0.137	0.021
ImperativeCalls	13.60	6.01	20.46	35.28	0.038	0.036
ParadigmScoreLandkroon	13.11	3.37	50.82	12.94	0.132	0.025
HasPointsLandkroon	12.77	3.09	57.80	9.54	0.138	0.031
HigherOrder	12.06	4.61	24.51	19.39	0.067	0.074
Nested	11.17	4.09	30.60	46.61	0.028	0.012
Recursive	9.22	4.25	42.51	44.47	0.013	0.021

Table B.4: Landkroon construct measurements prediction performance.

B.2 Landkroon's methodology

Name	Precision mean	Precision std.	Recall mean	Recall std.	MCC mean	MCC std.
Multivariate regression	13.87	5.78	57.77	8.29	0.155	0.057
HasPatternMatching	15.16	7.66	36.14	9.84	0.125	0.088
HasNestedMethods	14.88	8.49	13.80	6.40	0.065	0.045
HasVariableDefinitions	14.59	10.44	26.44	40.47	0.028	0.032
HasLazyValues	14.20	5.75	19.50	36.04	0.032	0.024
HasSideEffectFunctions	14.00	4.54	20.46	35.55	0.044	0.043
HasOuterVariableUsage	13.19	6.74	46.20	44.73	0.041	0.039
HasInnerVariableAssignment	12.23	8.27	29.03	41.59	0.020	0.030
HasVariables	12.21	6.01	23.84	33.98	0.041	0.040
FunctionalScoreBool	11.71	4.69	54.23	10.19	0.106	0.049
HasFunctionCalls	11.51	6.16	37.03	41.35	0.039	0.036
HasSideEffects	11.10	4.42	32.07	18.01	0.055	0.052
HasMultipleParameterLists	11.10	5.36	39.01	30.71	0.063	0.063
ImperativeScoreBool	11.06	4.59	45.89	28.77	0.052	0.051
IsFunction	11.06	7.31	61.48	48.27	0.040	0.045
HasSideEffectCalls	10.88	5.76	47.67	29.97	0.050	0.049
HasFunctions	10.76	4.57	52.95	15.60	0.075	0.047
HasOuterVariableAssignment	10.51	6.42	33.22	39.72	0.017	0.028
HasHigherOrderCalls	10.32	3.80	50.75	21.04	0.057	0.036
HasPointsBool	10.06	4.38	78.80	9.39	0.100	0.050
ParadigmScoreBool	9.79	4.74	59.45	9.28	0.062	0.065
HasFunctionParameters	9.36	3.93	14.24	11.60	0.018	0.017
IsRecursive	9.14	3.93	44.49	47.65	0.013	0.017
IsSideEffect	9.02	3.19	67.89	38.39	0.034	0.033
IsNested	8.95	3.10	49.28	47.13	0.015	0.015
HasCurrying	7.31	3.79	37.27	42.78	0.003	0.013

Table B.5: Boolean construct measurements prediction performance.

Name	Precision mean	Precision std.	Recall mean	Recall std.	MCC mean	MCC std.
Multivariate regression	14.61	6.44	55.67	7.25	0.160	0.069
CountNestedMethods	14.88	8.49	13.80	6.40	0.065	0.045
CountPatternMatching	14.82	7.79	37.56	9.80	0.117	0.100
CountSideEffectFunctions	14.00	4.54	20.46	35.55	0.044	0.043
CountLazyValues	13.32	6.60	19.42	35.68	0.029	0.027
CountOuterVariableUsage	13.18	6.75	38.03	43.15	0.042	0.038
CountVariableDefinitions	12.54	10.28	20.67	35.01	0.021	0.036
CountSideEffects	12.29	4.07	34.59	28.58	0.066	0.055
FunctionalScoreCount	12.25	4.82	40.17	8.78	0.097	0.071
ImperativeScoreCount	12.13	4.26	34.17	28.82	0.064	0.053
CountVariables	11.55	6.73	26.20	33.78	0.034	0.050
CountFunctionCalls	11.51	6.16	37.03	41.35	0.039	0.036
CountInnerVariableAssignment	11.44	7.29	31.03	46.78	0.019	0.032
CountSideEffectCalls	11.14	5.59	35.70	29.97	0.054	0.051
CountParameterLists	11.10	5.36	39.01	30.71	0.063	0.063
CountOuterVariableAssignment	10.57	6.36	44.69	46.01	0.019	0.027
CountHigherOrderCalls	10.19	3.44	31.25	5.69	0.044	0.057
CountFunctions	10.18	3.72	32.69	7.56	0.052	0.056
HasPointsCount	10.07	4.38	78.76	9.40	0.101	0.050
ParadigmScoreCount	10.05	4.35	59.88	9.75	0.075	0.047
CountFunctionParameters	9.50	3.89	13.28	11.60	0.020	0.014
CountRecursiveCalls	9.23	3.96	56.95	43.76	0.012	0.018
CountNestedDepth	8.95	3.10	49.28	47.13	0.015	0.015
CountCurrying	7.35	3.90	32.33	36.55	0.001	0.013

Table B.6: Count construct measurements prediction performance.

Name	Precision mean	Precision std.	Recall mean	Recall std.	MCC mean	MCC std.
Multivariate regression	13.01	5.12	59.44	7.83	0.144	0.050
FractionNestedMethods	14.78	8.72	25.29	29.59	0.062	0.050
FractionSideEffectFunctions	14.00	4.56	20.39	35.57	0.043	0.043
FractionPatternMatching	13.75	5.53	30.95	7.04	0.098	0.055
FractionVariableDefinitions	12.66	10.22	56.85	49.10	0.019	0.038
FunctionalScoreFraction	10.97	4.25	52.64	6.53	0.091	0.041
FractionInnerVariableAssignment	10.67	7.56	56.41	42.82	0.011	0.035
FractionSideEffectCalls	10.52	5.33	56.37	39.09	0.044	0.039
FractionLazyValues	10.44	4.26	61.34	46.74	0.002	0.041
FractionFunctions	10.23	3.84	44.87	18.26	0.057	0.031
FractionHigherOrderCalls	10.11	3.35	47.74	23.85	0.052	0.026
ParadigmScoreFraction	10.11	4.99	62.60	10.15	0.075	0.072
HasPointsFraction	10.06	4.38	78.80	9.39	0.100	0.050
FractionVariables	9.78	6.52	63.54	37.08	0.014	0.041
FractionSideEffects	9.75	4.06	52.25	29.33	0.037	0.049
FractionOuterVariableUsage	9.73	5.82	53.41	43.17	0.024	0.025
FractionFunctionCalls	9.69	5.87	68.40	38.32	0.013	0.052
ImperativeScoreFraction	9.57	4.34	66.26	28.96	0.026	0.063
FractionOuterVariableAssignment	8.87	2.99	61.55	47.71	0.014	0.022
FractionRecursiveCalls	8.23	2.63	72.16	44.65	0.011	0.017
FractionCurrying	7.44	3.95	57.60	49.85	0.005	0.011

Table B.7: Fractional construct measurements prediction performance.

Name	Precision mean	Precision std.	Recall mean	Recall std.	MCC mean	MCC std.
Multivariate regression	12.37	4.85	46.84	15.37	0.111	0.064
FunctionalCalls	13.57	5.10	38.07	7.85	0.106	0.085
ImperativeCalls	12.87	5.93	31.75	40.26	0.044	0.039
FunctionalScoreLandkroon	12.29	3.71	44.14	7.56	0.103	0.064
ImperativeScoreLandkroon	11.92	5.79	29.63	32.02	0.048	0.052
ParadigmScoreLandkroon	11.59	4.36	48.34	10.46	0.093	0.064
SideEffects	11.55	6.73	26.20	33.78	0.034	0.050
HasPointsLandkroon	11.34	4.23	55.46	9.51	0.099	0.064
HigherOrder	9.50	3.89	13.28	11.60	0.020	0.014
Recursive	9.24	3.89	44.55	47.59	0.014	0.017
Nested	8.95	3.10	49.28	47.13	0.015	0.015

Table B.8: Landkroon construct measurements prediction performance.

Appendix C

Baseline model average MCC per paradigm

This appendix contains the average Matthews Correlation Coefficient of the fault-proneness prediction performance results per paradigm over all the projects. The full fault-proneness prediction performance results can be found online at `https://github.com/svenkonings/ScalaMetrics/tree/master/data/analysisResults/baseline/split-regression`.

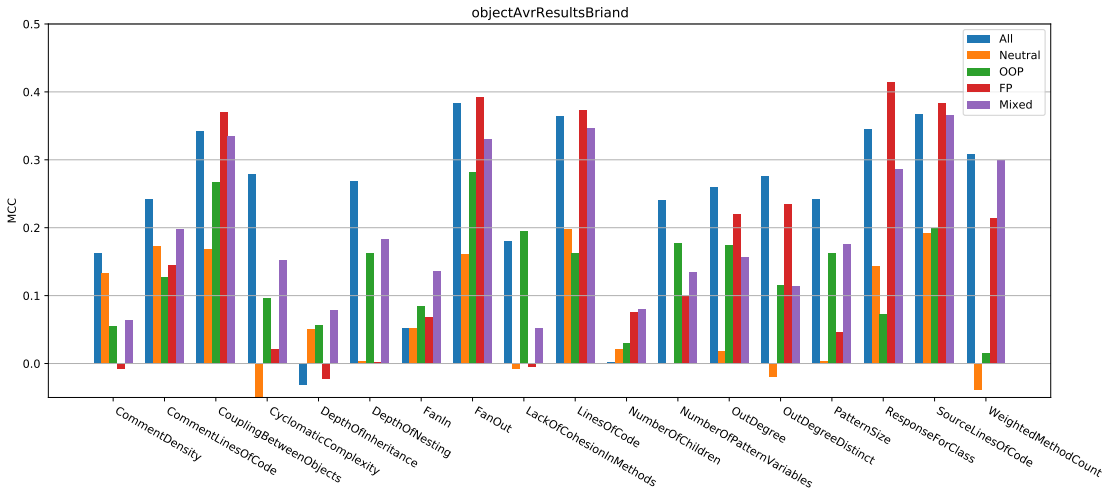


Figure C.1: Average Matthews Correlation Coefficient barcharts

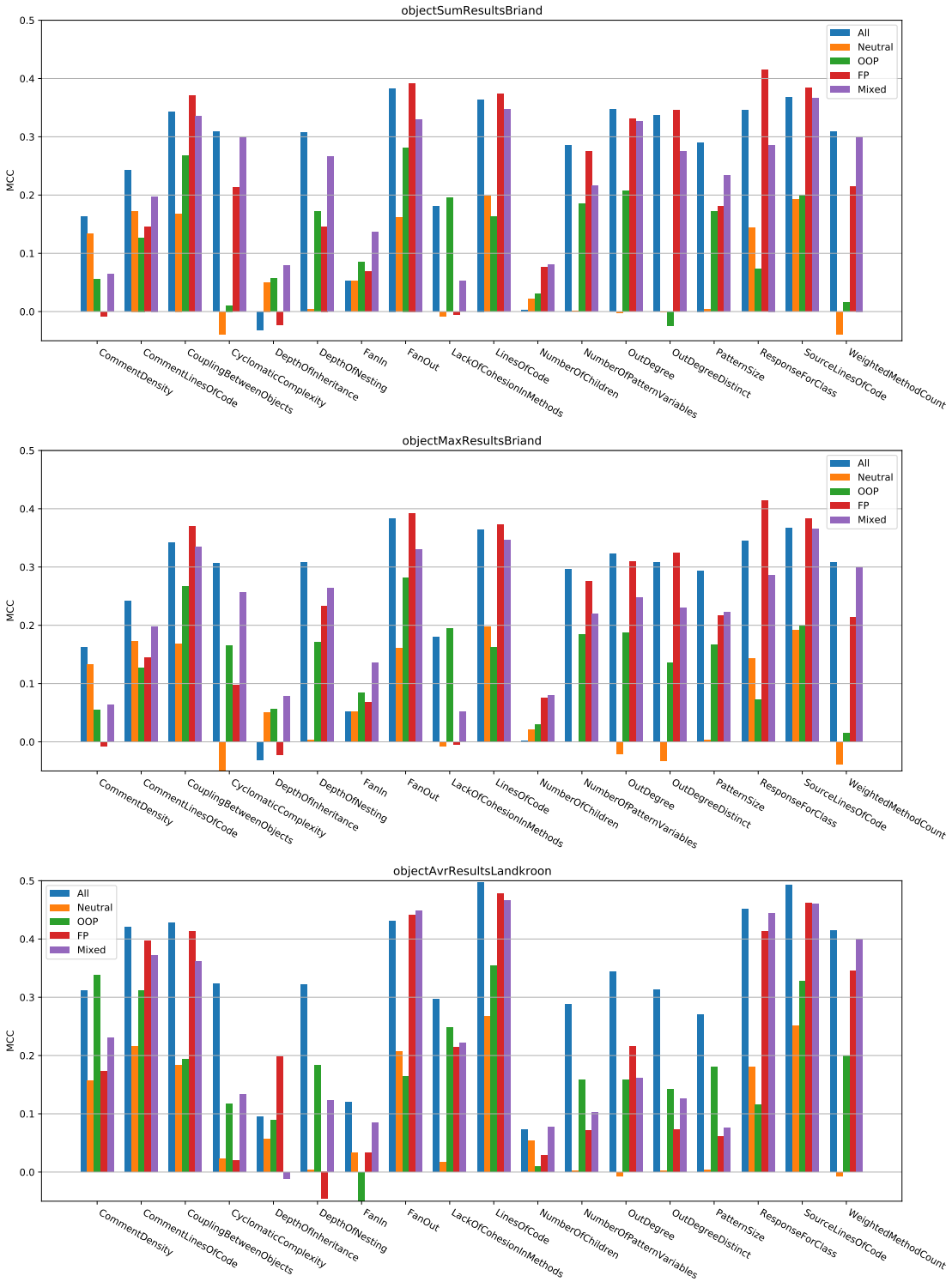


Figure C.1: Average Matthews Correlation Coefficient barcharts.

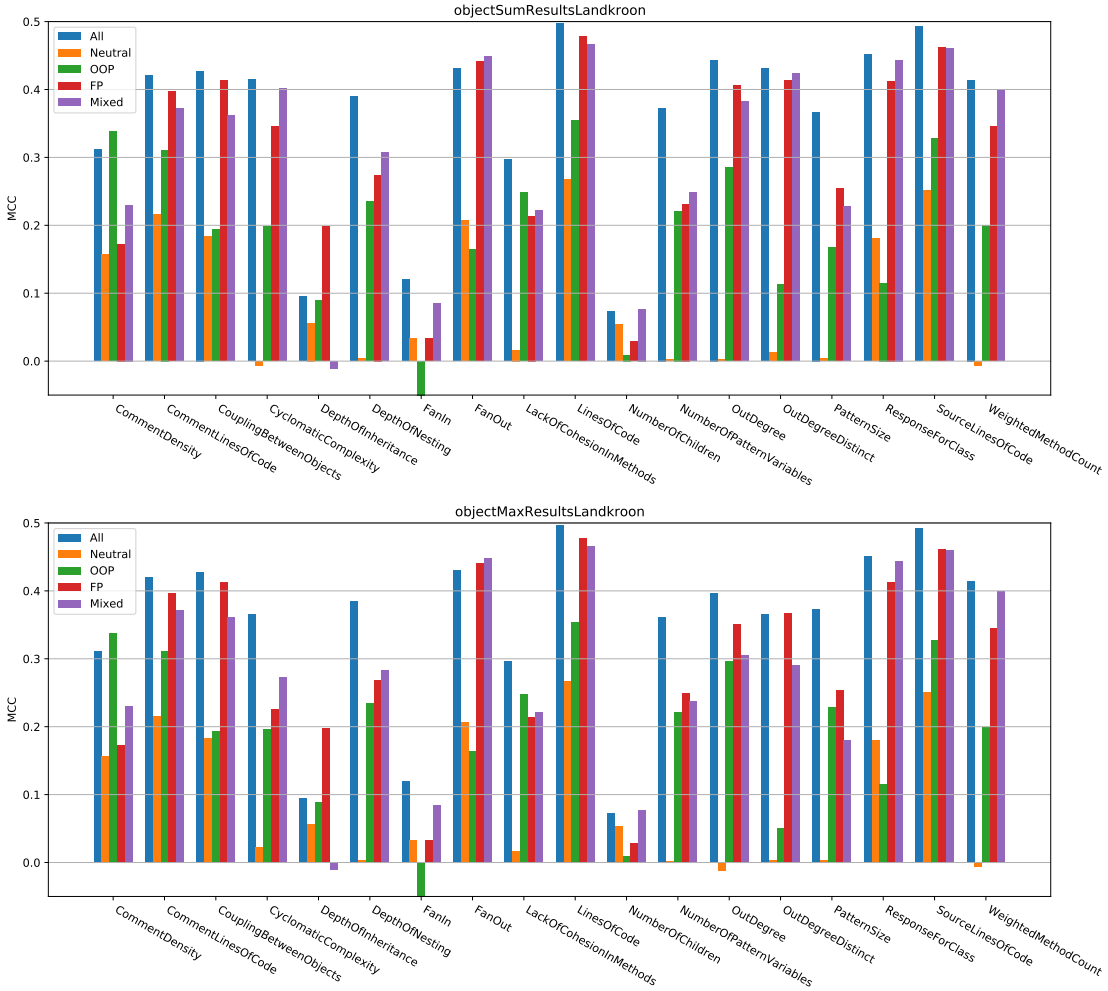


Figure C.1: Average Matthews Correlation Coefficient barcharts.

Appendix D

Multivariate baseline regression for objects with metric results

This appendix contains the results for the multivariate regression of the candidate metrics added to the baseline model, when only considering objects for which the candidate metric has data (see Table D.1). The full results can be found online at:

1. Zuilhof's metrics
   (a) `https://github.com/svenkonings/ScalaMetrics/tree/master/data/analysisResults/multiparadigm-zuilhof/regression/multivariate-baseline-hasdata`
2. Construct metrics
   (a) `https://github.com/svenkonings/ScalaMetrics/tree/master/data/analysisResults/multiparadigm-constructs/regression/multivariate-baseline-hasdata`

Name	Count	Precision mean	Precision std.	Recall mean	Recall std.	MCC mean	MCC std.
Briand's methodology							
NumberOfLambdaFunctions baseline	7	48.57	10.46	63.00	7.16	0.364	0.072
NumberOfLambdaFunctions	7	48.05	10.03	62.38	7.07	0.357	0.062
SourceLinesOfLambda baseline	7	48.57	10.46	63.00	7.16	0.364	0.072
SourceLinesOfLambda	7	47.85	9.66	62.81	6.80	0.356	0.071
LambdaScore baseline	7	48.57	10.46	63.00	7.16	0.364	0.072
LambdaScore	7	48.48	10.39	62.32	7.24	0.361	0.065
LambdaFunctionsUsingOuterVariables baseline	1	70.89		76.71		0.465	
LambdaFunctionsUsingOuterVariables	1	67.11		69.86		0.370	
LambdaFunctionsUsingLocalVariables baseline	1	57.14		64.00		0.284	
LambdaFunctionsUsingLocalVariables	1	62.07		72.00		0.392	
LambdaFunctionsWithSideEffects baseline	5	55.63	6.69	54.99	6.91	0.216	0.157
LambdaFunctionsWithSideEffects	5	57.14	9.51	55.18	7.67	0.235	0.195
LambdaFunctionsWithAssignment baseline	1	68.06		63.64		0.346	
LambdaFunctionsWithAssignment	1	71.83		66.23		0.411	
ImplicitConversions baseline	7	40.51	17.36	54.22	12.65	0.262	0.169
ImplicitConversions	7	40.55	16.61	54.75	12.48	0.264	0.164
ImplicitDefinitions baseline	7	50.63	9.81	64.63	5.91	0.400	0.090
ImplicitDefinitions	7	51.61	8.69	64.05	4.19	0.407	0.077
ImplicitParameters baseline	7	52.16	10.86	58.98	4.63	0.315	0.097
ImplicitParameters	7	52.91	8.26	59.57	2.36	0.327	0.060
UnitVariables baseline	1	75.00		69.23		0.647	
UnitVariables	1	75.00		69.23		0.647	
FunctionVariables baseline	7	47.47	11.56	55.93	11.23	0.273	0.212
FunctionVariables	7	48.11	9.42	57.05	8.88	0.285	0.181
UsageOfNull baseline	7	50.38	23.79	53.62	18.59	0.228	0.286
UsageOfNull	7	52.57	22.10	53.45	16.37	0.250	0.281
NumberOfReturns baseline	0						
NumberOfReturns	0						
OverridingPatternVariables baseline	6	59.59	10.80	63.13	4.54	0.314	0.125
OverridingPatternVariables	6	58.27	10.34	62.64	5.12	0.297	0.115
Landkroon's methodology							
NumberOfLambdaFunctions baseline	7	60.53	11.47	68.75	6.39	0.462	0.102
NumberOfLambdaFunctions	7	60.32	12.28	68.76	7.18	0.461	0.112
SourceLinesOfLambda baseline	7	60.53	11.47	68.75	6.39	0.462	0.102
SourceLinesOfLambda	7	60.21	11.97	67.84	6.28	0.455	0.099
LambdaScore baseline	7	60.53	11.47	68.75	6.39	0.462	0.102
LambdaScore	7	60.61	11.72	67.92	6.40	0.459	0.098
LambdaFunctionsUsingOuterVariables baseline	2	68.19	1.47	75.17	9.40	0.463	0.000
LambdaFunctionsUsingOuterVariables	2	68.79	0.62	76.79	7.11	0.479	0.023
LambdaFunctionsUsingLocalVariables baseline	2	71.68	8.62	66.48	4.02	0.373	0.126
LambdaFunctionsUsingLocalVariables	2	71.25	12.37	70.45	3.21	0.385	0.230
LambdaFunctionsWithSideEffects baseline	5	69.20	14.09	67.22	5.67	0.332	0.053
LambdaFunctionsWithSideEffects	5	70.49	15.16	68.49	5.89	0.358	0.093
LambdaFunctionsWithAssignment baseline	2	75.39	23.93	80.79	13.03	0.399	0.021
LambdaFunctionsWithAssignment	2	75.52	23.74	79.28	15.16	0.393	0.029
ImplicitConversions baseline	7	64.18	18.74	69.76	9.63	0.487	0.150
ImplicitConversions	7	63.63	18.38	68.56	10.57	0.476	0.156
ImplicitDefinitions baseline	7	62.30	13.93	68.81	6.38	0.468	0.082
ImplicitDefinitions	7	61.37	13.15	68.74	6.31	0.458	0.064
ImplicitParameters baseline	7	66.94	13.05	70.16	7.62	0.479	0.089
ImplicitParameters	7	66.24	13.68	69.49	7.59	0.469	0.091
UnitVariables baseline	2	70.67	23.80	74.62	8.69	0.584	0.251
UnitVariables	2	69.73	20.18	76.25	6.39	0.586	0.207
FunctionVariables baseline	7	63.30	12.80	67.68	8.71	0.477	0.080
FunctionVariables	7	62.81	12.30	68.28	7.39	0.475	0.062
UsageOfNull baseline	7	63.93	12.02	71.61	7.36	0.415	0.130
UsageOfNull	7	65.56	11.33	70.91	6.61	0.436	0.094
NumberOfReturns baseline	0						
NumberOfReturns	0						
OverridingPatternVariables baseline	7	71.37	13.82	74.17	10.43	0.459	0.138
OverridingPatternVariables	7	71.39	13.34	75.01	8.65	0.466	0.114

Table D.1: Multivariate regression fault-proneness prediction performance for objects with metric results.