Code Quality Evaluation for the Multi-Paradigm Programming Language Scala

Erik Landkroon
eriklandkroon@gmail.com

August 18, 2017, 65 pages

Host Supervisor: Rinse van Hees
Host organisation: Info Support, https://www.infosupport.com/
Academic supervisor: Clemens Grelck

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Contents

Abstract

Code metrics are used to measure properties of the source code. Using these measurements, estimations / statements can be made about the maintainability, fault-proneness and quality of the code. Code metrics are often designed for a specific programming paradigm. Metrics suites have been presented and validated for both the object-oriented and for the functional programming paradigm. However, Scala combines both the object-oriented and the functional programming paradigm. Therefore, we cannot assume, without proper validation, that there is a significant relation between the metrics and the code quality.

In this study, we investigate a relation between the fault-proneness of classes and the code metrics of an object-oriented and functional metric suite, as well as some general code metrics. We present an implementation, for each of the metrics selected for our research, for the Scala programming language. We present some Scala specific code metrics as well. Furthermore, we present our own novel improved validation methodology, based on an existing validation methodology.

Our results suggest that our 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 the existing methodology. Furthermore, the results suggest that there is a significant relation between the fault-proneness of classes and most of the metrics.

Chapter 1

Introduction

With the source code of (industrial) projects becoming larger, it becomes increasingly rewarding to have automated systems to detect code which might be fault-prone. These systems can help detect code that might need additional attention to reduce the possibility of faults within the code. These automated systems often make use of code metrics.

Code metrics are used to measure specific aspects of a software system. These metrics help to gain insight in the system, by expressing them using a value. Code metrics commonly measure properties such as the size and complexity of the code. These metrics can be used to make better estimations about the complexity, maintainability, quality and fault-proneness of the code and therefore help to increase or reduce them. This is especially useful for large software systems, were reviewing the code is time consuming.

Code metrics are often designed for a specific programming paradigm. Most of the commonly used programming languages follow the functional or object-oriented paradigm. For these paradigms, different metric suites have been proposed. The object-oriented metric suite from Chidamber et al. [1] is considered to be the most popular metric suite for object-oriented languages, which has already been researched and validated [2, 3, 4, 5, 6]. The functional metric suite proposed by Ryder et al. [7], is designed and validated for Haskell, a functional programming language.

However, there are an increasing amount of programming languages that combine multiple programming paradigms, such as Scala, OCaml and F#. We focus in our research on the Scala programming language. Scala is an object-oriented functional programming language [8]. We cannot simply assume that the metrics, proposed and validated for the functional and object-oriented paradigms, are useful measurements of code quality for Scala, a multi paradigm programming language.

Scala is designed to unify the object-oriented and functional paradigms. In Scala, every value is an object. Functions are first-class values in Scala, which means they are treated like any other value [8]. Therefore, functions can be passed as arguments to and be returned by other functions (*higher-order functions*). Scala also support other principles from the functional paradigm, such as pattern matching, anonymous functions, currying and immutable values [8].

For our study, we used both object-oriented and functional metrics, including some paradigm independent metrics. The object-oriented metrics are measured on object level and the functional metrics on function level. Therefore, we studied how to map these metrics to the same level. As well as how to implement them for Scala.

Before we can assume that these metrics are useful for Scala, we need to validate them. For the validation we use a method presented by Briand et al. [9]. This method investigates the relation between the code metrics and the fault-proneness of classes, by using the metrics as fault predictors. The assumption is that classes that have poor code quality, are more likely to contain faults. This is a commonly used method for the validation of code metrics [2, 3, 4, 5, 6]. This method uses information about faults and the metric values to construct a prediction model, often using (logistic) regression. This prediction model can be used to make prediction about the fault-proneness of classes. Accurate predictions suggest a strong correlation between the metric and the fault-proneness. It is assumed that when a metric can be used to predict fault-prone classes, it has a relation with the fault-proneness of the classes.

Briand's validation method uses the latest version of the code to calculate the metric values. This means the version of class for which the metric values are calculated, is not necessarily the version of the class which is affected by a fault. This has as a consequence, that the classes in the latest version, should be (almost) similar to the versions of the classes which are affected by the faults. This can be problematic for projects with longer life-cycles, where it is more likely that classes (completely) changed due refactoring of the code. This would result in less accurate predictions for projects with longer life-cycles. Therefore, the results might suggest that there is no relation between the metrics and the fault-proneness of classes. This would not be a fair representation of the actual relation between the metrics and fault-proneness of classes, since it is influenced by the life time of the project.

We propose our own method for validating metrics. In our proposed method, we use the affected versions, the version before the bug fix, of the classes, rather than the latest version. This guarantees the measured metric values are measured on the version of the class which is affected by a fault. This avoids the problem of classes being refactored between the fault and the latest version of the code.

For our research, we use three open source projects: Gitbucket, Akka and Shadowshock. These projects have different sizes; 592, 1054 and 131 classes respectively. We selected these projects from the list of trending Scala projects on Github.

The goal of this research project is to formulate and validate a set of metrics that can be used to measure the code quality for the Scala programming language. Therefore, this leads to the main research question:

> *RQ: How can code quality be evaluated, using code metrics, for the multi-paradigm programming language Scala?*

To answer this question, we will first need to research the existing code quality metrics for the different programming paradigms that are used by Scala. There are some general metrics that are used to measure code quality across the paradigms. However, these metrics might have different implementations and interpretations for the different paradigms. For example, imperative and functional programming might both measure the length of functions. However, it is possible that different methods are used to calculate the length of a function. Also the interpretation of the results can differ, a function that is considered large in functional programming, might not be large in imperative programming. Therefore, we first need to investigate whether there are metrics across the different paradigms than can be combined.

> *RQ1: Can metrics from multiple paradigms be combined to give a coherent result?*

Because Scala uses different paradigms, it is likely that metrics need to be modified before they can be implemented for Scala. For instance, a metric used in the imperative paradigm might not directly be ported to Scala without modification.

> *RQ2: How can metrics be modified/customized so they can be implemented for the Scala programming language?*

After a set of metrics is formulated for the Scala language, it is important to validate them. Without validation of the metrics we cannot conclude whether the metrics can be used to evaluate the code quality. To validate the metrics, we need to investigate whether there is a relation between the metrics and the quality of the code.

> *RQ3: Can we validate that there is a relation between the metrics and the fault-proneness of classes for the Scala programming language?*

In Chapter 2 we will introduce Scala, a multi-paradigm programming language. In Chapter 3, we discuss the selected code metrics and their definition as stated in the literature. In Chapter 4, we will discuss Briand's methodology used for validating metrics. In Chapter 5, we will discuss our critique on Briand's methodology and present our own validation methodology. In Chapter 6, the realization of the validation framework is described, as well as the implementation for Scala of the metrics discussed in Chapter 4. In Chapter 7, we will discuss the results of Briand's validation methodology. In Chapter 8, we will discuss the results of our validation methodology. In Chapter 9, we will discuss similar studies and compare the results. In Chapter 10, we will conclude on our research.

Chapter 2

The Multi-Paradigm Language Scala

In this chapter, a brief summary of Scala is given. We address the language design and motivation, as well as an overview of the functionalities which are unique for Scala in comparison to *Java* and *C#*.

2.1 General language design

Martin Odersky started in 2001 with the design of Scala [8, 10]. The motivation behind Scala is the increase in importance of web services and other distributed software systems [8]. Languages like *Java* and *C#*, use a model were mutable data is encapsulated by methods. Computations are done by method calls. By performing remote method calls, the system can be made distributed. Large data is often sent and stored in a tree like structure (e.g. XML). However, this model tend to not scale up very well for distributed systems [8]. The current object-oriented languages are not optimal for analyzing and transforming tree like structures. Also to reduce the damage of component failure, components are often made stateless. Object-oriented languages are often not designed to support these methodologies natively.

Scala tries to address these problems by unifying the object-oriented and functional programming paradigms [8, 10]. The functional paradigm avoids states and side effects, by using, for instance, recursion rather than iteration. The functional paradigm also includes functionalities that offer support for analyzing and transforming tree like structures, for instance pattern matching.

Scala is both an object-oriented and functional language, in the sense that all functions are values and every value is an object [8]. This means that functions are first class values and therefore can be treated as any other value. Furthermore, Scala is compiled to *Java* byte code, and runs therefore on the JVM. Therefore, Scala and Java can be freely mixed. Which allows Scala to use Java libraries.

2.2 Mutable and Immutable

Within Scala some objects have a state, which can change over time. This can be defined as: an object has a state when the behavior is influenced by the history of the object [8]. An example is an object representing a bank account, the value of the account changes over time.

In Scala, there is a distinction between stateful and stateless. Objects that have a state are mutable, their state can change over time. Objects that are stateless are immutable, they are not influenced by their history and therefore don't change over time. To distinguish between immutable and mutable objects, Scala introduced values and variables respectively. A variable definition is the same as a value, but it is called a *var* instead of *val*. The other difference is that a variable (mutable) can change and a value (immutable) cannot. The value given to a val during its deceleration, is its final value and can not be modified. The value of a var however, can be modified by assigning a new value

to the var, using an assign statement.

In pure functional programming, you cannot have side effects. This would mean that (mutable) variables in Scala should be avoided when programming purely functional.

2.3 Nested functions

It is encouraged, in functional programming, to construct functions out of multiple smaller (helper) functions [8]. Many of these helper functions are only relevant and used by the function they were originally made for. Normally, these functions should only be accessed by the original function. To help enforce this (and help to keep the name-space clean), Scala provides the option to nest these helper function within the original function [8]. A nested function looks as follows:

```
def sumSquares(a: Int, b: Int): Int = {
  def square(x: Int): Int = {
      x * x
    }
    if (a > b) 0 else square(a) + sumSquares(a+1, b)
}
```

In the example, the *square* function is nested and can only be used within the scope of the function.

2.4 Recursion and tail recursion

In functional programming, recursion is often used instead of iteration [8]. In recursion, the function invokes itself until a (base) condition is reached. It is a method that is often used in functional programming to help avoid side effects [11]. However, for some recursion, a new stack frame is created for each iteration. This results in an increasing stack size for each iteration and can become a problem with a large number of iterations. This is shown in the following example:

```
def factorial(n: Int): Int = if (n == 0) 1 else n * factorial(n-1)
```

In this example, the factorial function calls itself and multiplies the result with a value. This means that it first needs to calculate the outcome of each function call to itself, before it can calculate the result. Therefore, the value that is used to multiply the result, should be stored on the stack.

Instead, we would like to make the recursive function in such a way that it can be executed in constant space. The above function can be rewritten as follows:

```
def factorial(n: Int): Int = {
    @tailrec
    def factorialRec(n: Int, acc: Int): Int = {
        if (n == 0) acc
        else factorialRec(n-1, n*acc)
    }
    factorialRec(n, 1)
}
```

In this example, the intermediate result is passed as a parameter. This allows Scala to overwrite the previous stack frame of the function call, instead of creating a new one. This is called tail-recursion and allows the function to be executed in constant space [8]. Tail recursion in Scala can be enforced by using the annotation *@tailrec* above the tail-recursive function definition.

2.5 Higher-order function

In Scala, all functions are values [8]. Like any other value, functions can be passed as a parameter and returned as a result. Functions that take functions as parameters or returns them, are called *higher-order functions* [8]. Higher-order functions help to create abstract versions of functions, which can be reused for multiple purposes. An example of a higher-order function is as follows:

```
def sum(f: Int => Int, a: Int, b: Int): Int = {
  if (a > b) 0 else f(a) + sum(f, a+1, b)
}
```

This function takes a function f which takes an integer as parameter and returns an integer. The *sum* function also take a start value a and a stop value b to define the boundaries. If for instance a function is passed as a parameter that returns the square of a value, then the sum function will return the sum of the square of all the values within the interval. However, if a function is passed as a parameter, which simply returns the value, then the sum function will return the sum of the values within the interval.

2.6 Anonymous functions

Passing functions as parameters to other functions, tends to the creation of many small functions [8]. Instead of using named functions for these functions, it is also possible to create anonymous functions. These functions are not named and can directly be inserted into the function call. Anonymous functions can be constructed as follows:

```
(p1: T1, p2: T2, ..., pn: Tn) => E
```

Where $p_1, p_2, \ldots, p_n$ are the parameter names, $T_1, T_2, \ldots, T_n$ are the corresponding parameter types and E is the expression of the anonymous function.

2.7 Currying

Currying is a principle that is used in functional languages to break down a function with multiple arguments, into multiple functions that take a part of the arguments (often one argument per function). This can help to increase the re-usability of functions.

Currying makes use of the fact that functions are first class values, and therefore can be returned as function results. The following code shows a function that is uncurried:

```
def sum(a: Int, b: Int): Int = {
  a + b
}
```

The curried version of the sum function is as follows:

```
def sum(a: Int): Int {
  (b: Int) => a + b
}
```

The curried version of the code can be called as follows:

```
sum(2)(5)
```

We can now use the curried version of the sum function to create a function called add1, that adds one to the argument. The code of the add1 function looks as follows:

```
def add1: (a: Int => Int) = add(1)
```

2.8 Classes, Objects and Traits

Classes in Scala are similar to those of *Java* or *C#*. Every class extends the *Any* (similar to *Object* in *Java*). However, a class can extend additional classes or traits. A class also has one or more constructors (by default an empty constructor). Furthermore, classes contain similar features as Java classes (e.g. *abstract*, *private*, etc.). However, in Scala, there is no such thing as static values (or functions). Therefore, a class can only be used when an instance of the class is created first, using the *new* statement. An instance of a class is often called an object. A class is defined as follows:

```
class ClassName(p1: T1, p2: T2, ..., pn: Tn){
<Methods, values and variables>
}
```

In Scala, it is possible to make an object directly using the object statement, without using the class and new statements. The object statement will create an object which is accessible from anywhere within the code [8]. This means that any class, object or trait, can access and use the methods and instance variables of any object, with the exception of private or protected methods and instance variables. The object statement can be seen as a class and new statement combined, which is executed globally. This means that it is not possible to create multiple instances from the same object statement with the *new* statement. Objects created by the object statement are lazy evaluated, this means that an object is initialized when it is first used. Therefore, objects, created by the object statement, have no fixed execution order, because they are executed when they are used for the first time [8]. Because of this, the object statement cannot have constructor parameters. A object is defined as follows:

```
object ObjectName{
<Methods, values and variables>
}
```

Traits are similar to abstract classes, they add methods or values, or enforce the implementation of those to other classes, objects or traits [8]. However, the major difference is that a class, object or trait can extend multiple traits, while they can only extend one (abstract) class. This is useful to reuse the trait's functionalities in multiple unrelated classes. Therefore, traits are often used to add unrelated functionalities, like utility functions.

```
trait TraitName(p1: T1, p2: T2, ..., pn: Tn){
<Methods, values and variables>
}
```

2.9 Pattern matching

Pattern matching is a generalization of the switch statement to class hierarchies [8]. In the *Any* class, the root class in Scala, a method named match is defined. This method takes in a number of cases. Each case consist of a pattern and an expression. The selector value will be matched against these patterns until a matching pattern is found. Pattern matching in Scala is constructed as follows [8]:

```
e match { case p1 => e1 ... case pn => en }
```

Where e is the selector value, $p_1 \dots p_n$ are the patterns and $e_1 \dots e_n$ are the expressions. The factorial function using pattern matching looks as follows:

```
def factorial(n: Int): Int = n match {
   case 0 => 1
   case _ => n * factorial(n-1)
}
```

Chapter 3

Code Metrics

In this chapter we discuss what code metrics are (see section 3.1) and how to validate them (see section 3.2). We also provide an overview and the definition, as found in the literature, of the metrics researched in this study (see sections 3.3, 3.4 and 3.5).

3.1 Software Code Metrics

Software code metrics are used to measure specific aspects of the software system or process. The metrics are used as a way to gain insight in the process or system, by making estimations about certain properties. By giving these properties numbers, we can express and compare these properties better. The insights or estimations can help to make improvements or to make better estimations about the cost, quality, schedule, etc.

In this study, we focus on the metrics to measure properties about the system, specifically code based metrics. Commonly measured properties are the size, complexity, quality and maintainability. However, these measurements are often related to each other. Code based metrics are measured, as the name suggest, on the source code of the system. A code metric can be seen as a function that performs a measurement. These measurements can help to get insights about the complexity and quality of the code, and therefore the maintainability and fault-proneness of the code.

3.2 Metric Validation

To validate whether software metrics are useful measurements of code quality, the relation needs to be studied between the software metrics and the code quality. Briand et al. (1995) [9] presented an empirical validation method, were the relation between one or more internal attributes (e.g. cyclomatic complexity) and an external attribute (e.g. maintainability) is studied. However, code quality is a rather vague concept and there is no clear way to quantify the quality of a piece of code. Therefore, the fault-proneness of classes is often used to validate code metrics instead [9, 2, 3, 4, 5, 6]. The assumption is that classes that have poor code quality, are more likely to contain faults. However, this is not a perfect representation of the quality of the code.

A commonly used method to investigate the relation between the metrics and the fault-proneness of classes, is by constructing a prediction model [9, 2, 3, 4, 5, 6]. The assumption is that, if metrics can be used as fault predictors, than there is (most likely) a significant relation between the metrics and the fault-proneness. The ability of the metrics to predict faults, can also be used as indicator of the usefulness of the metrics. Logistic regression (see section 4.2) is often used to construct the fault prediction model.

3.3 General Code Metrics

3.3.1 Cyclomatic Complexity (CC)

Cyclomatic complexity (CC) is a metric to indicate the complexity of a section of code. CC is a graph-theoretic complexity metric and measures the number of independent paths through a section of code [12]. The metric uses the control flow diagram. The metric is defined as the number of paths that have one or more edges that has not been traversed before. The metric can be measured on function, pattern, class, module and program level.

3.3.2 Lines of Code (LOC)

Lines of code (LOC) is a metric to indicate the size of the source code. There are two common methods to measure LOC, the psychical and the logical method. The physical method counts the lines of text in the source code. The logical method counts the number of (executable) statements in the code. However, this definition depends on the programming language. The most common used method is the physical method [13]. The following three variants of the LOC metric are often used:

- **Lines of code (LOC).** The number of code lines, including the comment lines, in the source code, excluding blank lines. The assumption is that the larger the code, the more complex.
- **Source lines of code (SLOC).** The number of code lines in the source code, excluding blank and comment lines. SLOC is based on the same assumption as LOC, however, without the comments.
- **Comment lines of code (CLOC).** The number of comment lines in the source code, excluding blank and code lines. The assumption is that the complexer the code, the more comment lines are needed to explain the code.

3.3.3 Comment Density (CD)

The comment density (CD) is the ratio between the comments and the total lines of code. The equation for the CD is as follows:

$$CD = CLOC/LOC \tag{3.1}$$

The assumption is that if the comment density is to low, the code is under commented. This can lead to code comprehensibility problems. However, if the comment density is to high, the code is over commented. This can be a consequence of the code being to complex and therefore needing a relatively large number of comments.

3.4 Object-Oriented Code Metrics

3.4.1 Depth of Inheritance (DIT)

The depth of inheritance (DIT) is the maximum length of the node to the root in the inheritance tree [1]. The assumption is that the deeper a class is in the inheritance tree, the more definitions the class inherited from the ancestors, the more fault-prone the class is [2].

3.4.2 Number of Children (NOC)

NOC is defined as the number of direct descendants of a class [1]. The assumption is that the more children a class has, the more difficult it would be to modify the class, therefore the class would be more fault-prone [2].

3.4.3 Lack of Cohesion in Methods (LCOM)

LCOM measures the cohesion between the methods in a class. LCOM is defined as the number of pairs of methods without shared instance variables, minus the number of pairs of methods with shared instance variables [1, 2]. However, the value is often set to 0 for negative values.

The assumption is that classes with low cohesion are poorly designed (e.g. encapsulation of unrelated objects) [2].

The following variants of LCOM are commonly used:

- **LCOM** Negative values are set to 0
- **LCOMneg** Negative values are allowed

3.4.4 Coupling Between Objects (CBO)

Classes are considered coupled when a class uses methods or instance variables of another class. The Coupling between objects (CBO) of a class is the number of classes to which the class is coupled [1]. The assumption is that highly coupled classes are more fault-prone, due to inter-class activities [2, 5]. Highly coupled classes can also indicate weakness in module encapsulation [5].

3.4.5 Weighted Method Count (WMC)

WMC measures the complexity of a class. The WMC is measured by summing the complexities of the methods in the class [1]. If all methods are considered to be equally complex, then the WMC would be equal to the number of methods in each class [2]. Alternately, the cyclomatic complexity can be used instead of considering each method equally complex. The assumption is that a complex class tends to be more fault-prone than a less complex class.

3.4.6 Response for a Class (RFC)

The RFC of a class is the number of methods that can be called as response to a message received by the class, also known as the response set [1, 2]. The response set of a class is defined as the methods called by the methods of the class, plus the methods of the class itself. This leads to the following equations:

$$RFC = M \cup \{\forall x \in M | R_x\} \tag{3.2}$$

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

The assumption is that the larger the response set, the higher the complexity and the more fault prone the class is [1].

3.5 Functional Code Metrics

3.5.1 Pattern Size (PSIZ)

PSIZ measures the size of the pattern [7]. The PSIZ is defined as the number of abstract syntax tree (AST) nodes in the pattern. The assumption is that the pattern becomes more complex as the pattern sizes increases, and therefore more fault-prone. The PSIZ of a function is most likely correlated to the SLOC of the function, because both metrics measure some form of the size of the function.

3.5.2 Depth of Nesting (DON)

DON measures the maximum depth of nesting in a pattern [7]. The DON is defined as the maximum depth of the AST. The assumption is that the higher the depth of a pattern, the higher the complexity, and therefore the more fault-prone the pattern is.

3.5.3 Outdegree (OUTD)

OUTD of a function is defined as the number of functions called by the function [7]. The assumption is that the higher the OUTD, the more likely a function has to change because of a change in another function, which would increase fault-proneness.

The following variants of OUTD are commonly used:

- **OUTD** The number of function calls in the body of the function
- **OUTDdistinct** The number of distinct function calls in the body of the function

3.5.4 Number of Pattern Variables (NPVS)

NPVS is defined as the number of variables the pattern introduces into the scope [7]. The assumption is that the more variables a function introduces into the scope, the more variables a programmer must know to comprehend the code, therefore, the more fault-prone the code is.

Chapter 4

Briand's Validation Methodology

4.1 Validation

A common method to validate metrics, is to measure the performance of the metrics as fault predictors [2, 3, 4, 5, 6, 9]. This method investigates the relation between the metrics and the fault-proneness of classes, by coupling the information about faults in the class with the metric values of the class. From this data, a prediction model can be constructed.

The method collects the information about the faults in the system during the entire life-cycle of the system. For each class, that exist in the latest version of the code, the number of faults is counted that affected the class during the entire life-cycle of the class. After that, the metric values of the classes are calculated for the latest version of the code.

The algorithm used to collect and couple the information about the faults and the metric values of the classes is shown in Algorithm 1. This algorithm calculates for each class the metric values and the number of faults. It is important to notice that for getting the classes and calculating the metric values, the latest version of the code is analyzed. The result of this algorithm is a list of metric values of each class, combined with the number of faults in each class.

The data is then used to construct a prediction model using logistic regression. This model is then validated using cross-validation. The completeness and correctness are used to measure the performance of the model.

Data:
The latest version of the source code
The information about bugs
Result: List of metric values and bugs of each class

```
1  begin
2      S ← the source code of the project
3      B ← List of all the bugs in the project
4      C ← getClasses(S)
5      result ← []
6      foreach c ∈ C do
7          V ← getMetricValues(c) // Calculates metric values for the latest version of the class
8          F ← countBugsInClass(C, B) // Counts the bugs in the class
9          result += (F, V) // Adds a tuple of the number of bugs and the metric values of the class
10     end
11     return result
12 end
```

Algorithm 1: Algorithm to collect the information of the metric values and number of bugs of each class

4.2 Logistic regression

Regression is a commonly used method to describe the relation between a dependent (response or outcome) variable and one or more independent (predictor or explanatory) variables [14, 15, 16]. It is often the case that the dependent variable is categorical [14, 15]. In logistic, or logit, regression the dependent variable is dichotomous or binary (e.g. failed/success, present/absent or improved/not-improved). The logistic model estimates the possibility of a response based on one or more independent variables. In research, logistic regression is often used when the dependent variable is dichotomous [14, 15, 16].

The equation to predict the probability of an outcome of interest or event is as follows [14]:

$$\pi = \frac{e^{\alpha+\beta_1 X_1+\beta_2 X_2+\ldots+\beta_n X_n}}{1+e^{\alpha+\beta_1 X_1+\beta_2 X_2+\ldots+\beta_n X_n}} \tag{4.1}$$

Where π is the probability of the outcome of interest or event, X are the independent variables, α is the Y intercept and $\beta_1, \beta_2 \ldots \beta_n$ are the regression coefficients. The intercept (α) allows the prediction function to have an origin other than zero. α and $\beta_1, \beta_2 \ldots \beta_n$ are often estimated using the maximum likelihood method [16]. This method assigns values to the parameters to maximize the likelihood of reproducing the outcomes of the observed data.

Two methods can be used to construct a logistic regression model[14]:

Univariate Univariate regression uses only one independent variable and describes the relation between the independent variable and the dependent variable. Therefore, univariate regression is performed for each individual independent variable. This method is helpful in order to predict whether there is a relation between the independent and the dependent variable.

Multivariate Multivariate regression combines multiple independent variables to predict the outcome of the dependent variable. This method is used to determine how well the outcome of the dependent variable can be predicted using multiple independent variables.

4.3 Step-wise selection

When the number of independent variables (predictors) grows, the number of possible sub-sets grows exponentially. This makes it expensive to find the best combination of predictors for multivariate regression manually. An automated selection method, that aims to identify a sub-set with a high fit, can be used instead [14]. However, it is computational expensive to test all possible sub-sets when a large number of independent variables is used. Therefore, these methods use algorithms to determine which combinations to test.

Step-wise selection is a commonly used method. This methods adds or subtract variables from the set of independent variables based on some criterion. This criterion is often an indicator of the significance of the variable or the goodness of fit of the model (e.g. t-Test, Wald test, Akaike information criterion (AIC), R^2, etc). There are three types of step-wise selection:

- **Forward selection.** This type starts without any independent variables. Each step, the addition of each of the remaining independent variables is tested and the independent variable which improves the goodness of fit of the model the most, is added. This is repeated until there are no remaining variables or none of the remaining variables improves the goodness of fit of the model.
- **Backwards selection.** This type start with all the independent variables. Each step, the exclusion of each independent variable is tested and the independent variable which exclusion improved the goodness of fit of the model the most, is excluded. This is repeated until the exclusion of none of the independent variables improves the goodness of fit of the model.
- **Bidirectional selection.** This type uses a combination of the types described above, testing to either exclude or add an independent variable.

4.4 Model evaluation

To evaluate the performance, goodness of fit and significance of the model, measures can be used. These measures can also help to understand which independent variables are significant and contribute the most.

Coefficient The regression coefficient shows the relation between the independent variable and the dependent variable [14]. However, the range of the independent variable influences the coefficient. Therefore, the coefficients cannot be used to compare the strength of the relation between the independent and dependent variable, for independent variables with different ranges. However, the coefficient can be used to determine whether there is a positive or negative relation between the independent and dependent variable.

p-value The p-value is often used to determine the significance of statistical model. A statistical model is considered significant if the p-value is less than a certain threshold. In the regression model, the p-value can thus be used to determine the significance of a relation between an independent and dependent variable. In academic research, models with p-values less than 0.05 or 0.01 are often considered significant.

(Pseudo) R^2 R^2 is defined as the percentage of variation of the dependent variable that is explained by the regression model [17, 18]. The R^2 is used to asses the goodness of fit of a regression model. For logistic regression, there are different ways proposed for calculating the R^2. However, there is no consensus on which one should be used [18]. One of the methods that appears to be the most often reported and preferred in the field of statistics is McFadden method [18, 19].

Completeness The completeness of the model is defined as the number of events correctly classified as an event, divided by the total number of events in the data-set [4]. This measure shows the percentage of events the model would have classified as an event. The equation for the completeness is as follows (with the variables explained in table 4.1):

$$Completeness = \frac{E+}{(E-) + (E+)} \tag{4.2}$$

Correctness The correctness of the model is defines as the number of events classified as an events, divided by the total number of outputs classified as an event [4]. This measure shows the percentage of outcomes correctly classified as an event. A low correctness means that a high percentage of outcomes is incorrectly classified as an event. The equation for the correctness is as follows (with the variables explained in table 4.1):

$$Correctness = \frac{E+}{(N-) + (E+)} \tag{4.3}$$

Table 4.1: Actual and predicted event and non event variables

	Predicted Non Event	Predicted Event
Actual Non Event	N+	N-
Actaul Event	E-	E+

4.5 Model validation

Cross validation is a method commonly used to test and validate a prediction model [20, 21]. There are various methods for cross validation:

- **Holdout cross validation.** In holdout cross validation the data is randomly divided into two separate data-sets, often called train and test set. The sizes of train and test data-set can vary. However, the test set is usually smaller than the train set. The train set is used to train the model, the test set is used to test of the model. Multiple runs are often aggregated together to compensate for the randomness in the construction of the two data-sets.
- **k-fold cross validation.** In k-fold cross validation the data-set is randomly divided into k equally sized data-sets. Each of the k data-sets is used once as the validation set, and the remaining $k - 1$ data-sets as the training set. The results of each of these validations is folded together to obtain an averaged estimation.
- **Cross-system validation.** In cross-system validation the model is trained on the data-set obtained from one system, and tested on the data-set obtained from one or more other systems. This validation method helps to validate the generalizability of a model.

Chapter 5

Our Validation Methodology

In this Chapter we will present our own validation method for validating code metrics.

5.1 Critique of Briand's Methodology

The validation method of Briand et al. [9] calculates the metric values of the latest version of each class. However, bugs are counted over the entire duration of the project. In our opinion this results in a fundamental flaw in the method. The method assumes that the code of the latest version of a class, is similar to the version of the class when it contained bugs. If after a bug is detected in a class, the entire code of the class changes (e.g. due to a refactorings), then the code that is analyzed and is considered faulty, could be completely different than the actual faulty code. The longer the life cycle of a project, the more likely this will happen.

For smaller projects, with short life cylces, this is not a major problem, because it is less likely that entire classes are rewritten in a short time period. This likely applies to most of the metric validation research, because they mainly used student projects, with short life cycles. Gyimothy et al. [3], who performed their research on a large project (Morzilla), avoided this problem by only performing the analysis on a specific version of the code, and therefore artificially decreasing the life cycle.

5.2 New validation method

To solve this problem, we decided to measure the metric values on the moment the class contained a bug, instead of the latest version of the class. This ensures that the metric values that belong to faulty classes, are the metric values of the classes when it contained a fault. If for some reason a class is completely rewritten after a fault occurred, then the metric values that are measured in this method are still the metric values of the faulty class, instead of the rewritten class.

For this method, the algorithm used to collect and couple the information about the faults and the metric values of the classes needs to be modified. The improved method is shown in Algorithm 2. This algorithm uses the code repository of the project, instead of only the latest version of the code, as well as a list of all the bugs within the project.

First, it collects for each bug, the metric values of the classes that were affected by the bug (see line 6 to 13). This is done by iterating over each bug in the code. For each bug, the version of the code is selected which is affected by the bug. For this version, the classes are extracted which are affected by the bug. For each of the affected classes, the metric values are calculated and added to a list.

However, if a class is affected by multiple bugs, the class will also appear multiple times in the list. This can distort the results of the prediction model, because classes that appear multiple times are over represented. Therefore, we need to group the classes that appear multiple times in the data (see line 14). This grouping is done by taking the mean or median of the metrics for each class that appears multiple times. The grouped metric values of each class are combined with the number of faults in the class, by counting the number of occurrences of the class in the list.

```
Data:
The code repository (Github, Bitbucket, etc.)
The information about bugs
Result: List of metric values and bugs of each class
 1 begin
 2     S ← the code repository
 3     B ← List of all the bugs in the project
 4     interResult ← [] // The intermediate result
 5
 6     foreach b ∈ B do
 7         code ← getCode(b, S) // Get the version of the code were the bug appeared
 8         C ← getFaultyClasses(b, code) // Get the classes that contained the bug
 9         foreach c ∈ C do
10             V ← getMetricValues(c) // Calculates the metric values for the class
11             interResult += (c.name, V) // Adds a tuple of the class name and the metric values
12         end
13     end
       // Groups the duplicate classes
       // By taking the mean or median of each metric for each class
       // This results in a list of tuples of the number of bugs and the metric values for each class
       // [(number of bugs, metricValues)]
14     faultResult ← groupClasses(interResult)
15
16     latestCode ← getLatestCode(b, S) // Get the latest version of the code
17     C ← getClasses(latestCode)
18
       // Removes all the classes that don't exist in the latest version of the code
19     result ← filterExistingClasses(faultResult, C)
20
       // Add the metric values of the non-faulty classes to the list
21     foreach c ∈ C do
22         if x not in result then
23             V ← getMetricValues(c) // Calculates the metric values for the class
24
               // Adds a tuple of the number of bugs and the metric values of the class
               // The number of bugs is in this case always 0, because this are the non-faulty classes
25             result += (0, V)
26     end
27     return result
28 end
```

Algorithm 2: Improved algorithm to collect the information of the metric values and number of bugs of each class

It is also possible, that a class affected by a bug, does not even exist in the latest version of the code. Therefore, the data is filtered so that only faulty classes remain that exist in the latest version of the code (see line 19). This is necessary, because the distribution of the faulty and non-faulty classes in the data should represent the distribution of the actual faulty and non-faulty classes, to make sure the prediction model is as realistic as possible.

Finally, the metric values for the non-faulty classes should be added (see line 21 to 26). This is done by iterating over each class in the latest version of the code. For each class is checked whether the class already exist in the list of faulty classes. If the class does not exist in the list of faulty classes, then the metric values will be calculated for the class. Because the class is non-faulty, the number of faults in the class will be set to 0.

This algorithm will results in a list of metric values and number of bugs of each class, like the

previous algorithm. However, instead of the metric values of the latest version of the faulty classes, the metric values are now based on the metric values of the faulty classes when they contained a bug. The number of faults, the number of classes and the distribution of fault and non-faulty classes should be the same as the result of Algorithm 1. Therefore, the data of Algorithm 2 should be a good representation off the reality.

Chapter 6

Realization of the validation framework

6.1 Code Metrics

For this study, code metrics were selected for the different paradigms used by Scala (see chapter 3). In the following section, we will discuss which modifications were made and how they are implemented for Scala.

6.1.1 Cyclomatic Complexity (CC)

In Scala, CC can be calculated in a similar fashion as in *Java* or *C#*, by counting the number of occurrences of abstract syntax tree (AST) nodes of statements/expressions that create additional execution paths. This method avoids the need to construct a control flow diagram.

The following list contains the statements/expressions that create additional execution paths in Scala: if, else if, for, while, do while and case. The CC is calculated by adding 1 for each occurrence of the previous listed statements/expression. The start value of the CC is always 1, because there is always at least one execution path.

6.1.2 Lines of Code (LOC) and Comment Density (CD)

For Scala, we implemented the physical method for calculating the LOC, SLOC and CLOC. The blank lines should be removed for all three versions of the metric. The following regular expression (regex) can be used to detect blank lines:

$$\wedge \backslash s * \$ \tag{6.1}$$

Removing all the blank lines and counting the remainder of the lines, results in the LOC. To calculate the SLOC and CLOC variants, comments should be extracted from the code. Scala uses the following comment styles, similar to *Java*:

```
1 \\single line comment
2
3 \* multi line
4  * comment *\
```

The following regex can be used to detect comments:

$$\texttt{.*((\/*([\s\S]*?)*\/)|\/\/(.*)).*} \tag{6.2}$$

Removing the comment and blank lines from the code and counting the remaining line, results in the SLOC. Extracting and counting the comment lines, results in the CLOC.

When calculating the CLOC, empty leading and trailing comment lines are not counted in the result. This is to make the metric more precise. Empty leading and trailing comment lines are sometimes added as style choice, however, they add no additional information to the code.

It is important to notice that a line can be both a code and comment line simultaneously. This means that the LOC is not necessarily the sum of the SLOC and the CLOC.

The CD can be implemented by making use of the CLOC and LOC, as shown in Equation 3.1.

6.1.3 Depth of Inheritance (DIT)

In Scala object and traits exist besides classes. Objects cannot be extended, however, traits and classes can [8]. Traits are often used to reuse functionalities in multiple unrelated classes (e.g. utility functionalities). Therefore, the hypothesis is that, because traits often add non class specific functionalities, the depth of traits would be shallow, and therefore the exclusion of traits in the DIT metric, would have no influence or even improves the fault prediction capabilities of the DIT metric. To test this hypothesis, two versions of DIT are implemented:

- **DIT** Depth of inheritance, excluding traits
- **DITtraits** Depth of inheritance including traits

DIT can be calculated by creating an inheritance tree. The inheritance tree of a class or trait in Scala can be constructed using Algorithm 3.

```
Data: The class
Result: Inheritance tree of the class
1 Function IT(C ← class)
2     parents ← getParentClasses(C)
3     if metric == DITtraits then
4         parents += getParentTraits(C) // add this line to include traits (DITtraits)
5     result ← []
6     foreach p ∈ Parents do
7         result += (p, IT(p))
8     end
9     return result
```

Algorithm 3: Algorithm to calculate the DIT. Line 4 can be removed to exclude the traits from the result.

6.1.4 Number of Children (NOC)

In Scala both classes and traits can be extended and therefore have descendants, however, objects cannot [8]. This means that the NOC for objects is always 0. Furthermore, objects, classes and traits can all extend both classes and traits, and are therefore all counted as descendants.

NOC can be calculated by counting the number of extends of the class or trait in the source code. An extend of a class can be detected using the following regex:

```
1 (extends <name>) | (with <name>)
```

Where $< name >$ is the name of the class or trait. The NOC is the number of matches of the regex in the project source code.

6.1.5 Lack of Cohesion in Methods (LCOM)

LCOM can be calculated using Algorithm 4. This algorithm checks for every possible pair of methods in the class, object or trait, whether the intersection of the used variable sets of the methods, is an empty list or not. If the intersection of the two sets is empty, it means that the methods are not using

any similar instance variables. In Scala, both *var* and *val* are counted as instance variables.

```
Data: The class
Result: LCOM
1  Function LCOM(C ← class)
2      methods ← getMethods(C)
3      possiblePairs ⇐ methods.size(methods.size-1)/2 // The number of possible pairs
4
5      pairs ← []
6      foreach m1 ∈ methods do
7          foreach m2 ∈ methods do
8              if m1 != m2 and usedVars(m1) ∩ usedVars(m2) is not empty then
9                  pairs += (m1, m2)
10         end
11     end
12     return possiblePairs - 2 * pairs.size
```

Algorithm 4: Algorithm to calculate the LCOM

6.1.6 Coupling Between Objects (CBO)

In Scala objects, traits and classes can be coupled with each other. A pair is only coupled when one class, object or trait, uses a method or instance variable from the other class, object or trait.

Classes, objects and traits can use other methods and instance variables of other Classes, objects and traits as follows:

- Calls to inherit methods or variables from a class or trait;
- Calls to an instance of a class created by a new statement;
- Calls to an object;
- Calls to a class or trait that has been passed as parameter.

To measure if a pair is coupled, they both need to be checked if they make use of a method or instance variable from the other class, object or trait. The relation/link between the classes is symmetric, if class A is coupled to class B, then class B is coupled to class A.

6.1.7 Weighted Method Count (WMC)

In Scala, it is possible to write code directly in the class and can be seen as the constructor method of the class [8]. This code will most likely influence the complexity just like any other function would do. Therefore, it should be considered to treat this code like a function itself.

The following metrics are implemented to test the different variations and implementation decisions of the WMC metric for Scala:

- **WMC1** The number of methods in the class.
- **WMCcc** The sum of the CC of each method.
- **WMC1init** The number of methods in the class plus one for the code directly in the class.
- **WMCccinit** The sum of the CC of each method plus the CC of the code directly in the class.

6.1.8 Response for a Class (RFC)

The RFC has a strait forward implementations in Scala. Both the methods within the class and the methods that are called by methods within the class should be grouped in a list. All unique elements in this list should be counted to gain the RFC of the class. It is important to notice that inherited methods are not counted as method within the class. The same method can be used to calculated the RFC for objects or traits.

6.1.9 Pattern Size (PSIZ)

The functional metrics are originally developed for Haskell, a functional programming language. Functions in Haskell are often called patterns [7]. The following code snipped is example of the factorial function in Haskell:

```
factorial :: (Integral a) => a -> a
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

In Scala, the factorial function written using pattern matching looks as follows:

```
def factorial(n: Int): Int = n match {
   case 0 => 1
   case _ => n * factorial(n-1)
}
```

However, in Scala, not all functions make use of pattern matching. Therefore, not all functions are considered patterns. For example, the following code snippet shows the factorial function without the use of pattern matching:

```
def factorial(n: Int): Int = {
     if (n == 0)
         return 1
     else
         return n * factorial(n-1)
   }
```

However, functions in Scala without multiple cases can be considered a pattern with a single case. Therefore, the selected functional metrics, designed for Haskell, can be implemented on functional level in Scala.

We can use the AST of a function to calculate the PSIZ. We can define the PSIZ as the number of AST nodes in the AST of a function. However, because we look at the relation between the metrics and fault-proneness of classes, we need to transform the functional metrics from function level to class level. This can be done by grouping the results of all the functions of a class together. Therefore, we implemented the following variations of PSIZ on class level:

- **PSIZsum** The sum of all the pattern sizes of the functions in the class
- **PSIZavr** The average of all the pattern sizes of the functions in the class
- **PSIZmax** The maximum of all the pattern sizes of the functions in the class

6.1.10 Depth of Nesting (DON)

The DON has a similar implementation as the PSIZ metric. We can use the AST of the function and use the number of nodes as size indicator. However, instead of counting all AST nodes of a function, we need to count the number of nodes of the deepest path in the AST. DON can be calculated using the following algorithm:

Data: The AST of the function
Result: DON
1 **Function** *DON(A ← ASTnode)*
2 | *children* ← getChildren(A)
3 | *result* ← 0
4 | **foreach** $c \in children$ **do**
5 | | $result \leftarrow max(DON(c), result)$
6 | **end**
7 | **return** *result*

Algorithm 5: Algorithm to calculate the LCOM

Like the PSIZ, DON is calculated on function level. Therefore we implemented the following variations of DON on class level:

- **DONsum** The sum of all the DONs of the functions in the class
- **DONavr** The average of all the DONs of the functions in the class
- **DONmax** The maximum of all the DONs of the functions in the class

6.1.11 Outdegree (OUTD)

The OUTD can be calculated by counting all the function calls made by the function. The implementation of the OUTD is strait forward. We count all the FunctionCall nodes in the AST of the function. However, some function calls might call the same function. Therefore, we implemented an additional variant of the metric (OUTDdistinct) which only count the distinct function calls. This metric can be calculated by only counting unique functions calls.

Like the other functional metrics, we implemented various class level variations of the metrics:

- **sumOUTD** The sum of all the OUTDs of the functions in the class
- **avrOUTD** The average of all the OUTDs of the functions in the class
- **maxOUTD** The maximum of all the OUTDs of the functions in the class
- **sumOUTDdistinct** The sum of all the OUTDs of the functions in the class
- **sumOUTDdistinct** The average of all the OUTDs of the functions in the class
- **sumOUTDdistinct** The maximum of all the OUTDs of the functions in the class

6.1.12 Number of pattern variables (NPVS)

The NPVS counts the number of variables used in the pattern. This metric is originally designed for Haskell. As shown in section 6.1.9, Haskell patterns and Scala functions are similar, but not exact matches. Therefore, we need to use another definition for pattern variables. The definition we use for pattern variables is as follows: The variables created by patterns within the function plus the variables created by the function definition (parameters).

Like the other functional metrics, we implemented various class level variations of NPVS:

- **NPVSsum** The sum of all the NPVSs of the functions in the class
- **NPVSavr** The average of all the NPVSs of the functions in the class
- **NPVSmax** The maximum of all the NPVSs of the functions in the class

6.1.13 Inheritance

In Scala, both classes and traits can be inherited. Traits are unique features of Scala and can be inherited multiple times by an instance. The hypothesis is that the more traits are extended, the more complexer and difficult a class, object or trait is to comprehend. To investigate this hypothesis, we implemented an additional metric that measures the number of traits that are directly inherited.

Another hypothesis is that the directly inherited classes will be a less interesting metric, mainly because only one class can be inherited. This would lead to a binary metric value of 0 or 1. The assumption is that inheriting only one class has no measurable effect on the complexity or comprehensibility. To validate this hypothesis, the metric is implemented nevertheless.

6.1.14 Paradigm Score

In a functional and object-oriented language functions can be written in a functional or object-oriented (or imperative) programming style. Therefore, we implemented a metric to investigate whether the style of the functions of a class has influence on the fault-proneness of the class. However, there are no clear rules which state whether a function is written in a functional or object-oriented programming style. Furthermore, the border between the functional and object-oriented style is rather vague.

Determining whether a function is written in an object-oriented or functional programming style, is a study on its own. Therefore, we decided to use a naive approach. The paradigm score of a function is calculated by counting the number of functional elements used in the function, divided by the total number of elements (both functional and object-oriented) used in the function (see equation 6.3). This will always result in a value between zero and one. A value close to 1, means the function contains more functional elements, relative to the object-oriented elements. A value close to zero, means the function contains mainly object-oriented elements.

The elements counted as functional are the following: Folds, Maps, Filters, Counts, Exists, Finds, Recursion, Nesting and functions passed as arguments. For the object-oriented elements we counted the: For (-each) loops, (Do-) While loops and the number of side effects of the function.

$$paradigm\_score = \frac{functional\_elements}{functional\_elements + OO\_elements} \tag{6.3}$$

6.2 Code analysis

In the code analysis, the metric values of the classes are calculated. The metrics either need the abstract syntax tree (AST) of the code or the code as plain text. The Scala compiler can be used to parse the code to an abstract syntax tree. Besides parsing the code to an AST, the compiler also adds some basic context for each node. For instance, for a function call, the compiler adds the class of the called function to the node. The nodes also contain pointers to the location of the node in the unparsed code. These pointers can be used to extract the corresponding textual code for each node. However, the AST generated by the Scala compiler is rather complex and contains unnecessary information for the code analysis. Therefore, the AST is simplified, which contains only the information needed for the code analysis. In general, only the primary information (node type, position, children, name, parameters and owner) of the node remains.

For each metric, a tuple of the source code and AST of the class or function is passed as a parameter. However, some metrics are not only dependent of the class or function that is analyzed, but from the whole source-code. An example of such a metric is the Coupling between Objects (CBO) metric. Therefore, a list of all the project files is available for the metrics, as well as a runtime compiler, which can be used to compile a file to an AST. However, compiling files is time consuming. Therefore, strategies should be used to reduce the number of files which should be compiled for the metrics. The strategy used in our framework, is filtering using regular expressions before compiling.

Tree traversals are used to execute the metrics. The metrics can be calculated on class or function level, or both. When constructing a metric, this can be specified by extending the function- or class-metric trait, or both. Algorithm 6 shows the algorithm used for the traversal of the AST. For each class found in the AST, the class metrics are executed for the class. For each function found in the AST, the function metrics are executed for the function. However, in Scala, functions and classes can be nested. If an outer class contains an inner class, the metrics are calculated for the outer class and the inner class separately. This means that the code and AST nodes of the inner class are excluded when calculating the metrics for the outer class. The same applies to function, the inner functions are excluded when calculated the metrics for the outer function. The results of the metrics are placed in a result tree, that represent the structure of the files, classes and functions in the code.

Because the prediction model will be made on class level, we need to group the results of the function metrics. This is done by taking all the functions that are inside a class, excluding the functions that are in nested classes, and grouping the metric values of the functions. This grouping can be done by taking the mean, median or maximum of the metric values of the functions. All three the methods are implemented for the experiments. The last step is to flatten the result tree. This means that the tree, representing the class structure of the code, is flatten to a list of classes. This can be done by traversing over the result tree and adding each class, outer and inner classes, to a list.

```
Data:
The class
The affected lines
Result: Boolean value which indicates if the class is affected
 1 Function traverse(N: ASTnode, parent: Result)
 2 |   result ← null
 3 |   match N with
 4 |   |   case file⇒
 5 |   |   |   result ← new FileResult(N)
 6 |   |   end
 7 |   |   case object, class or trait⇒
 8 |   |   |   result ← new ObjectResult(runObjectMetrics(N))
 9 |   |   end
10 |   |   case function⇒
11 |   |   |   result ← new FunctionResult(runFunctionMetrics(N))
12 |   |   end
13 |   |   case Default⇒
14 |   |   |   pass
15 |   |   end
16 |   end
17 |   foreach c ∈ N.children do
18 |   |   traverse(c, result)
19 |   end
20 |   parent.add(result)
21 |   return parent
22 end
```

Algorithm 6: Tree traversal algorithm

6.3 Bug collection

During the bug collection, information about the bugs in the code is collected and coupled to the classes. For the validation, we use open-source projects which are available on Github. Therefore, the Github API can be used to collect the information about issues. By adding a label parameter in the API request, these issues can be easily filtered so only those issues remain which are labeled as a bug.

Next the commits that were made to close the issues, needs to be coupled to the issues. The issue closing pattern can be used, to find commits that closes an issue. The default issue pattern looks as follows [1] [2]:

```
1 (?i)(clos(e[sd]?|ing)|fix(e[sd]|ing)?|resolv(e[sd]?))
```

For detecting which issue is closed by a commit, we can use the reference to an issue in the commit message using the following (regular expression) pattern:

```
1 #\d*
```

If both the issue closing pattern and at least one reference to an issue is found within the commit message, the commit is considered an issue closing commit. The commit is then coupled to all the issues, labeled as a bug, which it references to. This results in a list of issues and the corresponding commits.

The patch data of the commits can then be used to couple the commits to the classes. The patch data contains the location of the lines of code the commit added or removed. The patch data of a commit looks as follows:

[1] https://help.github.com/articles/closing-issues-via-commit-messages/
[2] https://docs.gitlab.com/ee/user/project/issues/automatic_issue_closing.html

```
@@ -10,7 +10,8 @@ import ssca.validator._
*/
object Main {
  def main(args: Array[String]): Unit = {
-    val repoUser = "shadowsocks"
-    val repoName = "shadowsocks-android"
+    val repoUser = "gitbucket"
+    val repoName = "gitbucket"
+    val repoPath = "..\\tmp"

     val metrics = List(new Loc, new Complex, new DIT)
```

The pluses(+) and minuses(−) in front of the rows indicate whether a row is added or removed. When neither a plus or minus is in front of the row, it means that the row is unchanged. Furthermore, the part between the at-symbols(@@), indicate which source lines are shown by the patch data. The pattern is as follows:

```
@@ -a,b +x,y @@
```

Where tuple a, b points to the location before the commit, and x, y to the location after the commit. The first value of the tuple indicates the line number from which the patch starts, and the second value the number of lines.

With this data, we can detect which classes are affected by a bug. To do so, we use the version of the code before the commit. In the version before the commit, the bug still exists, while in the version after the commit (bug fix), the bug should be solved. We are interested in the classes which are faulty, so they should contain the bug, and therefore we need the version of the code which contained the bug.

The patch data should be converted into line numbers. The algorithm checks if the line is prefixed with a plus or minus. The plus lines are additions to the code. However, the added lines did not exist in the faulty version of the code. Therefore, we cannot simply use the line numbers of the patch data. An addition counts as if it altered the line before the addition. However, if a line is deleted in front of an addition, we can ignore the additions, because we already flagged the line as affected.

After we gathered the lines of the source code affected by the bug fix, we can check which classes are affected. We can do this by checking the intersect between the lines of code from the class and the affected lines of code (see Algorithm 7). The information about the start and end line of a class, can be extracted from the AST. If the intersect of these two sets is not empty, it means the class is affected. If a class has a nested (inner) class, we should exclude the lines of that class from the outer class. Because if a line of code affects the inner class, it should not be counted for the outer class. This can be done by subtracting the lines of code from the inner class, from the lines of code from the outer class.

It is important, when detecting which classes are altered by a commit, to use the version of the code before the commit. This is important to ensure the locations in the patch data correspond to the correct version of the code.

Data:
The class
The affected lines
Result: Boolean value which indicates if the class is affected.
1 $C \Leftarrow$ The class
2 $A \Leftarrow$ The affected lines
3 $L \Leftarrow getClassLineNumbers(C)$
4 **foreach** $c \in C.childClasses$ **do**
5 $\quad cL \Leftarrow getClassLineNumbers(c)$
6 $\quad L \leftarrow L \backslash cL$
7 **end**
8 **return** $L \cap A$ *is not empty*

Algorithm 7: Checks whether a class is affected by a bug fix

6.4 Data Collection - Briand's Method

To collect the data about the metric values and the bugs/faults, both the code analysis data and the bug collection data needs to be combined. Following the validation method as described by Briand et al. [9], the metric values of the classes of the latest version of the code needs to be coupled to the number of bugs in the class. This is done by calculating the metric values for each class for the latest version of the code and then adding the number of bugs found in the class using the result of the bug collection process. This will result in a list of classes, with the corresponding metric values of the classes and the number of bugs in the classes.

6.5 Data Collection - Our Method

For our validation method, modifications need to be made. Instead of calculating the metric values for each class for the latest version of the code, the metric values are calculated per bug. For each bug, the information about the classes that were modified to fix the bug are gathered. For these classes, the metric values are calculated, for the version of the code before the commit. This will result in a list of faulty-classes with the corresponding metric values. The classes are then grouped based on the class name, taking either the mean or median of each of the metrics. The number of bugs corresponding to the class is also added for each class. This will result in a list of faulty-classes with the corresponding metric values for the class and the number of bugs in the class. This list will be filtered, so only the classes remain that still exist in the latest version.

The next step is to add the non-faulty classes to the list, by calculating the metric values for each class, that does not exist in the faulty-classes list, for the latest version of the code. The non-faulty classes have all the number zero as number of bugs. The end result is a list of both faulty- and non-faulty classes, with the corresponding metric values and the number of bugs.

6.6 Data analysis

After the data is collected about the metric values and the number of bug in the classes, the data can be analyzed. Using the data a logistic regression model can be build. The number of bugs is the dependent variable and the metric values the independent variables. However, for logistic regression an binary or dichotomous data set is required, while the number of bugs can be 0 or more. Therefore, the number of bugs is set to 1 for values of 1 or more. Using the data and the prediction model, the metrics can be analyzed as fault-predictors.

Chapter 7

Empirical Validation Using Briand's Validation Methodology

7.1 Data

For the validation of the metrics, we investigate the relation between the metrics and the fault-proneness of classes. For the empirical validation we use open-source Scala projects. This allows us to validate the metrics for real world projects, instead of projects specially created for the experiments. However, for the validation, the source code and the information about the bug/faults of a project are needed. Fortunately, many open-source projects are available on Github.

Our framework [1] requires the use of the Github issue tracker, which allows us to detect faults in the code. It is important that the issues are labeled, namely because not all issues are necessarily faults. Therefore, only issues labeled as bugs are considered faults. Furthermore, our framework makes use of the default issue closing pattern to couple the issues to the commits. Therefore, it is required that the projects uses this default closing pattern to close commits.

We selected three open source Scala projects from the list of trending Scala projects on Github, which satisfy our requirements:

- **Gitbucket.** The Gitbucket project provides a Git platform powered by Scala. The project has at the time of writing around 3750 commits and around 200 issues labeled as faults. The data for this project was collected on 27 Juli 2017 (branch: master, commit hash: 53ae59271a3b5b832e3a7045e2b58205ca300d2a).

- **Akka.** The Akka project is a framework that helps building concurrent, reactive and distributed applications. The project has at the time of writing around 21100 commits and around 500 issues labeled as bugs. For the Akka project, a sub module, named Akka-Http, is used instead of the entire project. The reason behind this is because a large part of the project has not been actively changed. This can become a problem because a large part of the code is considered non-faulty, because it is not actively developed and therefore no bugs are reported or fixed. This generates noise in the analysis. Therefore, we only selected the most active module of the project, which also contained most of the faults. The data for this project was collected on 13 June 2017 (branch: master, commit hash: 398db4f40716cd91f86f8c07a57625af9ce2c696).

- **Shadowshock** The Shadowshock project is a framework to secure mobile Internet traffic for Android using proxies. The project has at the time of writing around 1900 commits and around 48 issues labeled as bugs. The data for this project was collected on 19 Juli 2017 (branch: release-2.4-http, commit hash: bb7727dee44364a6dff31ee99cad9ae3e6fe9830).

[1] https://github.com/ERLKDev/SSCA

7.2 Methodology

In this chapter, the metric validation method as described in the literature (see section 4.1) is used to collect the data. All three projects were analyzed separately. The analysis includes distributed statistics, distribution and logistic regression. The prediction model constructed using logistic regression is validated using k-fold validation and cross-system validation. The p-value is used to determine whether a metric has a significant relation with the fault-proneness of classes. We consider a relation with a p-value of 0.05 or lower as significant. The coefficient will be used to understand and explain the relation between the metric and the fault-proneness of classes. To measure the performance of the model, the correctness, completeness and $pseudoR^2$ are used (see section 4.4.)

- The correctness for the model is defined as the number of faulty classes correctly classified as an faulty class, divided by the number of total classes classified as faulty.
- The completeness for the model is defined as the number of faults in the classes classified as faulty, divided by the number of total faults in the system.

We will use univariate regression to study each metric separately. This will give insights in the performance, relation and significance of each metric. Multivariate regression will be used to study combinations of metrics. First of all, we studied the different metric suites separately, to investigate how well each metric suite performs for Scala. This will give insight in how well the metrics of each paradigm perform for Scala. Finally, we will combine the metrics from the different metric suites to see whether a combination of metrics from different paradigms improves the performance of a prediction model for Scala. Because we use in total 53 versions of the metrics, testing each possible combination is too time consuming. Therefore, we use forward step-wise selection as described in 4.3. To validate the prediction models, we will use k-fold cross validation (see section 4.5), with $k = 10$.

We will discuss the results of the Gitbucket project in more details. For the HTTP module of the Akka project and the Shadowshock project, only the highlights will be discussed. The remaining data of these projects can be found in Appendix A.

7.3 Gitbucket

7.3.1 Descriptive Statistics

The Gitbucket project contains 592 classes, of which 80 (208 faults) are faulty (13%). In table 7.1, 7.2 and 7.3 the descriptive statistics are shown for the general, object-oriented and functional metrics respectively. As seen in 7.1, the average class has about 25 lines of code, with the larges class having 849 lines of code. However, if we look at the 50% quartile, we see that over half of the classes have less then 5 lines of code and over 75% of the classes have less lines of code (17 loc) then the average (25.31 loc). This means the average is largely influenced by a small group of large classes.

We can also see that a large part of the classes, have no functions. Namely, because at the 50% quartile, the sum, average and maximum of the lines of code of the functions within the class is 0. These classes are most likely used for pattern matching, or simply to store data without the need to perform operations on the data. This can explain the low number of faulty classes, because it it less likely a class without any functions, is affected by a bug.

In table 7.2, we can see that DIT has a flat distribution, it is either 1 or 2, only two classes have a DIT of 3. This means their are no complex inheritance structures used. Classes either inherit the base class or another class that inherit the base class. This makes it less likely that there is a relation between the DIT and the fault-proneness of classes. The Inheritance metrics are similarly distributed as DIT. Therefore, the hypothesis is that the DIT and Inheritance metrics will not perform well as fault predictors.

Furthermore, only 11 classes have a NOC other than zero and 6 of those have a NOC of 2. This means most of the classes have no direct children. This was to be expected, because as the DIT suggest, most of the classes only inherit the base class and there are no complex inheritance structures. Therefore, most of the classes are not being inherited and have thus no direct children. Our hypothesis

is that there will be no relation between the NOC and fault-proneness of classes, and thus will not perform well as fault predictor.

In table 7.3, we can see that the metric values of the functional metrics is more variated. This makes it difficult to make perditions about the relation of the metrics and the fault-proneness.

	mean	std	min	25%	50%	75%	max
Object							
-CD	0.04	0.15	0.00	0.00	0.00	0.00	1.75
-CLOC	2.60	16.35	0.00	0.00	0.00	0.00	293.00
-LOC	25.30	72.56	0.00	1.00	5.00	17.00	854.00
-SLOC	22.85	60.66	0.00	1.00	5.00	16.75	728.00
Function Average							
-CD	0.00	0.02	0.00	0.00	0.00	0.00	0.42
-CLOC	0.07	0.40	0.00	0.00	0.00	0.00	5.00
-LOC	2.61	6.68	0.00	0.00	0.00	1.85	67.00
-SLOC	2.55	6.46	0.00	0.00	0.00	1.85	63.00
Function Sum							
-CD	0.02	0.10	0.00	0.00	0.00	0.00	1.25
-CLOC	0.32	1.74	0.00	0.00	0.00	0.00	19.00
-LOC	12.41	46.10	0.00	0.00	0.00	6.00	785.00
-SLOC	12.12	44.95	0.00	0.00	0.00	6.00	769.00
Function Maximum							
-CD	0.01	0.07	0.00	0.00	0.00	0.00	1.00
-CLOC	0.23	1.13	0.00	0.00	0.00	0.00	12.00
-LOC	5.54	13.31	0.00	0.00	0.00	4.00	114.00
-SLOC	5.36	12.74	0.00	0.00	0.00	4.00	112.00

Table 7.1: Descriptive statistics: General code metrics for the Gitbucket project using Briand's methodology

7.3.2 Univariate Regression

To study each metric in more depth, we used univariate logistic regression analysis. We will discuss the metrics of each metric suite separately. Table 7.4 presents the results of the object-oriented metric suite. The data suggest that the relation between all metrics are significant, except for NOC. This is as expected, because only 11 classes had a NOC other than zero. The relation of DIT is significant, however as expected, DIT does not perform well either. This is most likely because DIT is always either 1 or 2, with the exception of two classes.

The WMC (CC) inclusive init and RFC metrics have the best performance, followed by the LCOM, LCOM Negative and CBO. However, all metrics score a completeness below 40%. This suggests that there is a relation between the metrics and the fault-proneness, however the relation is not strong enough to make accurate predictions. When both faulty and non-faulty classes have the same metric values, it is impossible to classify all the metrics correctly. The more overlapping metric values the faulty and non-faulty classes have, the more classes will not be classified correctly.

Furthermore, the LCOM and LCOM Negative are highly correlated, they have both the same performance and coefficient. The same applies to WMC (normal) and WMC (normal) inclusive init. This is to be expected, because the metrics are quite similar. The difference between WMC (normal) and WMC (normal) inclusive init is that the WMC (normal) inclusive init is always 1 higher than the WMC (normal). The difference between the LCOM and LCOM Negative, is that the LCOM Negative is allowed to have negative values. However, if there are (almost) none negative values, the metrics are similar.

Table 7.5 presents the results of the functional metric suite. The data suggest that the relation between all metrics are significant. The performance of the metrics suggest that all metrics have a

	mean	std	min	25%	50%	75%	max
CBO	7.46	7.04	0.00	5.00	6.00	8.00	102.00
DIT	1.30	0.47	1.00	1.00	1.00	2.00	3.00
Inheritance	1.38	1.08	0.00	0.00	2.00	2.00	7.00
Class Inheritance	0.30	0.46	0.00	0.00	0.00	1.00	1.00
Trait Inheritance	1.08	0.98	0.00	0.00	1.00	2.00	7.00
LCOM	30.30	98.52	0.00	1.00	6.00	36.00	1429.00
LCOM Negative	30.28	98.53	-6.00	1.00	6.00	36.00	1429.00
NOC	0.09	0.97	0.00	0.00	0.00	0.00	19.00
RFC	21.72	33.21	0.00	8.00	11.00	27.00	508.00
WMC (CC)	10.44	13.03	0.00	3.00	5.00	16.00	139.00
WMC (CC) incl. Init	13.59	17.84	1.00	4.00	7.00	19.00	251.00
WMC (normal)	6.52	6.77	0.00	2.00	4.00	10.00	55.00
WMC (normal) incl. Init	7.52	6.77	1.00	3.00	5.00	11.00	56.00

Table 7.2: Descriptive statistics: Object-oriented code metrics for the Gitbucket project using Briand's methodology

relation to the fault-proneness to some degree, except the maximum variant of the DON metric. We found no clear explanation why this is the case for the maximum variant of the DON metric.

The sum variant of the Paradigm Score metric we introduced, performs relatively well (23.79% completeness and 83.33% correctness). This suggest there is a relation to some degree between the metric and the fault-proneness of classes. However, the results suggest that the average and maximum variants have no relation with the fault-proneness of classes. We found no explanation within the data why this is the case.

Furthermore, we see that overall, the sum variants of the functional metric perform best. This is most likely because the sum variation is influenced by the size of the class. There is a relation between the size of a class and the sum variants of the metric values. With larger classes, the sum variants of the metric values will most likely be higher. In table 7.6 we can see that the lines of code (Object LOC) is a relatively good fault-predictor. This confirms our hypothesis that the sum variants perform better than the average and maximum variants because they are correlated with the size. However, the average variants of the metrics do suggest that the metrics on their own, without being influenced by the size, have a relation with the fault-proneness in some degree.

Table 7.6 presents the results of the general code metrics. The data suggest that the relation between all metrics are significant, except the Object CD. This can be explained because there is a relatively low number of comment lines (Object CLOC) compared to the lines of code (Object LOC). Therefore, the CD will always be relatively low.

The Object SLOC and Object LOC perform quite well, however they are both correlated to each other. These metrics measure the size of the code. It is expected that the larger the class, the more change of faults within the class. Furthermore, the sum variants of the function lines of code (LOC) and function source lines of code (SLOC) perform well either. However, they are correlated with the object LOC and object SLOC.

Overall, the Object LOC and Object SLOC perform by far the best, with a score of around 65% completeness and around 82% correctness. The best performing metrics from the object-oriented and functional paradigm, score around 35% completeness and 85% correctness. The difference in scores are most likely the cause of a weaker relation between the metrics and the fault-proneness of the classes. Another factor that has influence on the results is the ratio between faulty and non-faulty classes. Logistic regression tries to generate a prediction model that fits the data best. However, when a large part of the classes have no faults, the prediction model would be biased to non-faulty classes, which would result in more classes being classified as non-faulty. Furthermore, metrics which are influenced by the size of the class, perform overall better than metrics who are not influenced by the size of the class.

	mean	std	min	25%	50%	75%	max
Function Average							
-CC	1.38	1.28	0.00	1.00	1.25	1.57	21.00
-DON	3.13	2.19	0.00	2.50	2.92	3.50	17.00
-Paradigm Score	0.15	0.25	0.00	0.00	0.00	0.17	1.00
-NPVS	1.95	3.15	0.00	0.50	0.80	2.05	34.00
-OutDegree	3.22	6.30	0.00	0.75	1.36	2.50	66.00
-OutDegree Distinct	1.76	2.56	0.00	0.69	1.00	1.67	29.00
-PatternSize	10.70	15.04	0.00	4.00	7.00	10.69	131.00
Function Sum							
-CC	10.41	13.04	0.00	3.00	5.00	16.00	139.00
-DON	22.27	29.01	0.00	8.00	13.00	33.00	393.00
-Paradigm Score	1.16	2.26	0.00	0.00	0.00	2.00	26.00
-NPVS	11.75	30.34	0.00	2.00	6.00	11.00	399.00
-OutDegree	20.43	50.74	0.00	4.00	6.00	21.00	814.00
-OutDegree Distinct	11.09	23.71	0.00	3.00	4.00	13.00	431.00
-PatternSize	72.06	139.90	0.00	15.00	36.00	86.00	2205.00
Function Maximum							
-CC	3.39	3.62	0.00	1.00	2.00	4.75	31.00
-DON	6.59	4.33	0.00	4.00	6.00	9.00	22.00
-Paradigm Score	0.47	0.50	0.00	0.00	0.00	1.00	1.00
-NPVS	5.14	7.22	0.00	1.00	3.00	7.00	87.00
-OutDegree	9.63	16.01	0.00	2.00	3.00	12.00	189.00
-OutDegree Distinct	4.48	5.01	0.00	2.00	3.00	6.00	48.00
-PatternSize	27.81	38.92	0.00	8.00	19.00	34.00	386.00

Table 7.3: Descriptive statistics: Functional code metrics for the Gitbucket project using Briand's methodology

7.3.3 Multivariate Regression

We use multivariate logistic regression analysis to study how well combinations of metrics perform as fault predictors. In table 7.7 we see the completeness and correctness scores of the multivariate regression analysis. The object-oriented and functional metric suites perform both above the 50% completeness and above 69% correctness. The data suggest that combining multiple metrics yield better predictions than using the metrics separately. The general code metrics have a completeness of about 67 % and a correctness of about 75%. However, combining the different metric suits yields the best results with a 75% completeness and a 80% correctness.

Furthermore, we see in table 7.8 that while the amount of faulty classes found is only 55%, the number of total faults in those classes is 75%. This means that the faulty classes with the most faults, are more easily found than classes with one or two faults.

In Appendix A the overview of the metrics selected by the step-wise selection algorithm for the multivariate regression analysis are shown. We see that the multivariate regression analysis uses functional, object-oriented and general code metrics to construct the prediction model. However, we cannot simply compare the influence of each metric used in the model. Each metric is measured on a different interval, therefore the coefficients are based on both the interval as well as the influence.

7.4 Shadowshock

The Shadowshock project contains 131 classes, of which 30 (73 faults) are faulty (22.9%). For the Shadowshock project the descriptive statistics (see figures A.9, A.10 and A.11) are similar to the Gitbucket project. The DIT, NOC and Inheritance metrics have little to no variation and are expected to have no relation with the fault-proneness, similarly to the Gitbucket project.

Metric	Constant	Coefficient	P-value	R^2	Completeness	Correctness
CBO	-2.8955	0.1179	0.0000	0.1083	18.45%	50.00%
DIT	-0.4908	-1.1357	0.0012	0.0291	0.00%	0.00%
Inheritance	-0.8666	-1.0110	0.0000	0.1201	0.00%	0.00%
ClassInheritance	-1.5919	-1.4101	0.0003	0.0400	0.00%	0.00%
TraitInheritance	-1.0651	-1.0395	0.0000	0.0901	0.00%	0.00%
LCOM	-2.0907	0.0062	0.0019	0.0459	17.48%	100.00%
LCOM Negative	-2.0907	0.0062	0.0019	0.0459	17.48%	100.00%
NOC	-1.8486	-12.2475	**0.9999**	0.0070	0.00%	0.00%
RFC	-2.7465	0.0329	0.0000	0.1414	34.47%	64.71%
WMC (CC)	-2.3353	0.0376	0.0000	0.0476	15.53%	100.00%
WMC (CC) incl. Init	-2.8694	0.0589	0.0000	0.1448	37.86%	86.67%
WMC (normal)	-2.1949	0.0447	0.0034	0.0179	6.80%	100.00%
WMC (normal) incl. Init	-2.2395	0.0447	0.0034	0.0179	6.80%	100.00%

Table 7.4: Univariate logistic regression: object-oriented metrics for the Gitbucket project using Briand's methodology

However, the sizes of the classes are differently distributed compared to the Gitbucket project. In the Shadowshock project, the classes have a overall higher number of lines of code (average LOC of 37.44). Additionally, the average is less influenced by a small group of large classes. 50% of the classes have a LOC of 17 or higher and 25% of the classes have a LOC of 40 % or higher. Furthermore, there are less classes without function compared to the Gitbucket project.

The results of the univariate regression analysis are similar as well compared to the Gitbucket project. As expected, the DIT, NOC and Inheritance metrics do not perform well, and suggest there is no relation between those metrics and the fault-proneness. Similarly, the sum variants of the functional metrics perform better compared to the average and maximum variants.

However, the LOC metrics for the Shadowshock project (31 % completeness and 72 % correctness) perform worse then for the Gitbucket project (65 % completeness and 83 % correctness). This might be due to the difference in LOC distribution. However, this can also be a consequence due to the flaw in the validation methodology discussed in section 5.1.

Overall the metrics perform slightly worse then for the Gitbucket project, however the best and worse performing metrics are similar, with the exception of LOC.

The results of the multivariate regression analysis are shown in table 7.9 and confirm our findings in the univariate regression analysis. The overall performance is lower for both the object-oriented and functional metric suits. However, the major difference is in the general metrics. Whereas the general metrics perform the best for the Gitbucket project, they perform the worse for the Shadowshock project. This difference is the consequence of the lower performance of the LOC metrics.

Combining all the metrics yield in a completeness of about 45% and a correctness of 65%. As expected, due to the lower performance of the metrics individual and combined, the overall performance is worse compared to the Gitbucket project.

7.5 Akka Http Module

The Akka http module contains 1054 classes, of which 113 (372 faults) are faulty (10.7%). The descriptive statistics are similar to the Gitbucket project. The DIT, NOC and Inheritance metrics have little to no variations and are expected to have no relation with the fault-proneness. Unlike the Shadowshock project, the Akka Http module has a similar lines of code distribution as the Gitbucket project.

In the univariate regression analysis there are major differences between the Akka Http module and the Gitbucket and Shadowshock project. The performance op the metrics is extremely low compared to the other two project. The metrics have a completeness rarely above 10%, with the highest at 18%.

Even though the metrics perform poorly compared to the other projects, we see similar best and

Metric	Constant	Coefficient	P-value	R^2	Completeness	Correctness
Function Average						
-CC	-2.4505	0.3841	0.0005	0.0379	2.43%	33.33%
-DON	-3.4514	0.4181	0.0000	0.1423	7.77%	50.00%
-Paradigm Score	-2.3219	2.1728	0.0000	0.0665	0.00%	0.00%
-NPVS	-2.5949	0.2794	0.0000	0.1330	21.84%	72.22%
-OutDegree	-2.5109	0.1442	0.0000	0.1480	14.56%	65.00%
-OutDegree Distinct	-2.6673	0.3406	0.0000	0.1408	14.08%	57.89%
-PatternSize	-2.7223	0.0611	0.0000	0.1554	16.50%	59.09%
Function Sum						
-CC	-2.3175	0.0363	0.0000	0.0451	15.53%	100.00%
-DON	-2.5366	0.0250	0.0000	0.0807	22.82%	90.00%
-Paradigm Score	-2.3787	0.3474	0.0000	0.0914	23.79%	83.33%
-NPVS	-2.6395	0.0572	0.0000	0.1730	31.55%	75.00%
-OutDegree	-2.7055	0.0330	0.0000	0.1860	34.47%	79.17%
-OutDegree Distinct	-2.6169	0.0553	0.0000	0.1480	29.61%	75.00%
-PatternSize	-2.6688	0.0090	0.0000	0.1441	29.61%	76.47%
Function Maximum						
-CC	-2.3219	0.1142	0.0001	0.0349	4.37%	66.67%
-DON	-2.7162	0.1148	0.0000	0.0402	0.00%	0.00%
-Paradigm Score	-2.3378	0.8478	0.0010	0.0250	0.00%	0.00%
-NPVS	-2.7729	0.1421	0.0000	0.1362	27.67%	68.42%
-OutDegree	-2.6578	0.0631	0.0000	0.1461	26.70%	75.00%
-OutDegree Distinct	-2.8737	0.1815	0.0000	0.1348	22.82%	66.67%
-PatternSize	-2.7062	0.0243	0.0000	0.1308	28.64%	81.25%

Table 7.5: Univariate logistic regression: functional metrics for the Gitbucket project using Briand's methodology

worst performing metrics. This suggest that the best performing metrics, perform the best across all three projects, and have most likely the strongest relation with the fault-proneness compared to the other metrics.

However, unlike the Shadowshock project, the LOC metrics tend to have the best performance in the Akka project. This suggest the distribution of the lines of code, might have influence of the performance of the metric. Namely, because for the Gitbucket and Akka project, the LOC metrics tend to perform best, while this is not the case for the Shadowshock project. Both the Gitbucket and Akka project have a similar lines of code distribution, while the Shadowshock project has a different lines of code distribution.

Table 7.10 presents the results of the multi-variate regression analysis of the HTTP module of the Akka project. For this project, the completeness is low. Our hypothesis is that this is due to the problem described in 5.1. Most likely, classes have (almost) completely changed since the the faults affected the classes. This is highly likely, because the project is larger and has a longer life cycle. Therefore, Briand's validation method would not work properly, because the metric values presenting faulty classes, does not correspond with the actual metric values of the classes when the faults affected the classes.

7.6 Threat to validity

The systems used for the analysis, contained by far more non-faulty classes than faulty classes. This will make the prediction model biased, there is far more change of a class being non-faulty then faulty. Therefore, only metrics with very strong relations, and almost no overlapping values will yield good results.

The method of counting the faults in classes, is based on the issue tracker. Faults were no issue was

Metric	Constant	Coefficient	P-value	R^2	Completeness	Correctness
Object						
-CD	-1.9125	0.9277	**0.1228**	0.0046	0.00%	0.00%
-CLOC	-2.1765	0.1151	0.0000	0.1281	40.78%	80.00%
-LOC	-3.0094	0.0364	0.0000	0.3395	64.08%	82.50%
-SLOC	-3.0725	0.0409	0.0000	0.3522	66.50%	82.93%
Function Average						
-CD	-1.9227	10.3907	0.0320	0.0135	0.49%	50.00%
-CLOC	-1.9606	0.8425	0.0010	0.0289	2.91%	50.00%
-LOC	-2.3683	0.1282	0.0000	0.1295	13.11%	55.00%
-SLOC	-2.3730	0.1326	0.0000	0.1303	9.22%	55.56%
Function Sum						
-CD	-2.0055	4.7490	0.0001	0.0517	10.68%	75.00%
-CLOC	-2.0694	0.3991	0.0000	0.0860	21.36%	66.67%
-LOC	-2.6140	0.0455	0.0000	0.2454	44.66%	70.59%
-SLOC	-2.6238	0.0473	0.0000	0.2465	44.66%	75.00%
Function Maximum						
-CD	-1.9992	6.4780	0.0002	0.0448	2.43%	57.14%
-CLOC	-2.0656	0.5129	0.0000	0.0771	18.45%	66.67%
-LOC	-2.5742	0.0809	0.0000	0.2010	32.52%	54.84%
-SLOC	-2.5883	0.0854	0.0000	0.2023	32.04%	59.26%

Table 7.6: Univariate logistic regression: general metrics for the Gitbucket project using Briand's methodology

	Completeness	Correctness
Object-oriented	51.44%	75.00%
Functional	57.69%	69.77%
General	67.79%	74.47%
All	75.00%	80.00%

Table 7.7: Multivariate regression: Completeness and correctness for the Gitbucket project using Briand's method

created for, will not be detected. Therefore, not all faults will be found, and some classes that were faulty, will be seen as non-faulty, which would distort the results of the prediction model. We also assumed that a commit that fixes an issue, solely makes changes to the classes that were needed to fix the issue. However, some issue fixing commits, might also make some other small changes unrelated to the fault.

Furthermore, this method calculates the metric values on the latest version of the code. As discussed in section 5.1, classes might have changed since the fault(s) appeared. Therefore, the metric values calculated for a faulty class, might not correspond with the actual metric values of the fault affected the class.

Finally, the framework used for the analysis and validation has not been validated for correctness by an external party.

	Not Faulty	Faulty
Not Faulty	486	11
Faulty	35 (52)	44 (156)

Table 7.8: Multivariate regression: Prediction table for the Gitbucket project using Briand's method

	Completeness	Correctness
Object-oriented	32.88%	56.25%
Functional	38.36%	56.25%
General	30.14%	81.82%
All	45.21%	65.00%

Table 7.9: Multivariate regression: Completeness and correctness for the Shadowshock project using Briand's method

	Completeness	Correctness
Object-oriented	16.30%	67.86%
Functional	18.00%	76.00%
General	17.76%	76.92%
All	14.11%	60.71%

Table 7.10: Multivariate regression: Completeness and correctness for the HTTP module of the Akka project using Briand's method

Chapter 8

Empirical Validation Using Our Validation Methodology

8.1 Methodology

In this chapter, we will discuss the results of our newly proposed metric validation method. Where the affected version of the classes are measured, instead of the latest version of the classes (see section 5.2). Our hypothesis is that the new method will perform better on larger projects with longer life-cycles than the previous method. We will use the same projects, measurements and model validation methods as used in chapter 7. However, the method of collecting the data for the prediction model differs.

We will discuss the results of the Akka Http module in more details. Mainly, because the differences between Briand's methodology and our's are the most relevant for the Akka Http module. For the Gitbucket and the Shadowshock project, only the highlights will be discussed. The remaining data of these projects can be found in Appendix A.

8.2 Akka Http Module

8.2.1 Descriptive Statistics

The number of faults and classes remains the same in our method compared to Briand's method. Therefore, we will not discuss them any further. In tables 8.1, 8.2 and 8.3 the descriptive statistics are presented for the Akka Http module using our methodology. While the exact numbers differ, the overall trend is the same compared to Briand's method. We will not discuss the descriptive statistics any further, mainly because they are relatively similar to the descriptive statistics using Briand's method.

8.2.2 Univariate Regression

Table 8.4, 8.5 and 8.6 presents the results of the univariate regression analysis. We can see major improvements for the Akka Http module using our methodology in comparison to Briand's methodology. Whereas Briand's methodology rarely had a completeness over 10% with the highest around 18%, our method yield results of above 40% completeness for the Akka Http module. This improvement is in line with our hypothesis, that our methodology will perform better for projects with longer life-cycles because it is less influenced by refactorings of the code.

Furthermore, the best and worse performing metrics are similar to the best and worse performing metrics using Briand's methodology. This suggest that our validation methodology yield similar relations between the metrics and the fault-proneness of class, however is able to validate metrics for projects with longer life-cycles.

	mean	std	min	25%	50%	75%	max
Object							
-CD	0.10	0.29	0.00	0.00	0.00	0.08	4.59
-CLOC	4.72	17.70	0.00	0.00	0.00	1.00	222.00
-LOC	28.24	72.23	0.00	1.00	4.00	14.00	526.00
-SLOC	23.96	59.80	0.00	1.00	4.00	12.00	468.00
Function Average							
-CD	0.01	0.03	0.00	0.00	0.00	0.00	0.33
-CLOC	0.25	1.39	0.00	0.00	0.00	0.00	14.50
-LOC	2.98	10.86	0.00	0.00	0.23	1.00	171.00
-SLOC	2.79	9.87	0.00	0.00	0.23	1.00	165.00
Function Sum							
-CD	0.04	0.15	0.00	0.00	0.00	0.00	2.00
-CLOC	0.73	3.16	0.00	0.00	0.00	0.00	36.00
-LOC	14.69	40.73	0.00	0.00	1.00	7.00	332.00
-SLOC	14.22	39.63	0.00	0.00	1.00	7.00	330.00
Function Maximum							
-CD	0.02	0.09	0.00	0.00	0.00	0.00	1.00
-CLOC	0.58	2.76	0.00	0.00	0.00	0.00	35.00
-LOC	6.11	16.64	0.00	0.00	1.00	4.00	171.00
-SLOC	5.70	14.96	0.00	0.00	1.00	4.00	165.00

Table 8.1: Descriptive statistics: General code metrics for the Akka Http module using our methodology

8.2.3 Multivariate logistic regression

Table 8.7 presents the completeness and correctness tables for the multivariate regression analysis for the Akka Http module using our validation methodology. The completeness and correctness of the object-oriented, functional and general metric suites is improved in comparison to the results of Briand's methodology. This means the prediction model constructed using our methodology yield better predictions.

Combining the different metric suites yield the best results, with a completeness of 51.95% and a correctness of 71%, compared to a completeness of 14.11% and a correctness of 60.71% using Briand's methodology.

Furthermore, the results suggest that most of the metrics have a relation with the fault-proneness of classes for the Akka Http module, while the results using Briand's methodology did not.

8.3 Gitbucket

In Appendix A, the results of the descriptive statistics and univariate regression analysis are presented for the Gitbucket project using our methodology. The exact numbers differ from the results of Briand's methodology, however they have a similar trend. Therefore, we will not discuss them any further.

The results of the multivariate regression analysis are quite similar as well (see table 8.8). However, the object-oriented metric suite performs slightly worse. We can not explain the exact reason for this. However, this can also be the consequence of the selection algorithm selecting different metrics.

Combining the metric suites performs slightly worse than the prediction model constructed by Briand's methodology. This is a consequence of the object-oriented metric suite performing worse using our method compared to Briand's method. However, the results suggest a similar relation between the metrics and the fault-proneness as the results of Briand's method, namely most metrics have a relation to some degree with the fault-proneness of classes.

	mean	std	min	25%	50%	75%	max
CBO	8.27	7.73	0.00	4.00	7.00	10.00	79.00
ClassInheritance	0.36	0.48	0.00	0.00	0.00	1.00	1.00
DIT	1.44	0.68	1.00	1.00	1.00	2.00	5.00
Inheritance	1.38	1.18	0.00	0.00	1.00	2.00	4.00
ClassInheritance	0.36	0.48	0.00	0.00	0.00	1.00	1.00
TraitInheritance	1.02	1.11	0.00	0.00	1.00	2.00	4.00
LCOM	63.22	345.46	0.00	1.00	6.00	36.00	6328.00
LCOM Negative	63.15	345.48	-20.00	1.00	6.00	36.00	6328.00
NOC	0.35	2.40	0.00	0.00	0.00	0.00	70.00
RFC	24.79	38.01	0.00	7.00	13.00	27.00	485.00
WMC (CC)	10.40	15.21	0.00	2.00	5.00	15.00	132.00
WMC (CC) incl. Init	14.39	19.19	1.00	5.00	8.00	17.00	207.00
WMC (normal)	7.47	10.78	0.00	2.00	4.00	10.00	113.00
WMC (normal) incl. Init	8.47	10.78	1.00	3.00	5.00	11.00	114.00

Table 8.2: Descriptive statistics: Object-oriented code metrics for the Akka Http module using our methodology

8.4 Shadowshock

In Appendix A, the results of the descriptive statistics and univariate regression analysis are presented for the Gitbucket project using our methodology. The exact numbers differ from the results of the descriptive statistics using Briand's methodology, however they have a similar trend. Therefore, we will not discuss them any further.

The results of the univariate regression analysis shows an overall improvement of the completeness for the metrics. Despite the project being relatively small, this does not exclude classes being refactored during the life time of the project. This could explain the improved performance of the metrics using our method, compared to the results of Briand's method.

Table 8.9 shows the results of the multivariate regression analysis. In line with the univariate regression analysis, we see a slight improvement in completeness compared to the results using Briand's methodology. Similar to the results using Briand's methodology, the results suggest there is a relation to some degree between most of the metrics and the fault-proneness of classes for the Shadowshock project.

8.5 Threat to validity

Similar to the previous method, the method to detect faults makes use of the issue tracker. Therefore, if no issue has been created for a fault, it will not me detected. Furthermore, if a commit fixes an issue, we assume all changes were necessary to fix the fault. This might not be always the case, and classes might be labeled falsely as faulty.

Finally, the framework used for the analysis and validation has not been validated for correctness by an external party. Furthermore, the validation method is newly proposed and therefore, it hasn't been validated or discussed by any external party.

	mean	std	min	25%	50%	75%	max
Function Average							
-CC	1.22	0.83	0.00	1.00	1.25	1.40	9.00
-DON	2.84	1.99	0.00	2.00	2.75	3.25	15.00
-Paradigm Score	0.15	0.22	0.00	0.00	0.06	0.20	1.00
-NPVS	1.75	4.65	0.00	0.25	0.58	1.00	51.00
-OutDegree	3.31	10.28	0.00	0.50	1.00	1.60	176.00
-OutDegree Distinct	1.65	3.23	0.00	0.50	1.00	1.33	45.00
-PatternSize	10.51	28.17	0.00	2.50	4.67	6.43	509.00
Function Sum							
-CC	10.32	15.17	0.00	2.00	5.00	15.00	132.00
-DON	24.12	45.01	0.00	6.00	13.00	31.00	650.00
-Paradigm Score	1.32	2.58	0.00	0.00	0.65	2.00	27.76
-NPVS	10.11	24.92	0.00	2.00	4.00	7.00	266.00
-OutDegree	20.19	52.55	0.00	2.00	4.00	15.00	654.00
-OutDegree Distinct	12.03	25.85	0.00	2.00	4.00	12.00	286.00
-PatternSize	66.00	141.87	0.00	9.00	22.00	59.00	1486.00
Function Maximum							
-CC	2.42	2.38	0.00	1.00	2.00	3.00	23.00
-DON	5.27	3.36	0.00	3.00	6.00	7.00	24.00
-Paradigm Score	0.50	0.50	0.00	0.00	0.65	1.00	1.00
-NPVS	3.96	7.65	0.00	1.00	2.00	3.00	66.00
-OutDegree	7.74	16.13	0.00	1.00	3.00	6.00	176.00
-OutDegree Distinct	3.86	5.08	0.00	1.00	2.00	4.00	45.00
-PatternSize	23.37	43.53	0.00	5.00	11.00	20.00	509.00

Table 8.3: Descriptive statistics: Functional code metrics for the Akka Http module using our methodology

Metric	Constant	Coefficient	P-value	R^2	Completeness	Correctness
CBO	-3.6035	0.1244	0.0000	0.1236	15.58%	64.29%
DIT	-0.7353	-1.3548	0.0001	0.0483	0.00%	0.00%
Inheritance	-1.6614	-0.7317	0.0000	0.0761	0.00%	0.00%
ClassInheritance	-2.0572	-1.7387	0.0000	0.0556	0.00%	0.00%
TraitInheritance	-1.9653	-0.5865	0.0001	0.0402	0.00%	0.00%
LCOM	-2.6802	0.0038	0.0000	0.0753	12.34%	77.78%
LCOM Negative	-2.6799	0.0038	0.0000	0.0753	12.34%	77.78%
NOC	-2.4126	-0.2707	**0.2073**	0.0055	0.00%	0.00%
RFC	-3.3970	0.0330	0.0000	0.1708	33.12%	57.14%
WMC (CC)	-3.0957	0.0521	0.0000	0.0930	12.99%	87.50%
WMC (CC) incl. Init	-3.6049	0.0697	0.0000	0.1844	25.97%	65.00%
WMC (normal)	-3.0794	0.0716	0.0000	0.0794	12.34%	77.78%
WMC (normal) incl. Init	-3.1510	0.0716	0.0000	0.0794	12.34%	77.78%

Table 8.4: Univariate logistic regression: Object-oriented metrics for the Akka Http module using our methodology

Metric	Constant	Coefficient	P-value	R^2	Completeness	Correctness
Function Average						
-CC	-3.1163	0.5096	0.0003	0.0261	3.25%	50.00%
-DON	-3.6096	0.3603	0.0000	0.0764	3.90%	50.00%
-Paradigm Score	-2.8455	2.1580	0.0000	0.0399	0.00%	0.00%
-NPVS	-2.6999	0.1220	0.0000	0.0573	14.94%	50.00%
-OutDegree	-2.6348	0.0485	0.0000	0.0408	5.19%	50.00%
-OutDegree Distinct	-2.7606	0.1617	0.0000	0.0526	3.25%	40.00%
-PatternSize	-2.6480	0.0167	0.0001	0.0365	1.95%	25.00%
Function Sum						
-CC	-3.0818	0.0513	0.0000	0.0907	12.99%	87.50%
-DON	-3.1308	0.0244	0.0000	0.1184	14.94%	83.33%
-Paradigm Score	-2.8934	0.2688	0.0000	0.0846	14.94%	75.00%
-NPVS	-3.0596	0.0565	0.0000	0.1491	27.27%	73.33%
-OutDegree	-2.9763	0.0227	0.0000	0.1408	21.43%	73.33%
-OutDegree Distinct	-3.1297	0.0497	0.0000	0.1518	22.08%	80.00%
-PatternSize	-3.0731	0.0082	0.0000	0.1442	21.43%	78.57%
Function Maximum						
-CC	-3.0044	0.1984	0.0000	0.0443	0.00%	0.00%
-DON	-3.7065	0.2090	0.0000	0.0662	0.00%	0.00%
-Paradigm Score	-2.8862	0.7477	0.0048	0.0166	0.00%	0.00%
-NPVS	-2.8973	0.0938	0.0000	0.0936	17.53%	46.15%
-OutDegree	-2.7931	0.0354	0.0000	0.0698	13.64%	36.36%
-OutDegree Distinct	-3.0336	0.1241	0.0000	0.0857	9.09%	41.67%
-PatternSize	-2.8373	0.0138	0.0000	0.0709	13.64%	44.44%

Table 8.5: Univariate logistic regression: Functional metrics for the Akka Http module using our methodology

Metric	Constant	Coefficient	P-value	R^2	Completeness	Correctness
Object						
-CD	-2.6148	1.2226	0.0001	0.0346	0.65%	25.00%
-CLOC	-2.9276	0.1101	0.0000	0.1890	29.22%	73.68%
-LOC	-3.2123	0.0254	0.0000	0.2320	45.45%	76.00%
-SLOC	-3.1372	0.0265	0.0000	0.2068	44.16%	70.83%
Function Average						
-CD	-2.5941	14.7796	0.0000	0.0412	1.95%	16.67%
-CLOC	-2.5586	0.3260	0.0000	0.0400	13.64%	57.14%
-LOC	-2.6030	0.0428	0.0000	0.0414	5.19%	50.00%
-SLOC	-2.6044	0.0466	0.0000	0.0398	8.44%	57.14%
Function Sum						
-CD	-2.6962	5.0196	0.0000	0.0962	13.64%	61.54%
-CLOC	-2.6397	0.2347	0.0000	0.0731	14.29%	71.43%
-LOC	-2.9569	0.0300	0.0000	0.1662	32.47%	72.22%
-SLOC	-2.9565	0.0313	0.0000	0.1640	32.47%	76.47%
Function Maximum						
-CD	-2.6795	6.5691	0.0000	0.0812	5.19%	44.44%
-CLOC	-2.5953	0.2126	0.0000	0.0532	12.99%	66.67%
-LOC	-2.7454	0.0390	0.0000	0.0786	11.69%	33.33%
-SLOC	-2.7501	0.0425	0.0000	0.0769	11.69%	37.50%

Table 8.6: Univariate logistic regression: General metrics for the Akka Http module using our methodology

	Completeness	Correctness
Object-oriented	40.26%	65.52%
Functional	46.10%	75.00%
General	40.91%	67.86%
All	51.95%	71.05%

Table 8.7: Multivariate regression: Completeness and correctness for the HTTP module of the Akka project using the our validation method

	Completeness	Correctness
Object-oriented	29.47%	63.64%
Functional	54.59%	66.67%
General	62.32%	64.44%
All	62.32%	72.00%

Table 8.8: Multivariate regression: Completeness and correctness for the Gitbucket project using our validation method

	Completeness	Correctness
Object-oriented	36.36%	53.85%
Functional	43.64%	57.14%
General	52.73%	61.11%
All	63.64%	59.09%

Table 8.9: Multivariate regression: Completeness and correctness for the Shadowshock project using our validation method

Chapter 9

Related work

Basil et al. [2], Gymthy et al. [3], Briand et al. (2002) [4], Tang et al. [5] and Aggarwal et al. [6] all researched the object-oriented metric suite introduced by Chidamber et al. [1]. They all did an empirical validation of the metric suite. However, the projects, project sizes and project types differed from each other. All of the projects they examined were written in C++ or *Java*, which are object-oriented languages. They all used the same empirical validation method presented by Briand et al. (1995) [9]. Logistic regression was used to construct the prediction model. However, in some of the research additional methods were used to construct prediction models. In some of the research they also researched additional metrics, beside the metric suit presented by Chidamber et al. [1].

Basil et al. [2] ran a controlled study for the duration of four months. They used the code of 8 teams. The teams consisted out of students from a graduate level class, who were not required to have any experience in Object-oriented methods. However, all students had experience with C or C++. Each team constructed a medium-sized management information system using the waterfall methodology. Forms were used to report bugs in the system, as well as a testing phase. For the analysis, the information about bugs from the entire life cycle, both the development and test phase, of the project were used, as well as the latest version of the code. The results showed that the prediction model based on the metrics had a 88% completeness and a 60% correctness in predicting whether a class is faulty or not. When weighting the classes based on the number of faults they contain, they scored a 93% completeness and a 92% correctness. They also found that besides LCOM, all metrics had a significant relation with the fault-proneness of classes.

Briand et al. (2002) [4] studied the effectiveness of a prediction model, based on the metric suite of Chidamber et al. [1] and some additional metrics, across multiple systems. They used two systems, Xpose and Jwitre, written in *Java*, with 144 and 68 classes respectively. The data about faults found (and reported) by customers was used as fault data. They only used the faults after certain versions, because since those version the classes remained about the same. Linear regression and multivariate adaptive regression splines (MARS) were used to construct the prediction models, instead of logistic regression. To validate the metrics for one system, they used k-fold cross validation. The linear model had a completeness of 62% and a correctness of 73.4%. The completeness is based on predicting faults, while the correctness is based on predicting faulty classes. They used cross system validation to measure the accuracy of the prediction model of one system applied to another system. They found that a good trade-off between completeness and correctness can only be found with a low cutoff value (0.22 instead of 0.5). It is also important to notice, that beside the object-oriented metric suite of Chidamber et al. [1], the results are based on the additional metrics as well.

Gymthy et al. [3] researched the validity of the metric suite on a real sized system. They used a real sized open source project, consisting out of 3677 classes, called Mozilla. Mozilla is written in C++. They used Bugzilla, a bug tracker for Mozilla, to collect the information about bugs in the code. The bugs were coupled with the classes by looking which classes the bug-fix affected. For each affected version, they checked which classes were affected by the bug fix, by searching for classes which interval in the code overlaps with the interval of the bug fix. If a class was affected by a bug fix, it was assumed that the class contained a fault. Besides logistic and linear regression, they used a decision tree and a neural network to construct a prediction model. The analysis was preformed on multiple

versions of Mozilla separately, by using the source code of the version and the bugs belonging to the specific version. The assumption was that the classes would remain about the same within a version. For the multivariate regression they only used version 1.0. The logistic regression model scored a 83.76% completeness and a 55.94% correctness. The completeness was based on the faults and the correctness was based on the faulty and non-faulty classes. They concluded that all metrics, except the number of children (NOC), were significant.

Tang et al. [5] studied, besides the metrics of Chidamber et al. [1], their own newly presented metrics. For their study they used three subsystems of a large system. The subsystems were of small size, containing between 20 and 45 classes. The three systems were written in C++. The fault reports from the last three years wear used to collect the information about faults. In their study, they distinguished three types of faults: object-oriented faults, object management faults and traditional faults. They researched how well the metrics performed on each type of fault. They concluded that WMC can be used as a good indicator of faulty classes and that RFC is a good indicator of object-oriented faults. Furthermore, they concluded that their newly presented metrics are an useful indicator of object-oriented fault-prone classes. However, they didn't study the completeness and correctness of their model.

Aggarwal et al. [6] studied, besides the metrics of Chidamber et al. [1], the coupling metrics introduced by Briand et al. For their research, they used twelve mid sized code projects written by twelve groups of four students each. The projects were written in java and the projects combined had a total number 136 classes, consisting out of 39 thousand lines of code (KLOC), from which 85 were system classes and 51 were standard library classes. They used a logistic regression model to predict whether classes were faulty or not. Their results show their model has a 86.5% completeness (or sensitivity) and above 90% correctness (or specificity).

In most of the metric validation studies, they scored a completeness of 60+% and a correctness of 55+%. However, most of the studies used additional metrics, besides the metrics of Chidamber et al. [1]. For the functional metric presented by Ryder et al. [7], no additional metric validation was found at the time of writing. Most of the studies also used small to medium sized project, with the exception of Gymthy et al. [3].

Our results for the Gitbucket project using Briand's methodology are within the ranges of similar studies. The results of the Akka Http module and the Shadowshock project, using Briand's methodology, are not within the ranges of similar studies. When using our methodology, the results are closer or even within the ranges of similar research. However, because of the usage of different metrics, programming language, projects and methodology, the comparison is not accurate or completely fair.

Chapter 10

Conclusion

In this research, we investigated the relation between code metrics and the fault-proneness of classes for the multi-paradigm programming language Scala. We formulated object-oriented, functional and general code metrics for the Scala programming language. The validation method described by Briand et al. [9] is used to investigate the relation between the code metrics and the fault-proneness of classes.

For one of the investigated projects using Briand's methodology, the results suggested that there is a relation between most of the code metrics and the fault-proneness of classes. The results also suggest that a combination of functional, object-oriented and general code metrics yield the best results for Scala. However, for the Akka Http module and the Shadowshock project, the results suggested that the relation is weak or non existing.

However, during our research we discovered a flaw in the methodology of Briand. Namely, the latest version of the code is used to calculate the metric values, while the faults occur in various versions of the code. This method does not take into account that classes affected by faults, can change (almost) completely between the faulty version and the latest version of the code. This has little influence on projects with short life cycles. Namely, because the probability of classes being refactored in short projects is low. However, this does influence projects with long life cycles. This means Briand's validation methodology will not yield accurate results for projects with longer life-cycles, such as industrial projects. This reduces the industrial application of the methodology.

Therefore, we presented our own validation methodology based on Briand's methodology. Our methodology measures the metric values of the faulty classes the moment the class is affected by a fault. Therefore, our methodology is not affected by refactorings of classes. Our results show that our method yield better prediction results (up to more than a two-fold increase in completeness), compared to Briand's methodology, for projects with longer life cycles and similar results for projects which were not affected by refactorings. However, these results have not been verified or replicated by external research. Therefore, additional research needs to be conducted, using different project and programming languages, to confirm our findings. Furthermore, the results of our validation methodology, suggest there is a relation between most of the code metrics, except DIT, Inheritance, NOC and CD, and the fault-proneness of classes for all three investigated projects. This leads to the following answers to our research questions:

RQ1: Can metrics from multiple paradigms be combined to give a coherent result? To combine the metrics of multiple paradigms, the results of the metrics should be mapped to the same level, in our case the class level, namely because we measure the fault-proneness of classes. The object-oriented metrics are measured on class level, were as the functional metrics are measured on function level. The function level results can be mapped to class level by a grouping strategy. Our results suggest that summing the results has overall the best performance, but is partially influenced by the size of the class.

RQ2: How can metrics be modified/customized so they can be implemented for the Scala programming language? To implement the object-oriented metrics for Scala, we need to take into account the addition of traits in Scala. Our data suggest that trait can be treated similar

to classes, namely because they are technical almost similar, with the exception that traits can be inherited multiple times.

To implement the functional metrics for Scala, we need to take into account that the functional metrics were designed for Haskell and thus specifically for patterns. However, in Scala not all functions make use of patterns. Therefore, functions which do not make use of patterns, can be considered as single case patterns.

Furthermore, some small modifications need to made to individual metrics to accommodate for smaller differences.

RQ3: Can we validate that there is a relation between the metrics and the fault-proneness of classes for the Scala programming language? We investigated the relation between the metrics and fault-proneness using Briand's validation methodology and our own validation methodology. The results suggest that there is a relation between the metrics, except for DIT, Inheritance, CD and NOC, and the fault-proneness of classes. However, the results of Briand's method only suggest this relation for two out of the three project used, while the results of our method suggest the relation for all three projects.

RQ: How can code quality be evaluated, using code metrics, for the multi-paradigm programming language Scala? Our results suggest that code metrics can be used to evaluate an aspect of code quality, namely fault-proneness, for the Scala programming language. Combining the metrics of the object-oriented and functional paradigm, including some general code metrics, yield the best results for the Scala programming language.

Bibliography

[1] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[2] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Transactions on Software Engineering*, vol. 22, no. 10, pp. 751–761, 1996.

[3] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 897–910, 2005.

[4] L. C. Briand, W. L. Melo, and J. Wust, "Assessing the applicability of fault-proneness models across object-oriented software projects," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 706–720, 2002.

[5] M.-H. Tang, M.-H. Kao, and M.-H. Chen, "An empirical study on object-oriented metrics," in *Software Metrics Symposium, 1999. Proceedings. Sixth International*, pp. 242–249, IEEE, 1999.

[6] K. Aggarwal, Y. Singh, A. Kaur, and R. Malhotra, "Investigating effect of design metrics on fault proneness in object-oriented systems.," *Journal of Object Technology*, vol. 6, no. 10, pp. 127–141, 2007.

[7] C. Ryder and S. J. Thompson, "Software metrics: measuring Haskell.," in *Trends in Functional Programming*, pp. 31–46, 2005.

[8] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala*. Artima Inc, 2008.

[9] L. Briand, K. El Emam, and S. Morasca, "Theoretical and empirical validation of software product measures," *International Software Engineering Research Network, Technical Report ISERN-95-03*, 1995.

[10] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger, "An overview of the Scala programming language," tech. rep., 2004.

[11] P. Hudak, "Conception, evolution, and application of functional programming languages," *ACM Computing Surveys (CSUR)*, vol. 21, no. 3, pp. 359–411, 1989.

[12] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, no. 4, pp. 308–320, 1976.

[13] V. Nguyen, S. Deeds-Rubin, T. Tan, and B. Boehm, "A SLOC counting standard," in *COCOMO II Forum*, vol. 2007, 2007.

[14] D. W. Hosmer Jr, S. Lemeshow, and R. X. Sturdivant, *Applied Logistic Regression*, vol. 398. John Wiley & Sons, 2013.

[15] C. M. Dayton, "Logistic regression analysis," *Stat*, pp. 474–574, 1992.

[16] C.-Y. J. Peng, K. L. Lee, and G. M. Ingersoll, "An introduction to logistic regression analysis and reporting," *The Journal of Educational Research*, vol. 96, no. 1, pp. 3–14, 2002.

[17] J. Cohen, P. Cohen, S. G. West, and L. S. Aiken, *Applied multiple regression/correlation analysis for the behavioral sciences*. Routledge, 2013.

[18] P. D. Allison, "Measures of fit for logistic regression," in *SAS Global Forum, Washington, DC*, 2014.

[19] D. McFadden, "The measurement of urban travel demand," *Journal of Public Economics*, vol. 3, no. 4, pp. 303–328, 1974.

[20] M. Stone, "Cross-validatory choice and assessment of statistical predictions," *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 111–147, 1974.

[21] R. Kohavi *et al.*, "A study of cross-validation and bootstrap for accuracy estimation and model selection," in *Ijcai*, vol. 14, pp. 1137–1145, Stanford, CA, 1995.

[22] E. J. Weyuker, "Evaluating software complexity measures," *IEEE Transactions on Software Engineering*, vol. 14, no. 9, pp. 1357–1365, 1988.

[23] L. C. Briand, J. W. Daly, and J. K. Wust, "A unified framework for coupling measurement in object-oriented systems," *IEEE Transactions on Software Engineering*, vol. 25, no. 1, pp. 91–121, 1999.

Appendix A

Results

A.1 Gitbucket

Metric	Coefficient	P-value
Intercept	-22.5759	0.9963
objectSLOC	0.8650	0.0813
CBO	-0.1225	**0.0046**
functionAvrOutDegreeDistinct	0.1872	**0.0008**
objectLOC	-0.8116	0.1020
objectCLOC	0.7644	0.1217
LCOMneg	0.0076	0.0697
objectCD	-6.5998	0.1974
functionMaxFunctionCD	16.5844	**0.0156**
functionAvrFunctionCD	-43.9271	**0.0188**
functionSumOutDegreeDistinct	-0.0232	0.2139
ClassInheritance	-21.0024	0.9965
DIT	19.9132	0.9967

Table A.1: Multivariate regression: variables for the Gitbucket project using Briand's methodology

	count	mean	std	min	25%	50%	75%	max
CBO	721.00	9.29	10.96	0.00	5.00	6.00	10.00	98.00
ClassInheritance	721.00	0.25	0.43	0.00	0.00	0.00	0.00	1.00
DIT	721.00	1.25	0.44	1.00	1.00	1.00	1.00	3.00
Inheritance	721.00	1.16	1.11	0.00	0.00	1.00	2.00	7.00
LCOM	721.00	37.44	95.65	0.00	1.00	6.00	37.00	903.00
LCOMneg	721.00	37.42	95.66	-6.00	1.00	6.00	37.00	903.00
NOC	721.00	0.08	0.88	0.00	0.00	0.00	0.00	19.00
RFC	721.00	32.24	53.04	0.00	9.00	14.00	33.00	489.00
TraitInheritance	721.00	0.91	0.99	0.00	0.00	1.00	2.00	7.00
WMCcc	721.00	12.03	15.53	0.00	3.00	6.00	18.00	130.00
WMCccInit	721.00	18.70	25.82	1.00	6.00	10.00	23.00	241.00
WMCnormal	721.00	7.00	7.37	0.00	2.00	4.00	10.00	51.00
WMCnormalInit	721.00	8.00	7.37	1.00	3.00	5.00	11.00	52.00

Table A.2: Descriptive statistics: Object-oriented code metrics for the Gitbucket project using our methodology

	count	mean	std	min	25%	50%	75%	max
functionAvrCC	721.00	1.50	1.36	0.00	1.00	1.25	1.67	21.00
functionAvrDON	721.00	3.69	2.53	0.00	2.75	3.00	5.00	17.00
functionAvrFunctionalScore	721.00	0.22	0.30	0.00	0.00	0.12	0.33	1.00
functionAvrNPVS	721.00	2.81	4.00	0.00	0.55	1.00	3.60	34.00
functionAvrOutDegree	721.00	4.89	7.71	0.00	1.00	1.56	5.71	66.00
functionAvrOutDegreeDistinct	721.00	2.45	3.20	0.00	0.79	1.09	3.50	29.00
functionAvrPatternSize	721.00	14.87	19.19	0.00	5.00	7.80	18.00	131.00
functionSumCC	721.00	12.02	15.53	0.00	3.00	6.00	18.00	130.00
functionSumDON	721.00	28.49	39.71	0.00	10.00	16.00	38.00	364.00
functionSumFunctionalScore	721.00	1.90	3.57	0.00	0.00	1.00	2.00	21.00
functionSumNPVS	721.00	20.59	46.07	0.00	3.00	7.00	15.00	372.00
functionSumOutDegree	721.00	35.95	78.95	0.00	4.00	13.00	34.00	768.00
functionSumOutDegreeDistinct	721.00	18.23	39.91	0.00	4.00	7.00	15.00	404.00
functionSumPatternSize	721.00	110.71	209.14	0.00	20.00	50.00	113.00	2078.00
functionMaxCC	721.00	3.71	3.94	0.00	1.00	2.00	5.00	31.00
functionMaxDON	721.00	7.40	4.64	0.00	5.00	6.00	10.00	21.00
functionMaxFunctionalScore	721.00	0.54	0.50	0.00	0.00	1.00	1.00	1.00
functionMaxNPVS	721.00	7.46	10.44	0.00	2.00	4.00	9.00	77.00
functionMaxOutDegree	721.00	14.65	23.27	0.00	2.00	6.00	18.00	176.00
functionMaxOutDegreeDistinct	721.00	5.88	6.71	0.00	2.00	5.00	7.00	48.00
functionMaxPatternSize	721.00	39.22	53.45	0.00	9.00	24.00	49.00	362.00

Table A.3: Descriptive statistics: functional code metrics for the Gitbucket project using our methodology

	count	mean	std	min	25%	50%	75%	max
objectCD	721.00	0.07	0.20	0.00	0.00	0.00	0.03	1.75
objectCLOC	721.00	9.15	29.19	0.00	0.00	0.00	1.00	230.00
objectLOC	721.00	54.82	112.03	0.00	1.00	9.00	39.00	801.00
objectSLOC	721.00	46.03	90.63	0.00	1.00	9.00	37.00	677.00
functionAvrFunctionCD	721.00	0.01	0.03	0.00	0.00	0.00	0.00	0.42
functionAvrFunctionCLOC	721.00	0.16	0.53	0.00	0.00	0.00	0.00	4.50
functionAvrFunctionLOC	721.00	4.19	7.93	0.00	0.00	0.00	6.00	67.00
functionAvrFunctionSLOC	721.00	4.04	7.60	0.00	0.00	0.00	6.00	63.00
functionSumFunctionCD	721.00	0.04	0.13	0.00	0.00	0.00	0.00	1.25
functionSumFunctionCLOC	721.00	0.98	2.84	0.00	0.00	0.00	0.00	19.00
functionSumFunctionLOC	721.00	27.36	72.64	0.00	0.00	0.00	23.00	738.00
functionSumFunctionSLOC	721.00	26.45	70.39	0.00	0.00	0.00	21.00	721.00
functionMaxFunctionCD	721.00	0.03	0.09	0.00	0.00	0.00	0.00	1.00
functionMaxFunctionCLOC	721.00	0.64	1.72	0.00	0.00	0.00	0.00	12.00
functionMaxFunctionLOC	721.00	10.14	18.46	0.00	0.00	0.00	13.00	116.00
functionMaxFunctionSLOC	721.00	9.65	17.53	0.00	0.00	0.00	13.00	114.00

Table A.4: Descriptive statistics: general code metrics for the Gitbucket project using our methodology

Metric	Constant	Coefficient	P-value	R^2	Completeness	Correctness
CBO	-2.6323	0.0934	**0.0000**	0.0643	9.27%	57.14%
ClassInheritance	-1.5493	-1.9226	**0.0000**	0.0598	0.00%	0.00%
DIT	-0.0501	-1.5329	**0.0002**	0.0449	0.00%	0.00%
Inheritance	-0.7665	-1.2216	**0.0000**	0.1617	0.00%	0.00%
LCOM	-2.0595	0.0063	**0.0013**	0.0230	7.80%	100.00%
LCOMneg	-2.0594	0.0063	**0.0013**	0.0230	7.80%	100.00%
NOC	-1.8486	-12.2475	0.9999	0.0070	0.00%	0.00%
RFC	-2.5440	0.0275	**0.0000**	0.0919	19.02%	61.54%
TraitInheritance	-0.9710	-1.2684	**0.0000**	0.1216	0.00%	0.00%
WMCcc	-2.1500	0.0257	**0.0094**	0.0139	4.39%	100.00%
WMCccInit	-2.5627	0.0461	**0.0000**	0.0733	16.59%	85.71%
WMCnormal	-1.9830	0.0174	0.3751	0.0017	0.00%	0.00%
WMCnormalInit	-2.0004	0.0174	0.3751	0.0017	0.00%	0.00%

Table A.5: Univariate logistic regression: object-oriented metrics for the Gitbucket project using our methodology

Metric	Constant	Coefficient	P-value	R^2	Completeness	Correctness
functionAvrCC	-2.2648	0.2708	**0.0089**	0.0211	2.44%	33.33%
functionAvrDON	-3.5170	0.4350	**0.0000**	0.1491	14.15%	47.06%
functionAvrFunctionalScore	-2.3313	2.1979	**0.0000**	0.0689	0.00%	0.00%
functionAvrNPVS	-2.5789	0.2796	**0.0000**	0.1260	17.07%	66.67%
functionAvrOutDegree	-2.4930	0.1452	**0.0000**	0.1377	18.05%	63.16%
functionAvrOutDegreeDistinct	-2.6506	0.3379	**0.0000**	0.1372	21.46%	60.00%
functionAvrPatternSize	-2.7085	0.0620	**0.0000**	0.1453	11.22%	52.63%
functionSumCC	-2.1359	0.0246	**0.0132**	0.0126	4.39%	100.00%
functionSumDON	-2.3918	0.0214	**0.0000**	0.0446	13.66%	100.00%
functionSumFunctionalScore	-2.3359	0.3428	**0.0000**	0.0697	19.02%	75.00%
functionSumNPVS	-2.6079	0.0592	**0.0000**	0.1440	35.61%	72.73%
functionSumOutDegree	-2.6251	0.0330	**0.0000**	0.1418	39.02%	73.91%
functionSumOutDegreeDistinct	-2.5476	0.0555	**0.0000**	0.1076	24.39%	68.75%
functionSumPatternSize	-2.5488	0.0085	**0.0000**	0.0977	22.44%	72.73%
functionMaxCC	-2.1303	0.0729	**0.0233**	0.0105	0.00%	0.00%
functionMaxDON	-2.6996	0.1138	**0.0001**	0.0362	0.00%	0.00%
functionMaxFunctionalScore	-2.3031	0.7938	**0.0019**	0.0221	0.00%	0.00%
functionMaxNPVS	-2.8134	0.1518	**0.0000**	0.1289	25.85%	62.50%
functionMaxOutDegree	-2.6365	0.0652	**0.0000**	0.1221	26.83%	66.67%
functionMaxOutDegreeDistinct	-2.8795	0.1871	**0.0000**	0.1247	21.46%	61.11%
functionMaxPatternSize	-2.7352	0.0264	**0.0000**	0.1116	25.37%	75.00%

Table A.6: Univariate logistic regression: functional metrics for the Gitbucket project using our methodology

Metric	Constant	Coefficient	P-value	R^2	Completeness	Correctness
objectCD	-1.9268	1.1477	**0.0457**	0.0077	0.00%	0.00%
objectCLOC	-2.1284	0.1155	**0.0000**	0.0989	39.51%	76.92%
objectLOC	-2.8514	0.0361	**0.0000**	0.2688	60.98%	75.00%
objectSLOC	-2.9050	0.0409	**0.0000**	0.2760	61.46%	80.00%
functionAvrFunctionCD	-1.9649	15.2246	**0.0038**	0.0301	0.98%	66.67%
functionAvrFunctionCLOC	-1.9927	1.0368	**0.0002**	0.0424	7.32%	50.00%
functionAvrFunctionLOC	-2.3417	0.1248	**0.0000**	0.1214	10.24%	47.06%
functionAvrFunctionSLOC	-2.3402	0.1279	**0.0000**	0.1202	10.24%	50.00%
functionSumFunctionCD	-2.0400	5.9952	**0.0000**	0.0631	11.22%	70.00%
functionSumFunctionCLOC	-2.0928	0.4619	**0.0000**	0.0935	26.34%	64.29%
functionSumFunctionLOC	-2.5520	0.0463	**0.0000**	0.2094	42.93%	67.74%
functionSumFunctionSLOC	-2.5526	0.0476	**0.0000**	0.2081	40.49%	68.97%
functionMaxFunctionCD	-2.0440	8.1229	**0.0000**	0.0605	5.37%	70.00%
functionMaxFunctionCLOC	-2.0849	0.5811	**0.0000**	0.0828	30.73%	68.75%
functionMaxFunctionLOC	-2.5411	0.0826	**0.0000**	0.1797	36.10%	51.72%
functionMaxFunctionSLOC	-2.5429	0.0863	**0.0000**	0.1775	26.34%	47.83%

Table A.7: Univariate logistic regression: general metrics for the Gitbucket project using our methodology

Metric	Coefficient	P-value
Intercept	-2.9934	**0.0000**
objectSLOC	0.0419	**0.0003**
Inheritance	-1.5674	**0.0000**
functionAvrPatternSize	0.3829	**0.0000**
functionAvrNPVSmatch	-3.2038	**0.0012**
functionSumNPVSmatch	1.2389	**0.0005**
functionAvrOutDegreeDistinct	-0.9493	**0.0001**
functionMaxPatternSize	-0.1840	**0.0000**
WMCnormalInit	0.2924	**0.0000**
functionSumOutDegreeDistinct	-0.1169	**0.0000**
functionMaxOutDegree	0.1526	**0.0072**
functionMaxOutDegreeDistinct	0.5309	**0.0010**
functionMaxNPVS	0.2019	0.0614
functionMaxNPVSmatch	-0.9481	0.0586
functionAvrFunctionCLOC	-1.5188	**0.0113**
CBO	-0.1171	**0.0126**
functionSumFunctionCD	3.3454	0.0511
functionAvrNPVS	-0.4002	0.0902

Table A.8: Multivariate regression: variables for the Gitbucket project using our methodology

A.2 Shadowshock

	mean	std	min	25%	50%	75%	max
CBO	5.82	4.82	0.00	2.00	4.00	8.00	23.00
ClassInheritance	0.10	0.30	0.00	0.00	0.00	0.00	1.00
DIT	1.15	0.53	1.00	1.00	1.00	1.00	5.00
Inheritance	0.31	0.75	0.00	0.00	0.00	0.00	5.00
LCOM	13.91	31.07	0.00	0.00	1.00	10.00	249.00
LCOMneg	13.82	31.12	-10.00	0.00	1.00	10.00	249.00
NOC	0.00	0.00	0.00	0.00	0.00	0.00	0.00
RFC	19.54	24.66	1.00	4.00	12.00	23.00	139.00
TraitInheritance	0.21	0.62	0.00	0.00	0.00	0.00	5.00
WMCcc	9.21	12.14	0.00	1.50	5.00	12.00	80.00
WMCccInit	11.38	14.31	1.00	3.00	6.00	14.00	83.00
WMCnormal	4.08	4.62	0.00	1.00	3.00	5.00	25.00
WMCnormalInit	5.08	4.62	1.00	2.00	4.00	6.00	26.00

Table A.9: Descriptive statistics: Object-oriented code metrics for the Shadowshock project using Briand's methodology

	mean	std	min	25%	50%	75%	max
functionAvrCC	1.92	1.55	0.00	1.00	1.50	2.65	9.00
functionAvrDON	3.66	2.47	0.00	2.12	3.67	5.00	12.00
functionAvrFunctionalScore	0.13	0.23	0.00	0.00	0.00	0.19	1.00
functionAvrNPVS	1.79	2.41	0.00	0.10	1.33	2.83	19.00
functionAvrOutDegree	4.99	5.63	0.00	0.50	3.67	7.50	28.00
functionAvrOutDegreeDistinct	3.17	3.26	0.00	0.50	2.43	5.00	20.00
functionAvrPatternSize	13.74	16.96	0.00	2.25	10.00	19.00	128.00
functionSumCC	9.21	12.14	0.00	1.50	5.00	12.00	80.00
functionSumDON	17.41	21.63	0.00	3.50	11.00	22.00	112.00
functionSumFunctionalScore	0.63	1.19	0.00	0.00	0.00	1.00	7.36
functionSumNPVS	7.91	11.27	0.00	0.50	4.00	10.00	56.00
functionSumOutDegree	24.31	41.46	0.00	1.00	10.00	26.00	313.00
functionSumOutDegreeDistinct	15.03	22.32	0.00	1.00	8.00	18.00	146.00
functionSumPatternSize	63.99	98.05	0.00	6.00	29.00	74.50	689.00
functionMaxCC	3.58	3.54	0.00	1.00	3.00	5.00	18.00
functionMaxDON	5.47	3.69	0.00	2.50	6.00	8.00	17.00
functionMaxFunctionalScore	0.33	0.44	0.00	0.00	0.00	1.00	1.00
functionMaxNPVS	3.79	4.37	0.00	0.50	2.00	5.50	20.00
functionMaxOutDegree	11.67	17.90	0.00	1.00	8.00	13.50	155.00
functionMaxOutDegreeDistinct	6.28	7.08	0.00	1.00	5.00	9.00	51.00
functionMaxPatternSize	29.51	38.56	0.00	3.50	19.00	37.50	295.00

Table A.10: Descriptive statistics: Functional code metrics for the Shadowshock project using Briand's methodology

	mean	std	min	25%	50%	75%	max
objectCD	0.04	0.09	0.00	0.00	0.00	0.04	0.44
objectCLOC	1.33	2.66	0.00	0.00	0.00	1.00	16.00
objectLOC	38.24	59.04	0.00	6.00	17.00	42.00	338.00
objectSLOC	37.44	58.12	0.00	6.00	16.00	40.50	335.00
functionAvrFunctionCD	0.01	0.02	0.00	0.00	0.00	0.00	0.21
functionAvrFunctionCLOC	0.12	0.37	0.00	0.00	0.00	0.02	3.00
functionAvrFunctionLOC	5.67	6.53	0.00	0.00	3.80	8.33	39.00
functionAvrFunctionSLOC	5.62	6.43	0.00	0.00	3.80	8.25	38.00
functionSumFunctionCD	0.05	0.11	0.00	0.00	0.00	0.01	0.62
functionSumFunctionCLOC	0.63	1.72	0.00	0.00	0.00	0.50	13.00
functionSumFunctionLOC	27.54	45.89	0.00	0.00	11.00	26.50	308.00
functionSumFunctionSLOC	27.31	45.33	0.00	0.00	11.00	26.50	307.00
functionMaxFunctionCD	0.04	0.08	0.00	0.00	0.00	0.01	0.33
functionMaxFunctionCLOC	0.40	0.82	0.00	0.00	0.00	0.50	4.00
functionMaxFunctionLOC	12.29	18.12	0.00	0.00	7.00	14.50	135.00
functionMaxFunctionSLOC	12.18	17.91	0.00	0.00	7.00	14.00	134.00

Table A.11: Descriptive statistics: General code metrics for the Shadowshock project using Briand's methodology

Metric	Constant	Coefficient	P-value	R^2	Completeness	Correctness
CBO	-3.1920	0.2522	**0.0000**	0.2317	32.88%	66.67%
ClassInheritance	-1.3531	-0.3516	0.6617	0.0016	0.00%	0.00%
DIT	-0.8874	-0.4465	0.4696	0.0056	0.00%	0.00%
Inheritance	-1.5198	0.3537	0.1630	0.0148	1.37%	100.00%
LCOM	-1.7927	0.0238	**0.0044**	0.0862	17.81%	71.43%
LCOMneg	-1.7798	0.0231	**0.0051**	0.0831	17.81%	71.43%
NOC	N/A	N/A	N/A	N/A	N/A	N/A
RFC	-2.6465	0.0531	**0.0000**	0.2346	35.62%	76.92%
TraitInheritance	-1.5373	0.5690	0.0838	0.0261	1.37%	100.00%
WMCcc	-2.4886	0.0972	**0.0000**	0.2050	35.62%	83.33%
WMCccInit	-2.6200	0.0909	**0.0001**	0.2258	35.62%	83.33%
WMCnormal	-2.4313	0.2087	**0.0001**	0.1584	26.03%	63.64%
WMCnormalInit	-2.6400	0.2087	**0.0001**	0.1584	26.03%	63.64%

Table A.12: Univariate logistic regression: object-oriented metrics for the Shadowshock project using Briand's methodology

Metric	Constant	Coefficient	P-value	R^2	Completeness	Correctness
functionAvrCC	-2.0897	0.3353	**0.0179**	0.0469	2.74%	25.00%
functionAvrDON	-1.9796	0.1519	0.0861	0.0238	0.00%	0.00%
functionAvrFunctionalScore	-1.5859	1.4051	0.1110	0.0192	0.00%	0.00%
functionAvrNPVS	-1.5568	0.0897	0.2589	0.0097	0.00%	0.00%
functionAvrOutDegree	-2.0245	0.1096	**0.0048**	0.0683	1.37%	33.33%
functionAvrOutDegreeDistinct	-1.8529	0.1315	**0.0426**	0.0336	0.00%	0.00%
functionAvrPatternSize	-1.7102	0.0215	0.0769	0.0268	0.00%	0.00%
functionSumCC	-2.4886	0.0972	**0.0000**	0.2050	35.62%	83.33%
functionSumDON	-2.4452	0.0495	**0.0001**	0.1867	26.03%	72.73%
functionSumFunctionalScore	-1.9715	0.7239	**0.0005**	0.1317	17.81%	75.00%
functionSumNPVS	-2.2733	0.0896	**0.0000**	0.1776	23.29%	72.73%
functionSumOutDegree	-2.2105	0.0280	**0.0001**	0.1771	31.51%	72.73%
functionSumOutDegreeDistinct	-2.2818	0.0479	**0.0001**	0.1815	35.62%	76.92%
functionSumPatternSize	-2.3016	0.0116	**0.0000**	0.1876	35.62%	76.92%
functionMaxCC	-2.8286	0.3337	**0.0000**	0.2033	24.66%	70.00%
functionMaxDON	-2.8795	0.2360	**0.0008**	0.1076	5.48%	33.33%
functionMaxFunctionalScore	-2.3384	2.1714	**0.0000**	0.1475	0.00%	0.00%
functionMaxNPVS	-2.3483	0.2063	**0.0001**	0.1440	15.07%	55.56%
functionMaxOutDegree	-2.0997	0.0532	**0.0017**	0.1059	20.55%	83.33%
functionMaxOutDegreeDistinct	-2.2154	0.1130	**0.0019**	0.0979	16.44%	50.00%
functionMaxPatternSize	-2.1913	0.0233	**0.0009**	0.1151	20.55%	62.50%

Table A.13: Univariate logistic regression: functional metrics for the Shadowshock project using Briand's methodology

Metric	Constant	Coefficient	P-value	R^2	Completeness	Correctness
objectCD	-1.3662	-0.5264	0.8410	0.0003	0.00%	0.00%
objectCLOC	-1.9015	0.2958	**0.0006**	0.1183	15.07%	57.14%
objectLOC	-2.5268	0.0246	**0.0000**	0.2486	31.51%	72.73%
objectSLOC	-2.5185	0.0251	**0.0001**	0.2469	31.51%	72.73%
functionAvrFunctionCD	-1.4947	10.3770	0.1916	0.0139	0.00%	0.00%
functionAvrFunctionCLOC	-1.4437	0.4066	0.4272	0.0047	0.00%	0.00%
functionAvrFunctionLOC	-2.0126	0.0952	**0.0052**	0.0693	2.74%	25.00%
functionAvrFunctionSLOC	-2.0230	0.0975	**0.0048**	0.0703	2.74%	25.00%
functionSumFunctionCD	-1.8387	7.1961	**0.0005**	0.1264	21.92%	75.00%
functionSumFunctionCLOC	-1.7054	0.4120	**0.0084**	0.0832	13.70%	75.00%
functionSumFunctionLOC	-2.3068	0.0268	**0.0000**	0.2053	32.88%	81.82%
functionSumFunctionSLOC	-2.3101	0.0271	**0.0000**	0.2054	32.88%	81.82%
functionMaxFunctionCD	-1.9108	11.0391	**0.0002**	0.1312	23.29%	70.00%
functionMaxFunctionCLOC	-1.7968	0.7933	**0.0019**	0.0847	12.33%	40.00%
functionMaxFunctionLOC	-2.1535	0.0524	**0.0005**	0.1299	19.18%	50.00%
functionMaxFunctionSLOC	-2.1621	0.0535	**0.0005**	0.1314	19.18%	50.00%

Table A.14: Univariate logistic regression: general metrics for the Shadowshock project using Briand's methodology

Metric	Coefficient	P-value
Intercept	3.2056	0.9112
objectLOC	0.1146	**0.0003**
functionMaxOutDegreeDistinct	-0.8927	**0.0060**
CBO	0.2326	0.2255
LCOMneg	-0.7072	0.2527
LCOM	0.6740	0.2764
functionMaxOutDegree	0.1532	0.0993
functionMaxFunctionalScore	5.0811	**0.0081**
functionSumFunctionalScore	-2.4568	**0.0101**
TraitInheritance	2.1980	**0.0182**
DIT	-7.3190	0.7992

Table A.15: Multivariate regression: variables for the Shadowshock project using Briand's methodology

	count	mean	std	min	25%	50%	75%	max
CBO	156.00	7.28	6.39	0.00	3.00	5.00	9.25	30.00
ClassInheritance	156.00	0.07	0.26	0.00	0.00	0.00	0.00	1.00
DIT	156.00	1.11	0.48	1.00	1.00	1.00	1.00	5.00
Inheritance	156.00	0.22	0.69	0.00	0.00	0.00	0.00	5.00
LCOM	156.00	28.13	67.10	0.00	0.00	3.00	21.25	404.00
LCOMneg	156.00	28.11	67.10	-1.00	0.00	3.00	21.25	404.00
NOC	156.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
RFC	156.00	26.98	32.04	1.00	6.00	14.50	32.00	131.00
TraitInheritance	156.00	0.15	0.56	0.00	0.00	0.00	0.00	5.00
WMCccInit	156.00	13.87	19.40	0.00	2.00	7.00	16.00	84.00
WMCccInit	156.00	16.67	20.98	1.00	3.00	9.00	19.00	85.00
WMCnormal	156.00	5.52	6.22	0.00	1.00	3.00	8.00	30.00
WMCnormalInit	156.00	6.52	6.22	1.00	2.00	4.00	9.00	31.00

Table A.16: Descriptive statistics: Object-oriented code metrics for the Shadowshock project using our methodology

	count	mean	std	min	25%	50%	75%	max
functionAvrCC	156.00	2.06	1.47	0.00	1.00	1.80	2.88	9.00
functionAvrDON	156.00	3.82	2.30	0.00	2.74	4.00	5.00	12.00
functionAvrFunctionalScore	156.00	0.13	0.22	0.00	0.00	0.00	0.17	1.00
functionAvrNPVS	156.00	1.96	2.37	0.00	0.36	1.50	3.00	19.00
functionAvrOutDegree	156.00	5.56	5.54	0.00	1.00	4.08	8.34	28.00
functionAvrOutDegreeDistinct	156.00	3.39	3.13	0.00	1.00	2.72	5.33	20.00
functionAvrPatternSize	156.00	15.07	16.29	0.00	4.67	11.17	22.00	128.00
functionSumCC	156.00	13.87	19.40	0.00	2.00	7.00	16.00	84.00
functionSumDON	156.00	24.31	30.63	0.00	5.00	13.50	30.50	144.00
functionSumFunctionalScore	156.00	0.75	1.24	0.00	0.00	0.00	1.00	7.36
functionSumNPVS	156.00	12.69	19.68	0.00	1.00	4.50	12.00	98.00
functionSumOutDegree	156.00	40.56	71.79	0.00	3.75	15.00	35.25	364.00
functionSumOutDegreeDistinct	156.00	23.36	36.64	0.00	2.00	10.00	22.25	185.00
functionSumPatternSize	156.00	105.38	173.49	0.00	12.25	41.50	102.00	828.00
functionMaxCC	156.00	4.48	4.65	0.00	1.00	3.00	5.00	26.00
functionMaxDON	156.00	6.12	3.78	0.00	4.00	6.00	8.00	17.00
functionMaxFunctionalScore	156.00	0.38	0.45	0.00	0.00	0.00	1.00	1.00
functionMaxNPVS	156.00	4.89	5.94	0.00	1.00	3.00	7.00	32.00
functionMaxOutDegree	156.00	15.10	21.93	0.00	2.00	9.00	18.25	155.00
functionMaxOutDegreeDistinct	156.00	7.84	8.52	0.00	2.00	6.00	10.25	51.00
functionMaxPatternSize	156.00	38.04	49.39	0.00	8.75	23.00	46.25	295.00

Table A.17: Descriptive statistics: functional code metrics for the Shadowshock project using our methodology

	count	mean	std	min	25%	50%	75%	max
objectCD	156.00	0.04	0.08	0.00	0.00	0.00	0.04	0.44
objectCLOC	156.00	2.03	3.63	0.00	0.00	0.00	2.00	16.00
objectLOC	156.00	67.07	102.76	0.00	8.00	23.50	68.25	471.00
objectSLOC	156.00	65.62	100.12	0.00	7.75	23.50	68.25	455.00
functionAvrFunctionCD	156.00	0.01	0.02	0.00	0.00	0.00	0.01	0.21
functionAvrFunctionCLOC	156.00	0.14	0.35	0.00	0.00	0.00	0.14	3.00
functionAvrFunctionLOC	156.00	6.65	6.62	0.00	1.00	5.41	9.12	39.00
functionAvrFunctionSLOC	156.00	6.59	6.51	0.00	1.00	5.41	9.12	38.00
functionSumFunctionCD	156.00	0.07	0.14	0.00	0.00	0.00	0.07	0.59
functionSumFunctionCLOC	156.00	1.13	2.69	0.00	0.00	0.00	1.00	12.00
functionSumFunctionLOC	156.00	50.88	91.38	0.00	2.00	17.00	52.00	442.00
functionSumFunctionSLOC	156.00	50.16	89.32	0.00	2.00	17.00	51.25	431.00
functionMaxFunctionCD	156.00	0.05	0.09	0.00	0.00	0.00	0.07	0.33
functionMaxFunctionCLOC	156.00	0.67	1.41	0.00	0.00	0.00	1.00	7.00
functionMaxFunctionLOC	156.00	17.19	24.94	0.00	1.00	10.00	20.00	136.00
functionMaxFunctionSLOC	156.00	17.03	24.70	0.00	1.00	10.00	20.00	136.00

Table A.18: Descriptive statistics: general code metrics for the Shadowshock project using our methodology

Metric	Constant	Coefficient	P-value	R^2	Completeness	Correctness
CBO	-3.0680	0.2430	**0.0000**	0.2082	38.18%	61.54%
ClassInheritance	-1.2698	-19.4100	0.9983	0.0413	0.00%	0.00%
DIT	N/A	N/A	N/A	N/A	N/A	N/A
Inheritance	-1.3911	0.0174	0.9526	0.0000	0.00%	0.00%
LCOM	-1.7328	0.0175	**0.0166**	0.0979	20.00%	75.00%
LCOMneg	-1.7319	0.0174	**0.0166**	0.0977	20.00%	75.00%
NOC	N/A	N/A	N/A	N/A	N/A	N/A
RFC	-2.5234	0.0499	**0.0001**	0.2100	36.36%	70.00%
TraitInheritance	-1.4424	0.2650	0.3975	0.0053	0.00%	0.00%
WMCcc	-2.3010	0.0839	**0.0007**	0.1779	29.09%	75.00%
WMCccInit	-2.4989	0.0831	**0.0003**	0.2124	36.36%	77.78%
WMCnormal	-2.4363	0.2042	**0.0001**	0.1754	21.82%	50.00%
WMCnormalInit	-2.6405	0.2042	**0.0001**	0.1754	21.82%	50.00%

Table A.19: Univariate logistic regression: object-oriented metrics for the Shadowshock project using our methodology

Metric	Constant	Coefficient	P-value	R^2	Completeness	Correctness
functionAvrCC	-2.0037	0.3013	**0.0347**	0.0364	0.00%	0.00%
functionAvrDON	-2.0367	0.1652	0.0644	0.0278	0.00%	0.00%
functionAvrFunctionalScore	-1.6086	1.5097	0.0770	0.0237	0.00%	0.00%
functionAvrNPVS	-1.6479	0.1289	0.1046	0.0216	0.00%	0.00%
functionAvrOutDegree	-1.9216	0.0962	**0.0137**	0.0503	1.82%	33.33%
functionAvrOutDegreeDistinct	-1.8007	0.1191	0.0643	0.0274	0.00%	0.00%
functionAvrPatternSize	-1.6999	0.0209	0.0834	0.0254	0.00%	0.00%
functionSumCC	-2.3010	0.0839	**0.0007**	0.1779	29.09%	75.00%
functionSumDON	-2.3973	0.0474	**0.0003**	0.1879	29.09%	66.67%
functionSumFunctionalScore	-1.8841	0.6450	**0.0012**	0.1082	16.36%	71.43%
functionSumNPVS	-2.1962	0.0768	**0.0002**	0.1873	38.18%	72.73%
functionSumOutDegree	-2.0138	0.0226	**0.0043**	0.1384	27.27%	71.43%
functionSumOutDegreeDistinct	-2.1309	0.0414	**0.0010**	0.1559	29.09%	66.67%
functionSumPatternSize	-2.1306	0.0096	**0.0013**	0.1645	36.36%	70.00%
functionMaxCC	-2.6170	0.2995	**0.0001**	0.1643	32.73%	66.67%
functionMaxDON	-3.0476	0.2587	**0.0004**	0.1253	14.55%	28.57%
functionMaxFunctionalScore	-2.2471	2.0221	**0.0001**	0.1307	0.00%	0.00%
functionMaxNPVS	-2.3408	0.1978	**0.0001**	0.1579	29.09%	60.00%
functionMaxOutDegree	-1.8722	0.0386	**0.0238**	0.0715	9.09%	50.00%
functionMaxOutDegreeDistinct	-2.1911	0.1101	**0.0024**	0.0973	21.82%	50.00%
functionMaxPatternSize	-2.0434	0.0194	**0.0044**	0.0981	14.55%	50.00%

Table A.20: Univariate logistic regression: functional metrics for the Shadowshock project using our methodology

Metric	Constant	Coefficient	P-value	R^2	Completeness	Correctness
objectCD	-1.3463	-1.1073	0.6897	0.0014	0.00%	0.00%
objectCLOC	-1.8610	0.2628	**0.0006**	0.1141	23.64%	62.50%
objectLOC	-2.4845	0.0231	**0.0002**	0.2584	43.64%	80.00%
objectSLOC	-2.4837	0.0236	**0.0002**	0.2586	43.64%	72.73%
functionAvrFunctionCD	-1.4689	8.3583	0.2856	0.0088	0.00%	0.00%
functionAvrFunctionCLOC	-1.4135	0.2143	0.6928	0.0012	0.00%	0.00%
functionAvrFunctionLOC	-2.0326	0.0993	**0.0055**	0.0689	0.00%	0.00%
functionAvrFunctionSLOC	-2.0484	0.1023	**0.0049**	0.0707	0.00%	0.00%
functionSumFunctionCD	-1.7222	6.0156	**0.0026**	0.0854	25.45%	66.67%
functionSumFunctionCLOC	-1.6253	0.3079	**0.0133**	0.0628	23.64%	75.00%
functionSumFunctionLOC	-2.2241	0.0254	**0.0008**	0.1916	34.55%	75.00%
functionSumFunctionSLOC	-2.2281	0.0256	**0.0008**	0.1925	36.36%	77.78%
functionMaxFunctionCD	-1.7797	9.1228	**0.0012**	0.0916	27.27%	62.50%
functionMaxFunctionCLOC	-1.6563	0.5338	**0.0124**	0.0582	16.36%	66.67%
functionMaxFunctionLOC	-2.0378	0.0462	**0.0036**	0.1108	21.82%	50.00%
functionMaxFunctionSLOC	-2.0545	0.0480	**0.0034**	0.1128	21.82%	50.00%

Table A.21: Univariate logistic regression: general metrics for the Shadowshock project using our methodology

Metric	Coefficient	P-value
Intercept	-3.8958	**0.0000**
objectSLOC	0.1410	**0.0001**
functionSumOutDegree	-0.1835	**0.0002**
functionMaxNPVSmatchParms	1.8114	**0.0017**
functionAvrNPVS	-2.2946	**0.0045**
functionAvrOutDegree	1.2529	**0.0153**
functionAvrOutDegreeDistinct	-1.0961	0.0694
functionAvrDON	-0.5044	0.2893

Table A.22: Multivariate regression: variables for the Shadowshock project using our methodology

A.3 HTTP Akka

	mean	std	min	25%	50%	75%	max
CBO	7.78	7.29	0.00	3.00	6.00	10.00	78.00
ClassInheritance	0.39	0.49	0.00	0.00	0.00	1.00	1.00
DIT	1.48	0.70	1.00	1.00	1.00	2.00	5.00
Inheritance	1.46	1.18	0.00	0.00	1.00	2.00	4.00
LCOM	40.59	239.97	0.00	1.00	6.00	35.00	6328.00
LCOMneg	40.52	239.99	-20.00	1.00	6.00	35.00	6328.00
NOC	0.43	2.51	0.00	0.00	0.00	0.00	70.00
RFC	19.86	29.13	0.00	5.00	11.00	25.00	465.00
TraitInheritance	1.06	1.13	0.00	0.00	1.00	2.00	4.00
WMCcc	8.98	11.64	0.00	2.00	5.00	14.00	132.00
WMCccInit	11.68	14.33	1.00	4.00	7.00	17.00	206.00
WMCnormal	6.67	8.17	0.00	2.00	4.00	10.00	113.00
WMCnormalInit	7.67	8.17	1.00	3.00	5.00	11.00	114.00

Table A.23: Descriptive statistics: Object-oriented code metrics for the HTTP Akka project using Briand's methodology

	mean	std	min	25%	50%	75%	max
functionAvrCC	1.15	0.71	0.00	1.00	1.25	1.36	9.00
functionAvrDON	2.62	1.74	0.00	2.00	2.75	3.00	15.00
functionAvrFunctionalScore	0.13	0.20	0.00	0.00	0.00	0.18	1.00
functionAvrNPVS	1.28	4.16	0.00	0.25	0.55	1.00	93.00
functionAvrOutDegree	2.42	9.47	0.00	0.46	1.00	1.43	180.00
functionAvrOutDegreeDistinct	1.36	3.04	0.00	0.44	1.00	1.22	61.00
functionAvrPatternSize	8.24	27.40	0.00	2.50	4.58	6.00	547.00
functionSumCC	8.92	11.63	0.00	2.00	5.00	14.00	132.00
functionSumDON	20.01	32.94	0.00	5.00	12.00	29.00	652.00
functionSumFunctionalScore	1.05	1.99	0.00	0.00	0.00	2.00	27.76
functionSumNPVS	7.01	17.25	0.00	2.00	3.00	7.00	286.00
functionSumOutDegree	14.19	39.48	0.00	1.00	4.00	13.00	658.00
functionSumOutDegreeDistinct	9.09	19.04	0.00	1.00	4.00	12.00	287.00
functionSumPatternSize	48.99	106.91	0.00	7.00	20.00	55.00	1597.00
functionMaxCC	2.22	2.22	0.00	1.00	2.00	3.00	24.00
functionMaxDON	4.96	3.18	0.00	3.00	6.00	6.00	24.00
functionMaxFunctionalScore	0.48	0.50	0.00	0.00	0.00	1.00	1.00
functionMaxNPVS	3.04	6.49	0.00	1.00	2.00	2.00	93.00
functionMaxOutDegree	6.19	15.39	0.00	1.00	2.00	5.00	180.00
functionMaxOutDegreeDistinct	3.37	4.79	0.00	1.00	2.00	4.00	61.00
functionMaxPatternSize	19.21	42.18	0.00	4.00	9.00	17.00	547.00

Table A.24: Descriptive statistics: Functional code metrics for the HTTP Akka project using Briand's methodology

	mean	std	min	25%	50%	75%	max
objectCD	0.09	0.35	0.00	0.00	0.00	0.04	8.53
objectCLOC	2.45	12.05	0.00	0.00	0.00	1.00	214.00
objectLOC	16.53	46.91	0.00	1.00	4.00	11.00	545.00
objectSLOC	14.28	38.83	0.00	1.00	4.00	10.00	484.00
functionAvrFunctionCD	0.00	0.02	0.00	0.00	0.00	0.00	0.33
functionAvrFunctionCLOC	0.12	0.94	0.00	0.00	0.00	0.00	14.50
functionAvrFunctionLOC	2.06	9.85	0.00	0.00	0.21	1.00	175.00
functionAvrFunctionSLOC	1.96	9.26	0.00	0.00	0.21	1.00	169.00
functionSumFunctionCD	0.02	0.13	0.00	0.00	0.00	0.00	2.00
functionSumFunctionCLOC	0.37	2.29	0.00	0.00	0.00	0.00	36.00
functionSumFunctionLOC	9.16	29.72	0.00	0.00	1.00	5.00	346.00
functionSumFunctionSLOC	8.92	28.94	0.00	0.00	1.00	5.00	346.00
functionMaxFunctionCD	0.01	0.08	0.00	0.00	0.00	0.00	1.00
functionMaxFunctionCLOC	0.30	1.99	0.00	0.00	0.00	0.00	35.00
functionMaxFunctionLOC	4.42	14.85	0.00	0.00	1.00	2.00	175.00
functionMaxFunctionSLOC	4.21	13.84	0.00	0.00	1.00	2.00	169.00

Table A.25: Descriptive statistics: General code metrics for the HTTP Akka project using Briand's methodology

Metric	Constant	Coefficient	P-value	R^2	Completeness	Correctness
CBO	-3.5546	0.1238	**0.0000**	0.1224	6.81%	61.54%
ClassInheritance	-2.1197	-1.0447	**0.0007**	0.0258	0.00%	0.00%
DIT	-1.5447	-0.6428	**0.0082**	0.0171	0.00%	0.00%
Inheritance	-1.5925	-0.7704	**0.0000**	0.0827	0.00%	0.00%
LCOM	-2.6187	0.0037	**0.0000**	0.0633	3.89%	71.43%
LCOMneg	-2.6183	0.0037	**0.0000**	0.0633	3.89%	71.43%
NOC	-2.3891	-0.1312	0.3792	0.0022	0.00%	0.00%
RFC	-3.2835	0.0307	**0.0000**	0.1580	12.41%	57.89%
TraitInheritance	-1.8293	-0.7665	**0.0000**	0.0605	0.00%	0.00%
WMCcc	-2.9654	0.0470	**0.0000**	0.0702	3.89%	80.00%
WMCccInit	-3.4476	0.0637	**0.0000**	0.1634	13.14%	72.22%
WMCnormal	-2.9695	0.0661	**0.0000**	0.0631	3.89%	71.43%
WMCnormalInit	-3.0356	0.0661	**0.0000**	0.0631	3.89%	71.43%

Table A.26: Univariate logistic regression: object-oriented metrics for the HTTP Akka project using Briand's methodology

Metric	Constant	Coefficient	P-value	R^2	Completeness	Correctness
functionAvrCC	-2.9342	0.4100	**0.0033**	0.0154	0.00%	0.00%
functionAvrDON	-3.5602	0.3589	**0.0000**	0.0752	2.43%	42.86%
functionAvrFunctionalScore	-2.6985	1.6986	**0.0003**	0.0221	0.00%	0.00%
functionAvrNPVS	-2.6578	0.1215	**0.0000**	0.0573	3.16%	50.00%
functionAvrOutDegree	-2.6095	0.0498	**0.0000**	0.0483	3.41%	57.14%
functionAvrOutDegreeDistinct	-2.7244	0.1642	**0.0000**	0.0532	1.22%	33.33%
functionAvrPatternSize	-2.6266	0.0173	**0.0000**	0.0455	2.68%	57.14%
functionSumCC	-2.9520	0.0462	**0.0000**	0.0682	3.89%	80.00%
functionSumDON	-3.0298	0.0232	**0.0000**	0.0980	4.87%	80.00%
functionSumFunctionalScore	-2.7663	0.2345	**0.0000**	0.0565	3.89%	62.50%
functionSumNPVS	-2.9919	0.0552	**0.0000**	0.1384	6.81%	73.33%
functionSumOutDegree	-2.9071	0.0216	**0.0000**	0.1332	7.30%	73.33%
functionSumOutDegreeDistinct	-3.0510	0.0491	**0.0000**	0.1320	7.06%	78.57%
functionSumPatternSize	-3.0063	0.0079	**0.0000**	0.1377	8.76%	81.25%
functionMaxCC	-2.8295	0.1543	**0.0000**	0.0301	0.73%	50.00%
functionMaxDON	-3.6431	0.2052	**0.0000**	0.0662	0.00%	0.00%
functionMaxFunctionalScore	-2.7530	0.6038	**0.0185**	0.0110	0.00%	0.00%
functionMaxNPVS	-2.8282	0.0861	**0.0000**	0.0927	4.62%	54.55%
functionMaxOutDegree	-2.7535	0.0324	**0.0000**	0.0826	4.87%	53.85%
functionMaxOutDegreeDistinct	-2.9799	0.1190	**0.0000**	0.0896	3.41%	45.45%
functionMaxPatternSize	-2.7888	0.0124	**0.0000**	0.0841	5.35%	61.54%

Table A.27: Univariate logistic regression: functional metrics for the HTTP Akka project using Briand's methodology

Metric	Constant	Coefficient	P-value	R^2	Completeness	Correctness
objectCD	-2.5403	1.0176	**0.0016**	0.0257	0.24%	25.00%
objectCLOC	-2.8530	0.1093	**0.0000**	0.1704	14.11%	75.00%
objectLOC	-3.1113	0.0234	**0.0000**	0.2189	18.49%	80.00%
objectSLOC	-3.0489	0.0245	**0.0000**	0.1968	17.76%	73.91%
functionAvrFunctionCD	-2.4952	10.7576	**0.0010**	0.0184	0.00%	0.00%
functionAvrFunctionCLOC	-2.4967	0.2878	**0.0001**	0.0291	1.22%	40.00%
functionAvrFunctionLOC	-2.5685	0.0420	**0.0000**	0.0471	1.95%	60.00%
functionAvrFunctionSLOC	-2.5728	0.0461	**0.0000**	0.0469	1.95%	50.00%
functionSumFunctionCD	-2.5963	3.9247	**0.0000**	0.0716	5.35%	72.73%
functionSumFunctionCLOC	-2.5900	0.2270	**0.0000**	0.0671	3.16%	71.43%
functionSumFunctionLOC	-2.9067	0.0291	**0.0000**	0.1642	10.22%	73.68%
functionSumFunctionSLOC	-2.9101	0.0306	**0.0000**	0.1634	10.71%	78.95%
functionMaxFunctionCD	-2.5795	5.0679	**0.0000**	0.0587	0.97%	60.00%
functionMaxFunctionCLOC	-2.5450	0.2080	**0.0000**	0.0470	1.70%	60.00%
functionMaxFunctionLOC	-2.7203	0.0368	**0.0000**	0.0937	5.11%	53.85%
functionMaxFunctionSLOC	-2.7243	0.0399	**0.0000**	0.0929	5.11%	58.33%

Table A.28: Univariate logistic regression: general metrics for the HTTP Akka project using Briand's methodology

Metric	Coefficient	P-value
Intercept	-3.3192	**0.0000**
objectLOC	-0.0020	0.6691
Inheritance	-0.6659	**0.0000**
functionSumDON	0.0454	**0.0000**
CBO	0.1056	**0.0000**
NOC	-0.1835	0.3015
functionSumFunctionalScore	-0.1573	0.0650
functionMaxFunctionLOC	0.0331	**0.0001**
functionSumPatternSize	-0.0069	**0.0028**
objectCLOC	0.0275	0.1160

Table A.29: Multivariate regression: variables for the HTTP Akka project using Briand's methodology

Metric	Coefficient	P-value
Intercept	-2.7075	**0.0000**
objectLOC	-0.3333	**0.0163**
ClassInheritance	-1.2638	**0.0050**
objectCLOC	0.3702	**0.0079**
objectSLOC	0.3335	**0.0151**
CBO	0.0883	**0.0027**
TraitInheritance	-0.4916	**0.0224**
functionSumDON	0.0946	**0.0000**
NOC	-0.3592	0.1103
functionAvrFunctionCLOC	0.1490	0.2700
functionSumPatternSize	-0.0301	**0.0000**
WMCnormalInit	-0.2166	**0.0095**
functionSumCC	0.1325	**0.0031**
functionAvrNPVS	0.1617	**0.0117**
functionSumFunctionSLOC	0.0379	**0.0445**
functionAvrDON	-0.2183	0.1087
LCOMneg	-0.0035	0.0678

Table A.30: Multivariate regression: variables for the HTTP Akka project using our methodology