Code Quality Metrics for the Functional Side of the Object-Oriented Language C#

Bart Zuilhof
bart_zuilhof@hotmail.com

August 18, 2019, 48 pages

Research supervisor: Clemens Grelck, c.grelck@uva.nl
Host supervisor: Rinse van Hees, rinse.vanhees@infosupport.com
Host organisation: Info Support BV., www.infosupport.com

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

Contents

Abstract

With the evolution of object-oriented languages such as C#, new code constructs which originate from the functional programming paradigm are introduced. We hypothesize that a relationship exists between the usage of these constructs and the error-proneness of classes. Code metrics provide a standard for reporting internal attributes of a codebase, such as specific constructs. The code metrics defined in this study will focus on functional programming inspired constructs in which object-oriented features are used. These functional programming inspired constructs often affect the purity of the code. Built on these measures we try to define a code metric that relates the usage of the measured constructs to error-proneness. To validate the code metric that would confirm our hypothesis, we implement the methodology presented by Briand et al. [1] for empirical validation of code metrics. The results of this research granted new insights into the evolution of software systems and the evolution of programming languages, regarding the usage of constructs from the functional programming paradigm in object-oriented languages. Most projects analyzed, gained increased error-proneness prediction performance from the inclusion of our proposed measures. Even though the performance increase seems marginal for now, we hypothesize that the relevance of our proposed measures will increase. This hypothesis is based on the ongoing evolution of C# regarding functional programming.

Chapter 1

Introduction

The popularity of multi-paradigm languages such as Scala is blooming [2]. Features that are inspired by the functional programming (FP) paradigm are added to the significant object-oriented (OO) languages such as Java [3] and C# [4]. Therefore, code evaluation for multi-paradigm has become more significant. Landkroon has shown [5] that metrics from the OO paradigm and the FP paradigm can be mapped to the multi-paradigm language Scala.

For this research, we have chosen to investigate the OO language C#. The reason for this decision is the aggressive evolution of C#. The language C# is able to so evolve at its current speed because of several reasons.

- C# is relatively young. Therefore, it had a clean start. It could not break anything.
- The requirement for backward compatibility is ignored during development for newer versions.
- The company that designed C# is Microsoft, Microsoft has a huge R&D budget. More than twice as big as Oracle's budget, the company managing Java has [6].

A big set of the added features include syntactic sugar for already existent functionality. The syntax changes toward a declarative syntax as described in Section 2.4.

The integration of FP features in OO languages, introduces constructs that are neither covered by OO-targeting code metrics nor by FP-targeting code metrics. These constructs, such as usage of mutable class variables in lambda functions whose execution might be deferred, are not possible in the FP languages but are used in a functional manner. To evaluate these patterns, the metrics that are proposed for the OO paradigm [7–9] and FP paradigm [10, 11] are unsuitable to give a valuable indication of quality regarding the usage of these combined constructs.
We use the term 'measure' as used by Briand et al. [1]. Where the term 'measure' refers to an assessment on the size of an internal attribute of a codebase. Measures that indicate complexity might have an intuitive relationship with the error-proneness of code. Error-prone classes are more vulnerable to errors and more likely to contain errors. But this does not have any concrete meaning and usefulness since it is not reasonable to substantiate a prediction based on an intuition. Therefore, evidence must be provided to show that a measure is useful. This can be done by proving there is a relationship to an external attribute such as error-proneness [1].

The purpose of this research is to explore the relationship between the usage of the FP inspired constructs and the error-proneness of the classes for the language C#. Our approach is defining measures that will cover the usage of these constructs and empirically relate them to the error-proneness of the corresponding class.

1.1 Research Questions

To address the above-described problem, we formulate the following research questions.

RQ *How does usage of functional programming inspired constructs correlate to error-proneness in the object-oriented language C#?*

To be able to answer this question, it is mandatory to analyze the evolution of the programming language C#. Which brings us to the first sub research question:

RQ1 *What new constructs occur, that are neither functional nor object-oriented, when introducing func-*

tional programming inspired features to the object-oriented language C#?

The constructs that we identified in *RQ1* play a major role in this research. Therefore, we introduce the term 'FP inspired constructs' for easy reuse. The term refers to code constructs, occurring in the OO language C#, where the syntax is inspired by the functional programming paradigm. First we create a clear picture of how the programming language C# evolved by adding features that are inspired by the FP paradigm. With this knowledge we can define measures that will capture the constructs that are known neither in the OO nor the FP paradigm.

We add these measures to a baseline error-proneness prediction model (see Section 3.5), which consists of a set of commonly used code metrics. We will try to create an improved version of the error-proneness prediction model. Which leads us to the following research question.

RQ2 *To what extent can we improve our baseline prediction model by adding code metrics covering FP inspired constructs?*

By analyzing the significance of the improvement of the model, we can make a claim on to what extent the FP inspired constructs impact the error-proneness of the classes they are used in.

1.2 Host Organisation

The research is hosted by and conducted at the IT-consultancy company 'Info Support'. Info Support is a specialist in developing, managing and hosting custom software and BI and integration solutions. Info Support instructors train IT professionals in using the latest software development techniques and methods. Info Support has over 500 employees that help our clients in industries like financial services, government, healthcare, insurance and various industrial sectors [12].

1.3 Contributions

Code metrics for languages using features from other paradigms than they were designed for, is a novel research area. By validating new metrics specific to the functional side of C#, improved detection of error-proneness classes is possible. This is would be another step for observing and assessing software evaluation for the OO language C#. This research covers ground and provides referable knowledge for other researches. This research provides a better understanding of the impact of functional features on software quality in the OO language C#. Info Support will have additional knowledge to reason about software quality and therefore, given an edge to consult other companies.

The framework (described in Chapter 5) used to conduct this research has been published on GitHub. This artifact is open to the research community and can be used to reproduce our results. As the artifact is structured modular, modules can be reused for future research. The output from our analysis framework in *csv*-format is also included in the Github repository. The repository can be found at `https://github.com/bzuilhof/StaticCodeAnalysis`.

1.4 Outline

In Chapter 2 we describe the background of this thesis. In Chapter 3 we describe the problem we are researching, our approach to this problem and our dataset. In Chapter 4 we show what code metrics we are using for our baseline and describe our candidate measures. Chapter 5 discusses and motivates our implementation of the framework that is required for the validation of our candidate measures. Results are shown and discussed with their threats to validity in Chapter 6. Chapter 7, contains the work related to this research. Finally, we present our concluding remarks in Chapter 8 together with future work.

Chapter 2

Background

In this chapter, background information is given to give a deeper understanding of the problem statement. In Section 2.1 we describe the concept of code quality and its impact. In the next Section 2.2, we show how claims can be made about code quality with the help of code metrics. We explain what happens when the OO paradigm uses features from the FP paradigm in Section 2.3. To give an introduction to the problem analysis in Section 3.1, background is given in what kind of FP inspired constructs are introduced to the OO language C# Section 2.4.

2.1 Code Quality

Code quality makes the distinction between bad and good code. However, code quality is a very broad term. The standard ISO25010:2011 [13] describes a model for product quality. The product quality model is build-up from eight characteristics. Namely

- functional suitability
- performance efficiency
- compatibility
- usability
- reliability
- security
- maintainability
- portability

To make statements about these characteristics, models need to be used that help the software engineer what and how to measure. For some characteristics, it can be hard to create such a model. This is mostly because there are too many variable factors involved. Maintainability is defined by the ISO-standard as "The maintainability of a system is a measure of the ability of the system to undergo maintenance or to return to normal operation after a failure." The maintainability characteristic is composed of a set of related sub-characteristics. Namely, modularity, reusability, analysability, modifiability and testability [13]. Heitlager et al. [8] proposed a model for measuring all these sub-characteristics. When aggregating these results, the software engineer is able to make claims about the maintainability of the codebase.

2.2 Code Metrics

Making statements about code quality based on intuition, whether it is good or bad is subjective. Different organizations and teams will have different standards and definitions. To make an objective observation about code quality one can choose for code metrics.
Code metrics provide a standard to measure an internal property of a software system. The application of the metrics enables software engineers to do an objective and reproducible analysis of the codebase. Based on the measurements and empirical studies, the software engineer is able to make estimations on external attributes of a codebase such as error-proneness and maintainability. Heitlager described a model for measuring maintainability based on internal measurable attributes of a codebase [8]. Utilizing this model, a software engineer is able to make a claim about the code quality of a codebase. Because maintainability is a characteristic of code quality [13].

2.2.1 Paradigm Independent Code Metrics

Because volume or number of execution paths through a code section are attributes that every unit of code has, regardless of the paradigm or programming language. Therefore, metrics have been defined these attributes, which can be applied to any section of code.

2.2.2 Object-Oriented Code Metrics

Chidamber et al. defined a metrics suite for OO design [7]. The defined metrics were proposed to help grant insights on how well the OO design principles are followed. These measures were later empirically validated by a study performed by Basili et al. [14]. Basili et al. concluded that five out of six OO metrics that were proposed when added to the set of traditional code metrics, yielded a better prediction for error-proneness. The only metric that showed no significance was 'Lack of Cohesion on Methods'. The study also shows that the metrics appear relatively independent from each other.

2.2.3 Functional Programming Code Metrics

Ryder et al. state that software metrics for FP languages have been neglected identifies a collection of metrics [10] which can be used for the functional programming language Haskell [15]. The study validates the metrics by performing statistical analysis on the correlation between the metric values and the number of bug fixing changes.
Ryder et al. show how a set of metrics from the imperative language can be directly translated to the FP language Haskell, such as Cyclomatic Complexity (CC) and Source Lines of Code (SLOC). However, some features such a pattern matching is not covered by these metrics. Even though, it is hypothesized that this a significant factor for the complexity of the program. Therefore, measures are proposed for attributes of the patterns, such as the size of the pattern or the number of pattern variables that are overwritten.

2.3 Introducing Functional Programming in Object-Oriented Languages

The mainstream object-oriented languages were designed to primarily support imperative programming. Therefore, in traditional OO programming, an imperative style is commonly used. In an imperative style the developer will describe how tasks need to be performed and how changes in states have to be done [16]. Functional programming is a form of declarative programming, by describing program logic instead of a control flow path. The functional programming paradigm has its origins in the lambda calculus. The lambda calculus is a formal system to express computable functions by definition and substitution of variables [17, 18]. These lambda functions are pure by definition. The lambda expressions within the significant OO languages, are introduced as syntactic sugar for writing anonymous functions. However, anonymous functions in OO languages, do not add a constraint regarding purity. Therefore, the purity that is suggested by the term lambda, does not apply for OO languages.

2.4 Functional Programming in C#

Since version 3.0 a set of features that are inspired by functional programming (FP) were added to C#. The FP inspired features that were added, allow less verbose syntax. Microsoft indicates, that this set will only grow with the release of newer versions [19]. In this section we inspect and describe the FP inspired features that were added to C#. Furthermore, we also describe what fp inspired constructs are possible with the introduction of these FP inspired features.

2.4.1 First-Class Functions

A first-class is defined as function is treated as a first-class datatype. With functions being first-class, it is possible to assign functions to variables and pass them around as arguments to other functions [20]. Delegates in C# provide a way to create a function reference, providing a method signature. A delegate initialization results into a function pointer. An example of a delegate initialization is shown in Listing 1.

```
1 delegate int ExampleDelegate(int i);
2 int AddOne(int i) => i + 1;
3
4 void Foo()
5 {
6     ExampleDelegate x = AddOne;
7 }
```

Listing 1: Delegate in C#

C# version 3.0 introduces a built-in generic delegate, which can define the method signature inline. This way a beforehand delegate definition is not nescessary, because as we can use the generic delegate. In Listing 2 an example usage of the generic delegate is shown.

```
1 int AddOne(int i) => i + 1;
2
3 void Foo()
4 {
5     Func<int, int> x = AddOne;
6 }
```

Listing 2: Generic delegate in C#

What Problem Does It Solve?

With first-class functions, we can assign behavior to variables. This gives a concise way of naming behavior. Using an identifier, the behavior can be later referenced or executed.

2.4.2 Higher-Order functions

Higher-order functions are functions that take one or more functions as a parameter or have a function as a return type [21]. Because functions are first-class datatypes, they can now be used as a return type or as a parameter. Therefore, first-class functions enable higher-order functions.

In Listing 3 we shown an example of a higher-order function.

```
1 int DoMathOperation(Func<int, int, int> f, int i1, int i2)
2 {
3     return f(i1, i2);
4 }
5
6 Func<int, int, int> multiplication = (i1, i2) => i1 * i2;
7 int a = DoMathOperation(multiplication, 3, 2);
```

Listing 3: Higher-order functions in C#

The function `DoMathOperation()` expects a function as a first parameter, a function that maps two integers to one integer. This requirement is defined in the method signature on `line 6`. `DoMathOperation()` matches the method signature; therefore, we are able to use the function as an argument to the higher-order function `line 7`.

What Problem Does It Solve?

Higher-order functions provide an abstraction over functions. This way, the behavior of the function can change depending on its arguments. This makes reuse of often-used functionality, such as looping over collections easy.

2.4.3 Lambda Functions

Delegates provide a way in C# to have pointers to functions, in which the delegate defines the method signature as described in Section 2.4.1. C# version 2.0 allows declaration of an anonymous delegate in-line. In C# version 3.0 lambda expression can be used to initialize an anonymous delegate with a different syntax. The syntax of the lambda function is shown in Listing 4.

```
1  delegate int ExampleDelegate(int i);
2  static int AddOne(int i) => i + 1;
3
4  static void Main(string[] args)
5  {
6      // Original delegate initialization syntax with a named method.
7      ExampleDelegate a = AddOne;
8      // Delegate initialization with an inline anonymous method.
9      // This syntax was introduced with C# version 2.0.
10     ExampleDelegate b = delegate(int i) { return i + 1; };
11
12     // C# version 3.0 introduces lambda expressions.
13     ExampleDelegate c = i => i + 1;
14 }
```

Listing 4: Delegates and lambda expressions in C#

These anonymous delegates and lambda expressions have access to all objects, methods and variables, that any expression would have in that closure.

Expression trees

The only difference, aside from syntax syntax, between lambda expressions and anonymous delegates is that lambda expressions can be used to define expression trees. Expression trees are a way of describing **what** a piece of code should do instead of **how**. Expression trees are not executable but are a data structure. These expression trees are commonly used with an external data source. E.g. if the goal is to retrieve data from an external data storage, these expressions trees can be used to create a query. Expression trees can be seen as functions, intended to be passed as an argument. This concept design heavily relies on the concepts of first-class functions and higher-order functions.

What Problem Does It Solve?

Describing an anonymous delegate uses a lot of boilerplate code, with lambda expressions this boilerplate code is reduced to a minimum. The difference in boilerplate code can be seen when comparing `line 10` vs `line 13` in Listing 4.

2.4.4 Lazy Evaluation

The concept of lazy evaluation, which comes along with the LINQ[1] query syntax was previously only possible by using the `Lazy<T>`-keyword.

[1] Language Integrated Query, a uniform query syntax in C# to retrieve data from different sources [22]

Lazy Initialization

When initializing an object with the `Lazy<T>`-wrapper class, it is possible to defer the execution of the constructor until the first usage of the object. This way, the software developer is able to avoid unnecessary memory usage and save computation power [23]. An example of this usage is shown in Listing 5.

```
1 // User object is not yet created at this point
2 Lazy<User> user = new Lazy<User>();
3
4 // User object is created and the age is requested from the object
5 int age = user.GetAge()
```

Listing 5: Example of Lazy<T>usage in C#

The object is initialized on `line 2`. But because the lazy wrapper is wrapping the class, the object will not be created and the constructor will not be executed until a class method or a class variable is used. On `line 5`, we need a class method to evaluate the expression. Before this class method is executed, the creation and execution of the constructor need to happen. At this point, the object will be created.

Language Integrated Query (LINQ)

LINQ introduced syntax for list operations like the ones known from functional languages. These include the higher-order functions such as `map`, `filter` and `sort`. This syntax enables list mutations with a concise syntax in C# as shown in Listing 6.

```
1 Enumerable.Range(1, 10)
2     .OrderBy(i => -i); //sort
3     .Where(i => i % 2 == 0) //filter
4     .Select(i => i * 10) //map
5 // ["100", "80", "60", "40", "20"]
```

Listing 6: C# With LINQ

The statement on `line 1` creates a enumerable which would evaluate into a list with numbers ranging from 1 to 10. On `line 2` we used the extension method `OrderBy` to order the list. The `OrderBy` method expects a compare function as an argument. The lambda expression given as an argument to the method, causes the list to be ordered in descending order. The `Where` method filters the list by with a function that can be applied to every element of the list. The return type of this function has to be a `boolean`. The lambda expression that is given as an argument to the filter function, will remove all uneven elements from the list. The `Select` method is used to transform each element of the list. The argument given will determine how each element is transformed. In the example a lambda expression is given as an argument that will multiply each element in the list by 10.

If the expression in Listing 6 is assigned to a variable, the variable contains a pointer to an enumerable with a query attached to it. Meaning the query has not been executed yet. These LINQ-queries (in the context of object-querying) are only evaluated when a terminating extension method is used, such as: `ToList()`, `ToArray()`, `First()`, `Count()`, etc. which demand a list, object or an integer as a return type.

What Problem Does It Solve?

Lazy evaluation has the benefit that execution or initialization can be postponed until the object or the result of the evaluated expression is needed. This way expression evaluations or object initializations that are not necessary for the functionality of the code, can be avoided.

2.4.5 Pattern Matching

In C# version 7.0 [24] a new syntax using the `when`-keyword was introduced. Using this keyword it is possible to do a *switch-case*-statement on multiple variables. This keyword allows the usage of multiple conditions. Previously this was only possible using an **If/Else-If**-statement with multiple conditions. An example of the usage of the `when`-keyword is given in Listing 7.

```
static State ChangeState(State current, Transition transition, bool hasKey)
{
    switch (current)
    {
        case Opened when transition == Close:
            return Closed;
        case Closed when transition == Open:
            return Opened;
        case Closed when transition == Lock && hasKey == true:
            return Locked;
        case Locked when transition == Unlock && hasKey == true:
            return Closed;
        default:
            throw new InvalidOperationException("Invalid transition");
    }
}
```

Listing 7: Multi-variable switch-case in C#

In C# version 8.0 (to be released) a more declarative syntax is introduced for pattern matching [19]. The syntax is more focused on **what** has to be compared instead of by defining a control flow (**how**). The same **Switch-Case**-statement shown in Listing 7, is shown with the new more declarative syntax in Listing 8.

```
static State ChangeState(State current, Transition transition, bool hasKey) =>
(current, transition, hasKey) switch
{
    (Opened, Close,  _)    => Closed,
    (Closed, Open,   _)    => Opened,
    (Closed, Lock,   true) => Locked,
    (Locked, Unlock, true) => Closed,
    _ => throw new InvalidOperationException("Invalid transition")
};
```

Listing 8: Pattern matching in C# 8.0

Comparing the used syntax in Listing 8 to the syntax that the can be used in the functional programming language Haskell to implement the same logic, the gap in syntax style seems to shrink. The implementation in Haskell is shown in Listing 9.

```haskell
changeState :: State -> Transition -> Bool -> State
changeState Opened Close _      = Closed
changeState Closed Open _       = Opened
changeState Closed Lock true    = Locked
changeState Locked Unlock true  = Closed
changeState state _ _           = state
```

Listing 9: Pattern matching in Haskell

What Problem Does It Solve?

The more declarative syntax for pattern matching gives a more concise syntax. Even in Listing 7 a lot of boilerplate code is needed to implement the **Switch-Case**-statement. The switch is performed on 1 out of 3 variables, the other 2 variables are still accessible because the switch-case closure is not enclosing. So the outside closure can affect the results of **Switch-Case**-statement. This impure statement can be avoided by using the pattern matching syntax that is introduced in version 8.0 as shown in Listing 8. Purity has several advantages such as better maintainability and readability [16].

Chapter 3

Research Methodology

The new FP inspired constructs as described in Section 2.4, are unknown in both the FP and the OO paradigm. Therefore, these new constructs also introduce a new problem space and a new kind of complexity. Commonly used code metrics are unfit to cover this new complexity. We describe this new problem space in Section 3.1. In Section 3.2 we give based on the problem analysis, our hypothesis and our approach to addressing the problem. We approach the problem by defining measures that cover this new kind of complexity. We present the methodology that we are using for validating our measures in Section 3.3. We introduce and motivate our choice of data analysis technique for validating the measures in Section 3.4. To show the significance of our measures, we must define a baseline for the research. In Section 3.5 we describe this baseline. Making an improvement on the baseline model can be done by substituting baseline metrics with our candidate measures. The result of this substitution is a new error-proneness prediction model. In Section 3.5.1 we describe how we evaluate and validate this prediction model.

3.1 Problem Analysis

The introduction of lambda functions in OO languages, such as C#, enables the developer to use functional programming features within OO languages. The compilers for functional languages as Haskell enforce purity [25]. A function is pure if it satisfies the constraints shown below [26].

- *It has no side effects, that is to say, invoking it produces no observable effect other than the result it returns; it cannot also e.g. write to disk or print to a screen.*
- *It does not depend on anything other than its parameters, so when invoked in a different context or at a different time with the same arguments, it will produce the same result.*

This enforcement does not happen with lambda functions for object-oriented languages. Even though lambda functions, that originate from the lambda calculus [17] are designed to be pure. Lambda functions do not have to be pure in object-oriented languages, which can lead to unexpected results. As shown in Listing 10, a lambda function is allowed to have a reference to a mutable class variable. This can result in an invocation of the lambda function with the same parameters, but a different return value (which makes the function impure), therefore, it can lead to unexpected results.

```
class ClassA
{
    private int _x = 2;
    public void PrintDoubles(int ub)
    {
        IEnumerable<int> multipliedNumbers = Enumerable.Range(0, ub + 1)
            .Select(i => i * _x);
    }
}
```

Listing 10: Lambda function refering class variable

Furthermore, the developer able to use variables from outside of the lambda function its closure. Moreover, the developer is also able to mutate the state of the class from within the lambda function, as shown in Listing 11. This is known as a side effect which implies impurity.

```
class ClassA
{
    private int _x = 2;
    public void PrintDoubles(int ub)
    {
        IEnumerable<int> multipliedNumbers = Enumerable.Range(0, ub + 1).Select(i =>
        {
            _x++;
            return i * _x;
        });
        foreach (var number in multipliedNumbers) Console.WriteLine(number);
    }
}
```

Listing 11: Lambda function mutating class variable

The FP inspired constructs shown above, are implemented in C# with the modern IDE Rider. Rider uses the Roslyn as IntelliSense as described in Section 5.2.1. Roslyn provides IntelliSense in the Visual Studio IDE as well. Neither the C#-linter nor the compiler gives a warning about these dangerous constructs. The impure usage of lambda functions can have unexpected results. These unexpected results can be caused by the state. A state is non-existent in the functional paradigm. The state can influences the result of the lambda function.

Besides the immunity to bugs related to mutable states, purity has several other advantages [27]:

- **Testability** Because no state influences the outcome of the function, there is no needed for a mocked stated which makes testing easier.
- **Parallelization** Because the same input will always have the same output it does not matter in what order the function is called. Therefore, the computation can run on the same iterable on multiple threads at the same time.
- **Memoization** If a function is called twice with the same parameters. The cached result from the first function call is returned on the second call. This is possible because nothing else besides the given parameters can influence the result of the function.
- **Lazy evaluation** Because nothing besides the given parameters influences the result of the function, it does not matter when the function is evaluated. Therefore, the evaluation can be postponed.

3.1.1 Hidden Complexity

In the following code snippets, two implementations are presented which have the functionality to get a list with vehicles that start with 'Red'. For the first implementation (Listing 12) an imperative approach is chosen. The snippet has a Source Lines of Code (SLOC) count of 11 and a Cyclomatic Complexity (CC) of 3 since there are two branching points. This is how the general complexity of the snippet translates back to the values for code metrics.

```
List<string> vehicles = new List<string>() {"Red Car", "Red Plane", "Blue Car"};

List<string> redVehicles = new List<string>();
for (int i = 0; i < vehicles.Count; i++)
{
    if (vehicles[i].StartsWith("Red"))
    {
        redVehicles.Add(vehicles[i]);
    }
}
```

Listing 12: Traditional implementation

In the second implementation show in Listing 13, the LINQ library is used. The LINQ library encourages the use of lambda expressions because only a small part of the behavior of these higher-order functions need to be defined by the developer. The snippet has a SLOC count of 5 and a CC of 1 since there are no branching points. Even though the functionality and the logical complexity of both implementations are the same, the CC and SLOC differ drastically.

```
List<string> vehicles = new List<string>(){"Red Car", "Red Plane", "Blue Car"};

List<string> redVehicles = vehicles
    .Where(t => t.StartsWith("Red"))
    .ToList();
```

Listing 13: LINQ implementation

3.2 Problem Approach

We hypothesize that constructs that occur shown in Section 3.1, induce error-proneness in their corresponding class. By defining measures and exploring the relationship of these measures to error-proneness, we can make statements about the above-stated hypothesis. The measures cover the new problem space. The problem space that is created by introducing FP inspired features in the object-oriented language C#.

3.3 Metric Validation Approach

Briand et al. [1] describes a methodology for empirical validation of metrics. To validate the metric, three assumptions that should be satisfied. The assumptions are listed below.

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

Assumption 1 The hypothesized relationship between attribute A_1 and A_2 can be tested if Assumption 1 and Assumption 2 are assumed, by proving a relationship between X_1 and X_2.
Projecting this on our research, we can test the relationship between error-proneness and functional constructs, by proving a relationship between bug fixes and values for our candidate measures.

Assumption 2 Measure X_1 will measure defined attributes of the code such as mutable external variables used in lambda functions. This measure X_1 will be assumed to measure on A_1, A_1 will the internal attribute such as purity of the lambda usages.

Assumption 3 Measure X_2 will measure the error-proneness A_2 of a given class. The measure X_2 will be the boolean if, during the lifetime of the project, a bug was fixed in the concerning class.

3.4 Relating Functional Constructs to Error-Proneness

Investigating the relationship between code metrics to error-proneness is commonly done by creating a prediction model. The prediction model will predict error-proneness based on code metrics [1, 14, 28, 29].

We create a prediction model using the logistic regression classification technique, this technique can be used to predict a variable that is dichotomous. In our case, the dichotomous variable will be the error-proneness of classes.

Logistic regression [30] is often used to create such a prediction model [5, 14, 28, 31].

With a logistic regression model trained with the data from our analysis framework, which processes repositories, we explore the relationship between our measured constructs and error-proneness.

Univariate

With a univariate logistic regression model, we can evaluate the performance of the prediction model for error-proneness in isolation. The independent variable in the prediction model will be the values of our measured constructs. Using Equation 3.1 we construct a prediction model.

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

In the above equation, $\beta_l X_l$ is the coefficient multiplied with the value of the candidate measure.

Multivariate

Furthermore, we will be looking into multivariate logistic regression to test if we can improve the prediction model with the in-place OO metrics. The baseline for the evaluation of the model will be a multivariate logistic regression model based on our baseline set of metrics. To see if we can achieve an increased performance compared to our baseline model, we substitute the baseline dependent variables with candidate measures as l. We construct the predicition model using Equation 3.2.

$$P(faulty = 1) = \frac{e^{\beta_0 + \beta_1 X_i + \beta_2 X_2 + ... + \beta_n X_n + \beta_l X_l}}{1 + e^{\beta_0 + \beta_1 X_i + \beta_2 X_2 + ... + \beta_n X_n + \beta_l X_l}} \tag{3.2}$$

3.5 Baseline

To show that our measures are useful regarding error-proneness prediction, their inclusion must yield better results than metrics that are being used in the industry. This set of metrics will define the baseline for this study. We use the union of a set of general code metrics as described in Section 4.1.1 with a set of object-oriented metrics as described in Section 4.1.2.

3.5.1 Model Validation

We choose to validate our model using cross-validation, which is commonly used for the validation of prediction models [32, 33]. We use the Holdout method for cross-validation. By default, holdout cross-validation separates the data set into a train set and a smaller test set. To compensate for the randomness of the division, we run the model fitting with multiple different selections of the training set and average the

results and assess the standard deviation. Based on a classification report created for with the hold-out set, we assess the performance of the model. We use the F_1-score to assess our model performance. The F_1-score calculates the harmonic mean of the precision and recall, shown in Equation 3.3.

$$F_1 = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{3.3}$$

Our dataset is unbalanced, as seen in Table 6.3. Accordingly, one could choose to calculate the micro-average between the F_1-scores for the 'faulty-classes'-class and the 'non-faulty-classes'-class, in which the support for each class is weighted. On the contrary, since we want good prediction performance in both classes we use the macro-average instead which calculates the harmonic mean between the two F_1-scores [34].

Chapter 4

Code Metrics

In Section 3.5 we described the purpose and choice for a baseline. In Section 4.1 we describe the set of all of the baseline metrics. We also discuss and motivate our interpretation of the metrics. In Section 4.2 we propose candidate measures for this study.

4.1 Baseline Metrics

For each of the metrics, we describe how we implemented the metrics in our 'Code Analysis Framework', as described in Section 5.2.

4.1.1 General Metrics

For general code metrics we take the following metrics:

Source Lines of Code *(SLOC)*
The SLOC measure gives an indication on the size of a class. It is arguable what counts as source line of code and what does not (e.g. do we count lines with only a curly bracket as code?). For our definition, we follow the standard described by Nguyen [35]. In this standard, we exclude the lines without code, as in lines with only comments or only white space characters.

Cyclomatic Complexity (*CC*)
Cyclomatic Complexity, also known as McCabe Complexity, is a standard of measure that counts the linearly independent execution paths through a class. For our definition, we also consider additional conditions in e.g. an *If*-statement as an extra path. This is because C# uses short-circuiting [36]. Short-circuiting meaning, for the statement: `A && B`, `B` is only evaluated if `A` evaluates to `True`. Therefore, there are two execution paths through this statement.

Comment Density *(CD)*
Comment Density can be used as a quality indicator for software [37, 38]. We calculate the density using the definition as described by SonarQube [37]. Using Equation 4.1, the measured value will result into 1 if all the lines in a class are comments and 0 if none of the lines in the class are comments.

$$CD = \frac{CommentLines}{CommentLines + SLOC} \tag{4.1}$$

In which comment lines are all lines that contain a comment. Therefore, an empty line within a multi-line comment is excluded from the *CommentLines*.

4.1.2 Object-Oriented Metrics

The object-oriented metric suite that we use for this study was defined by Chidamber [7]. For our baseline, we implemented the metrics which showed any significance in the study.

Weighted Methods per Class *(WMC)*
The weighted methods per class can be calculated with Equation 4.2. Where $m_1 ... m_n$ are the methods in class c and $C()$ calculates the complexity of a method.

$$WMC_c = \sum_{i=1}^{n} C(m_i) \tag{4.2}$$

The metric definition does not define how the complexity should be computed, but suggests McCabe's Cyclomatic number may be appropriate [7]. We follow up on this suggestion.

Depth of Inheritance Tree (DIT)
The depth of inheritance tree is the distance in inheritance to the root class. The value for this metric is calculated by walking up the inheritance tree until the root node is encountered. The implication is here: The higher number of depth implies more reuse through inheritance and therefore, higher complexity.

Response For a Class (RFC)
Response for a class is defined as the size response set for a class. The response set for a class is calculated by the definition in Equation 4.3.

$$RFC = |M \cup \{\bigcup_{i=1}^{n} R_i | i \in M\}| \tag{4.3}$$

The set of methods of the corresponding class are represented by M. R_i are all the function invocations in method i. The definition for RFC states that methods refers to methods available to the object. Since lambda functions are not available directly to the object, they are excluded for calculating the cardinality.

Number of Children of a Class (NOC)
Number of Children of a Class is defined as the number of direct children in the inheritance tree. This measure is computed by counting the classes that are direct subclasses of the concerning class.

Coupling between Object Classes (CBO)
Coupling in the context of this measure is defined as usage of a method or instance variable of another class. The number of coupling is indicated by the number of usages of such methods and variables. The implication is that this is an anti-pattern to encapsulation, which is one of the four basic principles of object-oriented programming [39].

Lack of Cohesion of Methods (LCOM)
The Lack of Cohesion of Methods measure is used to measure the cohesion within a single class. This is done by subtracting the number of methods that do not share an instance variable by the number of methods that do share an instance variable. The definition for this measure is shown in Equation 4.4.

$$\begin{aligned} P &= \{(I_i, I_j | I_i \cap I_j = \emptyset\} \\ Q &= \{(I_i, I_j | I_i \cap I_j \neq \emptyset\} \\ LCOM &= |P| - |Q| \end{aligned} \tag{4.4}$$

Where the class contains n methods $M_1, M_2 ... M_n$ and I_i is the set of instance variables used by method M_i.

The original definition as given by Chidamber [7] makes no statement on what defines a method. We make the assumption that methods refer to class methods. This assumption is based on that cohesion between class methods is desirable, since it promotes encapsulation. Which is one of the four object-oriented programming basic principles [39]. Therefore, anonymous methods and lambda functions are excluded from the set of methods.

4.2 Candidate Measures

For this study, we defined a set of candidate measures. For each of the following measures, we explore the relationship to error-proneness.

Number Of Lambda Functions Used In A Class (*LC*)
Lambda functions in the context of OO languages are a concise way to write anonymous functions inline. Compared to a regular method, the parameter type, the return type can all be omitted. This might introduce constructs which are harder to understand. An example of this scenario is given in Listing 14. To calculate the value for this measure, we traverse the abstract syntax tree (AST). For each *SyntaxNode* with type *LambdaExpression*, we raise the counter for this measure.

```
List<int> numbers = new List<int>() { 1, 2, 3 };
IEnumerable biggerThan2 = numbers
    .Where(x => x > 2);
```

Listing 14: Lambda expression

Source Lines of Lambda (*SLOL*)
Simple lambda expressions might not be extra information to reason about the execution. Once the lambda expressions become more complex this might not be the case. In listing 15, an example is given with a multiline lambda expression. As curly braces are taken included in the 'source lines of code'-measure [8], we also include these curly braces when calculating the span of the lambda expression. Therefore, the snippet in Listing 15 has an 'Source Lines of Lambda'-count of $1 + 1 + 4 = 6$.

```
private IEnumerable<int> bla = Enumerable.Range(1, 10)
    .Select(i => i * 10)
    .Where(i => i % 2 == 0)
    .OrderBy(i =>
        {
            return -i;
        });
```

Listing 15: Lambda expression spanning multiple lines

Lambda Score (LSc)
The density of the usage of lambda functions in a class can give an indication of how functional a class is. Our hypothesis for this measure is, that a relationship exists between how functional a class is and the error-proneness of the class. We calculate this lambda density with Equation 4.5.

$$LSc = \frac{SLOL}{SLOC} \tag{4.5}$$

Evaluating into 1 if each line of the classes is spanned by a lambda expression, 0 if none of the lines are spanned by a lambda expression.

Number Of Lambda Functions Using Mutable Field Variables In A Class (*LMFV*)
Sometimes it might be hard to predict when a lambda function is executed, therefore, it might be hard to reason about what value for the mutable field variable will be used. An example of this scenario is given in Listing 16.
To calculate the value for this measure, we traverse the AST. For each variable inside a lambda expression, we check if the variable is non-constant and field scoped by using the semantic data model (SDM) of the class. If this test passes, we increase the counter for this measure.

```
class A
{
    int _y = 2;
    void F()
    {
        Func<int, bool> biggerThanY = x => x > _y;
    }
}
```

Listing 16: Lambda expression with reference to mutable field variable

Number Of Lambda Functions Using Mutable Local Variables In A Class (*LMLV*)
Sometimes it might be hard to predict when a lambda function is executed, therefore, it might be hard to reason about what value for the mutable local variable will be used. An example of this scenario is given in Listing 17.
To calculate the value for this measure, we traverse the AST. For each variable inside a lambda expression, we check if the variable is non-constant and local scoped by using the SDM of the class. If this test passes, we increase the counter for this measure.

```
void F()
{
    int y = 2;
    Func<int, bool> greaterThanY = x => x > y;
}
```

Listing 17: Lambda expression with reference to mutable local variable

Number Of Lambda Functions With Side Effects Used In A Class (*LSE*)
We think that the combination of side effects in lambda functions with e.g. parallelization or lazy evaluation is dangerous because it can be hard to reason about when these side effects occur. An example of this scenario is given in Listing 18.
To calculate the value for this measure, we traverse for each class its AST. For each variable inside a lambda expression, we check if local or field variables are being mutated.

```
static int _y = 2;

Func<int, bool> f = x =>
{
    _y++;
    return x > _y;
};
```

Listing 18: Lambda expression with side effect to mutable field variable

Number Of Unterminated Collection Queries In A Class (*UTQ*)
By not terminating a collection query, it will be hard to reason when the query will be executed. These collection queries may contain functions that contain side effects and use outside scope variables. Therefore, the execution at different run-times can yield different and unexpected results. An example of this scenario is given in Listing 19.
To calculate the value for this measure we traverse the AST and count how many `IEnumarable<T>` are initiated.

```
List<int> nmbs = new List<int>() { 1, 2, 3 };
int y = 2;
IEnumerable biggerThanY = numbers
    .Where(x => x > y);
```

Listing 19: Unterminated LINQ-query

Chapter 5

Project Analysis Framework

To validate our candidate measures from Section 4.2 additional tooling and data are required. Our approach is to validate our measures on a set of relevant projects. In Section 5.1 we describe what makes a project relevant and what constraints the projects need to satisfy. To calculate values for our baseline and candidate measures we implement a static code analysis framework, described in Section 5.2. In Section 5.3 we describe the implementation of our candidate measures. We present our approach to determine whether the classes analyzed are error-prone in Section 5.4. In Section 5.5 we describe our policy for dealing with the mismatch between the classes static analyzed and the classes marked as error-prone.

5.1 Data Collection

5.1.1 Criteria

The following requirements have to be satisfied for each project to be usable in our data set. The data set which is needed for the validation of our metrics.

C# classes For this research it C# is chosen. Therefore, it is essential that the project consists of a significant amount of C# classes (100+). Otherwise, on a small set of classes, results could give an incorrect indication caused by randomness.

Issue tracker For measuring error-proneness, we need to be able to mark classes error-prone and not error-prone as described in Section 5.4. To be able to do this we need to link known bug-issues to commits. Therefore, we need an issue tracker in which links are made from bug-issues to bug-fixing commits.

Semi-balanced data set If the results from the error-proneness marking are very unbalanced (¡1% faulty), these are unsuitable for validating our proposed metrics. As the results could give wrong indications because of randomness.

5.1.2 Data sets

To make claims about the generalizability of our results, a data set is needed that represents as much projects as possible. We made the distinction between open and closed source.

Open source projects Within the open source projects, we choose to have two subsets. One set with projects from the .NET Foundation. The .NET Foundation has over 70 public C# projects. We assume that these projects are more strictly managed than projects without such a big organization in the managing role. This might influence the results. Moreover, we also have a set of what we call 'community managed projects. For most of the projects, GitHub's issue tracker is used. The issue tracker provides a way of linking bugs to commits. This can be done by looking at the commit message.

Closed source-projects 'Info Support' has provided a project 'KnowNow' which is a knowledge-sharing platform. The repository is located in VSTS (Visual Studio Team Services). In VSTS all issues are logged, the issues that were created regarding a bug received the 'bug' label. VSTS automatically migrated to Azure DevOps in September 2018 [40]. Azure DevOps Services REST API offers endpoints

where it is possible to request all the issues that are labeled bug and all the commits with their metadata [41]. Because the developers who worked on the project kept strict policies on the Git commit message, it is possible to link the commits back to the issues.

5.2 Static Code Analysis

5.2.1 'Roslyn' .NET Compiler Platform SDK

The .NET Foundation provides an open-source compiler platform named 'Roslyn'. Roslyn enables rich code analysis for the C# and Visual Basic .NET languages [42]. After the solution is loaded into a build workspace, it is possible to request the compilation unit for each of the projects. The compilation unit contains the abstract syntax tree (AST) for each of the files in the projects. For each of these AST a semantic data model (SDM) can be requested from the compilation unit [43]. This SDM contains all semantic data from a file. It can be compared to the data that is provided by IntelliSense, such as what the type of a variable is or where it was initialized.

Roslyn is the default compiler in the Microsoft provided IDE Visual Studio. Roslyn also provides the IntelliSense features within the IDE.

5.2.2 Implementation

By loading a solution file into the compiler, an AST is created from each C# file in the solution. Calculating the value for the measure of Coupling between Objects (cbo) requires knowledge from other classes. Namely how many classes instantiate the concerning class, or use the instance variables of functions of the concerning class. To calculate this value some kind of map is needed where we can see how many references there are to the concerning class. Therefore, an extra iteration upfront is needed to construct such a map. During this extra iteration, we also construct an inheritance map, where we keep track of how many subclasses each class has. This data is needed for the calculation of the Number of Children (NOC) measure. This way we have enough data to calculate the values for all of our baseline and candidate measures during the traversal of the solution. By building the project and by traversing the AST in combination with the available semantic model we are able to compute the values from most of our implemented metrics. For the metrics of Coupling between Objects and Number of Children, traversing the AST with the available semantic model does not suffice. Therefore, we do an extra iteration is done to create a map of what classes derives from what classes.

Algorithm 1: Calculating the measure values for each class

```
Result: Classes with measures values
Data: Solution
ClassResults ← [];
InheritanceMap ← [];
ObjectCouplingMap ← [];
Projects ← Solution.projects;
for p ∈ Projects do
    ASTs ← GetASTs(project) for AST ∈ ASTs do
        for class ∈ AST.classes do
            InheritanceMap.Include(class);
            ObjectCouplingMap.Include(class);
        end
    end
end
for p ∈ Projects do
    ASTs ← GetASTs(project) for AST ∈ ASTs do
        for class ∈ AST.classes do
            Classes += CalculateMeasureValues(class, InheritanceMap, ObjectCouplingMap);
        end
    end
end
```

5.3 Candidate Measures Implementation

For the candidate measures as described in Section 4.2 we show and discuss the implementation.

5.3.1 Lambda Measures

As all candidate measures count constructs regarding the usage of lambda expressions, excluding the UTQ measure, we compute these values in a single AST traversal.

Algorithm 2: Counting Lambda Usages

```
Data: ClassNode
Result: Values for Lambda Measures
LambdaMeasures ← Object;
for n ∈ ClassNode.DescendantNodes do
    if n.Type == LambdaExpression then
        LambdaMeasures.LambdaCount++;
        if n.usesLocalVariable then
            LambdaMeasures.LocalVariableUsed++;
            if n.updatesLocalVariable then
                LambdaMeasures.SideEffect++;
            end
        end
        else if n.usesFieldVariable then
            LambdaMeasures.FieldVariableUsed++;
            if n.updatesLocalVariable then
                LambdaMeasures.SideEffect++;
            end
        end
    end
end
```

5.3.2 Unterminated Collection Queries

Unterminated collection queries can be identified by looking variable declarations where the identifier type is an `IEnumerable` (Systems.Collections) or `IEnumerable<T>` (System.Linq). Which come from different namespaces, as shown above. If a terminating statement is executed on these, the queries will be evaluated and it will return a result. Example terminating statements are `Count()` which returns an integer, `ToList()` which returns a list or `first()` which return an object of the generic type of the collection.

Algorithm 3: Counting unterminated collection queries in a class

```
Data: ClassNode
Result: Unterminated collection queries
UtqCount ← 0;
for n ∈ ClassNode.DescendantNodes do
    if n.Type == VariableDeclaration and n.FirstChild.SymbolType == "IEnumerable" then
        UtqCount++
    end
end
```

5.4 Measuring Error-Proneness

To make an estimation on how error-prone a class is, we make the assumption that if a class during the lifetime of a project was updated by a bug fix, it is error-prone. Unfortunately, the GitHub API does not provide an easy way to identify bug-fixing commits. From a GitHub repository, we can request all the issues that were created regarding a bug. Then we identify all commits that close an issue by searching

for issue closing keywords as described by GitHub [44]. Then all commits that mention an issue that was identified as a bug related issue, are marked as bug-fix commits. In Listing 4 pseudo-code is given for the above-described algorithm.

Algorithm 4: Identifying bug-fix commits

```
Result: Bugfix commits
Commits ← getAllCommits();
BugIssues ← getAllBugIssues();
BugFixCommits ← [];
for c ∈ Commits do
    if ClosesIssue(c.message) and BugIssues.Contains(c.message.issue) then
        BugFixCommits += c;
    end
end
```

We then extract the affected lines from the metadata of the commit. Then derive with which classes the affected lines intersect in the parent version of the bug-fix commit. This is done by parsing the AST for the updated file. We use the parent version of the bug-fix commit since this is the version where the bug existed. Each of these intersected classes will be marked as error-prone.

Algorithm 5: Identifying faulty classes

```
Data: Bugfix commits
Result: Faulty classes
FaultyClasses ← [];
for c ∈ BugFixCommits do
    for f ∈ c.PatchData.ModifiedFiles do
        AST ← ParseText(f);
        Classes ← GetClasses(AST) ;
        AffectedLines← CalculateAffectedLines(f);
        AffectedClasses ← AffectedLines ∩ Classes;
        FaultyClasses ← FaultyClasses ∪ AffectedClasses;
    end
end
```

5.5 Combining Results

Not all the classes marked as faulty exists on the same location as when the bug was reported. There seems to be a significant mismatch. This mismatch can be explained by multiple reasons.

- **The class was renamed.** To deal with this one could do a similarity analysis between every class. However, dealing with this problem is too complex to include in this research.
- **The class was merged into another class.** Because a merged class will have different functionalities and characteristics than the original class, it is not reasonable to treat it as the original class.
- **The class split up in multiple classes.** A split-up class is for the same reasons as above, not reasonable to consider the same as the original class.
- **The class was moved to another location.** Only the parent folder is changed for this class. The file name and class name are kept intact. Therefore, it is possible to deal with this (partly) reason.

Our approach to deal with this mismatches only partly solves the last reason for the mismatch, namely moved classes. Matching a class purely on name and project will result in too many incorrect matches. Matching on the absolute file path would result in too many mismatches. So our approach is to include the hash value of the filename together parent folder as metadata. This way we are able to deal with directory structure refactoring, which would otherwise result in a big set of mismatches. In Listing 6, we describe our algorithm in pseudo-code.

Algorithm 6: Matching Analyzed with Faulty Classes

```
Data: FaultyClasses
AnalyzedClasses
Result: Faulty Analyzed Classes
ResultClasses ← AnalyzedClasses;
for c ∈ FaultyClasses do
    if ResultClasses.Contains(c.hash) then
        ResultClasses[c.hash].faulty = true
    end
end
```

Chapter 6

Validation

In Section 6.1, we present an overview with all projects that were analyzed in this research. To investigate how disjunctive our candidate measures are we did a correlation analysis. This analysis is described in Section 6.2. We present the performance of our candidate measures as error-proneness predictors in Section 6.3. In Section 6.4, we discuss the threats to validity to our results.

6.1 Analysed projects

For this study, we analyzed the following projects. For each of these projects, we provide the commit hash of the version we analyzed in our framework. In this section, we will continue to refer to the projects by the abbreviation that is provided. The projects presented, satisfy the constraints defined for the dataset provided in Section 5.1.1.

6.1.1 .NET Foundation Managed Projects (Open Source)

- **CLI** The .NET Core command-line interface (CLI) is a new cross-platform toolchain for developing .NET applications. The CLI is a foundation upon which higher-level tools, such as Integrated Development Environments (IDEs), editors, and build orchestrators, can rest [45]. *Analyzed version: bf26e7976*
- **ML** Machine Learning for .NET is a cross-platform open-source machine learning framework which makes machine learning accessible to .NET developers. ML.NET allows .NET developers to develop their own models and infuse custom machine learning into their applications, using .NET, even without prior expertise in developing or tuning machine learning models [46]. *Analyzed version: b8d1b501*
- **IS4** IdentityServer is a free, open-source OpenID Connect and OAuth 2.0 framework for ASP.NET Core [47]. *Analyzed version: da143532*
- **ASP** ASP.NET Core is an open-source and cross-platform framework for building modern cloud-based internet connected applications, such as web apps, IoT apps and mobile backends. ASP.NET Core apps can run on .NET Core or on the full .NET Framework [48]. *Analyzed version: 5af8e170bc*
- **EF** EF Core is an object-relational mapper (O/RM) that enables .NET developers to work with a database using .NET objects. It eliminates the need for most of the data-access code that developers usually need to write [49]. *Analyzed version: 5df258248*

6.1.2 Community Managed Projects (Open Source)

- **AKK** Akka.NET is a community-driven port of the popular Java/Scala framework Akka to .NET. Akka is a toolkit for building highly concurrent, distributed, and resilient message-driven applications. Akka is the implementation of the Actor Model. [50]. *Analyzed version: bc5cc65a3*
- **JF** Jellyfin is a Free Software Media System that puts you in control of managing and streaming your media [51]. *Analyzed version: d7aaa1489*
- **ORA** OpenRA is an Open Source real-time strategy game engine for early Westwood games such as Command & Conquer: Red Alert written in C# using SDL and OpenGL [52]. *Analyzed version: 27cfa9b1f*

- **DNS** dnSpy is a debugger and .NET assembly editor. You can use it to edit and debug assemblies even if you don't have any source code available [53]. *Analyzed version: 3728fad9d*
- **ILS** ILSpy is the open-source .NET assembly browser and decompiler. [54] *Analyzed version: 72c7e4e8*
- **HUM** Humanizer meets all your .NET needs for manipulating and displaying strings, enums, dates, times, timespans, numbers and quantities [55]. *Analyzed version: b3abca2*

6.1.3 Closed Source Projects

- **KnowNow**[1] KnowNow is a knowledge management system developed in house at the host organization 'Info Support This project was initiated in early 2015 and discontinued in early 2017. The application was actively developed for these two years by 3 core developers. *Analyzed version: 5d6ec465.*

6.2 Correlation Analysis

Measures with a high correlation to each other can be considered redundant [56]. Meaning adding a metric that has a high correlation with another metric in the same metrics suite, would not yield additional value. Basili et al. [14] did a correlation analysis on the metric suite defined by Chidamber et al. [7]. This research showed that no strong correlation exists between the metrics in this suite. Therefore, none of the metrics in this suite are redundant over each other, according to this research.

In our research, we investigate the correlation between our candidate measures. The results and the motivation of this analysis are presented in Section 6.2.1. The results and motivation for the correlation analysis between our candidate measures and the baseline metrics are described in Section 6.2.2.

6.2.1 Candidate Measures

Since almost all of our candidate measures as described in Section 4.2 are attributes regarding lambda expression usage, we hypothesize our measures are not completely disjunctive and therefore, capturing redundant information.

Theoretical Correlation Analysis

In Figure 6.1, a the hypothesized relationship between the non-disjunctive measures are visualized.

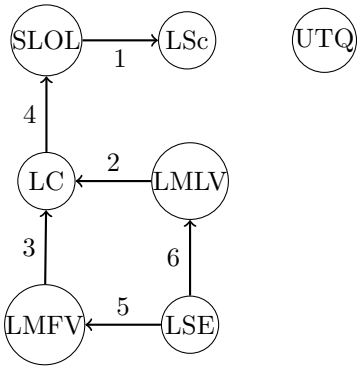

Figure 6.1: Relationship diagram

The measure *UTQ* is disjunctive with the lambda-related measures, since it is possible to have lambda expressions without collection queries and vice versa. Therefore, we do not expect a strong relationship to exists between *UTQ* and the other measures. For each of the edges shown in Figure 6.1, we elaborate on why we hypothesize the corresponding relationship.

1. Even though it is possible that *SLOL* has a high value and *LSc* has a low value or vice-versa, because of the other dependant variable namely *SLOC*. And *SLOC* has an almost implicit correlation with

[1]The dataset from this project was extremely unbalanced (<1% faulty classes), therefore, it was not reliable to use in our set of projects used for validation.

all the defined measures, as bigger classes contain more constructs. Because of this, we also expect *SLOL* to have a relationship to *LSc*.

2. The measure *LMFV* is counting all lambda functions with an extra condition that the lambda function uses a variable that was declared on field scope level. Therefore, $LC \supseteq LMFV$ will always be true.
3. The measure *LMLV* is counting all lambda functions with an extra condition that the lambda function uses a variable that was declared on a local scope level. Therefore, $LC \supseteq LMLV$ will always be true.
4. Since all lambda expressions span at least one line, there will always be at least as many source lines of lambda as lambda expressions. Therefore, $SLOL \supseteq LC$ will always be true.

5./6. For lambda expressions to mutate state outside of its own closure, it will need to use a mutable variable on either class or method level. Therefore, each lambda expression mutating an outside of its own closure variable is also using either a mutable local or field variable.
So, $LMFV \cup LMLV \supseteq LSE$ is valid.

To elaborate on the enumeration above, we describe the mutual inclusion of our candidate measures in Figure 6.2.

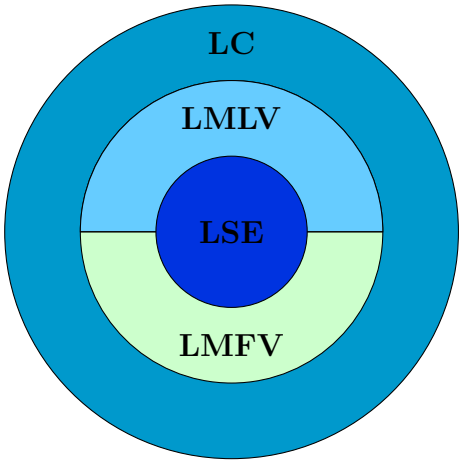

Figure 6.2: Mutual inclusion candidate measures

Empirical Correlation Analysis

For all the projects described in Section 6.1, we analyzed the relationship between our candidate measures. Because correlation only gives an indication of the strength of the relationship, it is more interesting to analyze what proportion of the variance for the dependent variable can be attributed to the independent variable. This is possible by using the statistical measure R-squared (R^2) which is the coefficient of determination as used by Basili et al. [14]. The results of this analysis are shown in Appendix A.1.

For each measure, we look at the R^2 coefficient between the measures and discuss noteworthy findings, which can be calculated by squaring the correlation coefficient. For 3 of the tables the row/column for UTQ/LSE is empty, this can be explained by the constructs that the respective measures are counting do not occur. This results in the values being zero for each of the classes for the measure. If all values are zero for a measure, then the standard deviation will become zero. Since calculating the Pearson correlation requires a division by the multiplied standard deviations, as shown in Equation 6.1, the correlation can not be computed.

$$\rho_{X,Y} = \frac{cov(X,Y)}{\sigma_X \sigma_Y} \tag{6.1}$$

Going through the tables which show the coefficient of determination (R^2), the following findings were made.

- **LSc** A relatively high R^2 value is noticeable with the **SLOL** measure. Even, going up to 0.86 in the Humanizer-project, as seen in Table A.7. This confirms our hypothesis stated in Section 6.2.1.
- **UTQ** As expected the R^2-coefficient to other measures was very low overall project. One exception we found in the Jellyfin-project, is that a still a relatively high coefficient exists to the **LMFV**

measure. This would indicate that unterminated collection queries are relatively often used in combination of mutable field variables.
- **SLOL** Is as we hypothesized, strongly correlating with the **LC** measure.
- **LC** The hypothesized coefficient to **LMFV** and **LMLV** shows a big variance between the analyzed projects. Where this coefficient is relatively high in the openRA project

6.2.2 Baseline Metrics Versus Candidate Measures

We create a univariate regression model (Section 6.3.1) to assess the performance of the measures as an error-proneness predictor. The measure might be a good predictor, but if the measure has a strong correlation to one of the metrics in our baseline model (Section 4.1), then the measure will not add significant value. Therefore, we also analyze the correlation between our defined measures and our baseline.

Theoretical Correlation Analysis

The baseline metrics are focused on either the principles of OOP or general attributes of classes. Our measures are focusing on FP inspired constructs. Therefore, targeting another type of area. So we do not expect a strong correlation between the two sets of measures. However, if the FP inspired constructs occur consistently throughout the codebase, there could be a relationship between our defined measures and **SLOC**.

Empirical Correlation Analysis

The results for the Jellyfin-project are shown in Table 6.1. We see that the coefficient of determination between **SLOL** and **LC, SLOC** is relatively high. This could indicate consistent usage of lambda functions throughout this project, because then more code would imply more lambda functions. The strongest correlation between metrics for this project was **SLOL** and **RFC**. In the definition of Response for Class (RFC) in Section 4.1, we show that we excluded lambda functions in calculation of the response set. Therefore, this correlation is unexpected.

Table 6.1: R^2-values for the Jellyfin-project

	CC	SLOC	CD	WMC	DIT	NOC	CBO	RFC	LCOM
LSc	0.022	0.034	0.005	0.029	0.002	0.0	0.016	0.048	0.012
LC	0.364	0.513	0.006	0.443	0.0	0.008	0.136	0.605	0.514
SLOL	0.468	**0.606**	0.004	0.518	0.0	0.005	0.083	**0.675**	0.5
LMFV	0.199	0.289	0.002	0.242	0.0	0.0	0.035	0.347	0.313
LMLV	0.377	0.46	0.002	0.381	0.002	0.001	0.027	0.505	0.282
UTQ	0.132	0.211	0.003	0.169	0.0	0.009	0.1	0.239	0.209
LSE	0.232	0.281	0.0	0.243	0.001	0.002	0.017	0.323	0.262

Looking at Table 6.2 we see much lower coefficients than for the Jellyfin project. Including the coeffictincts Also for coefficient between **SLOC** and **LC, SLOL, RFC**.

Table 6.2: R^2-values for the ASP.NET-project

	CC	SLOC	CD	WMC	DIT	NOC	CBO	RFC	LCOM
LSc	0.0	0.001	0.001	0.001	0.001	0.001	0.004	0.007	0.0
LC	0.016	**0.029**	0.002	0.025	0.002	0.0	0.029	0.113	0.009
SLOL	0.006	**0.011**	0.0	0.01	0.0	0.0	0.01	**0.04**	0.001
LMFV	0.002	0.004	0.0	0.004	0.0	0.0	0.004	0.012	0.0
LMLV	0.005	0.007	0.0	0.008	0.0	0.0	0.004	0.021	0.0
UTQ	0.01	0.021	0.0	0.016	0.001	0.0	0.007	0.064	0.049
LSE	0.005	0.006	0.0	0.008	0.0	0.001	0.004	0.007	0.001

The coefficients of determination between the baseline metrics and candidate measures as presented in Appendix A.2. We see a lot of variance between the projects. For our candidate measures to be redundant over the baseline metrics, a consistently strong correlation needs to be present. This is not the case, therefore, our measures are not redundant over the baseline metrics.

6.3 Results

In Table 6.3 descriptive stats are shown for the output of our project analysis. The test projects have been excluded from our analysis. Test projects are likely to be modified in a bug-fixing commit to detect the bug if it would occur again, results in a false positive in our error-prone marking of classes.

Table 6.3: Descriptive Stats

Project	Classes	Bug-fixes	Faulty classes
IS4	331	40	49
AKK	1621	171	199
ASP	3212	99	118
EF	1432	975	437
CLI	328	54	24
ML	1404	27	18
HUM	124	23	10
DNS	5345	?[2]	159
JF	1420	81	88
ILS	1011	95	70
ORA	1990	227	149

Notable, the relationship of bug-fixing commits on faulty classes can be either positive or negative. This can be explained by that a set of the commits are only updating the configuration or non-C# code files, therefore, $|BugFixes| > |FaultyClasses|$ is possible. The alternative scenario, one commit is able to modify multiple classes. Therefore, $|BugFixes| < |FaultyClasses|$ is possible.

The variance in the ratio of $\frac{Classes}{FaultyClasses}$ between project is also notable, this can be partly attributed during the lifetime of a project. However, some projects contradict this hypothesis. ILSpy is one of the older projects that were analyzed, the ratio is not higher than e.g. the AKKA.NET project. Even though the ILSpy project is more than twice as old.

If the FP inspired constructs that were defined for our measures are underrepresented, then the results might give incorrect insights caused by randomness. E.g. if only two classes contain lambda functions and these classes would by coincidence also be faulty. Then the prediction model will have a strong bias towards predicting that classes with lambda functions are faulty. The total amount of classes and the classes where the corresponding construct occur for each project are shown in Table 6.4.

[2]During the research the labels in the repository of dnSpy were deleted. Therefore, we are unable to derive the bug-fix count.

Table 6.4: FP inspired constructs occurrences

	Classes	LC	LMFV	LMLV	UTQ	LSE
ID4	331	58	22	10	0	0
AKK	1621	487	198	207	36	84
ASP	3212	398	61	130	100	18
EF	1432	334	53	91	0	4
CLI	328	109	24	31	38	3
ML	1404	589	188	339	4	97
HUM	124	13	2	3	3	0
DNS	5345	833	240	240	14	44
JF	1420	258	77	140	85	11
ILS	1011	207	51	77	32	13
ORA	1990	775	584	388	221	113

6.3.1 Univariate Logistic Regression Prediction Model

To evaluate the measures in isolation, we fit and evaluate a prediction model using univariate regression. All our candidate measures are evaluated in isolation for each project. The results for this evaluation are shown in Table 6.5. The results present the macro-average F1-score for each project in combination with each candidate measure.

Table 6.5: Descriptive Stats

F1-score	LSc	LC	SLOL	LMFV	LMLV	UTQ	LSE
ID4	0.46	0.46	0.46	0.47	0.47	0.46	0.46
AKK	0.47	0.48	0.48	0.47	0.48	0.47	0.47
ASP	0.49	0.49	0.49	0.49	0.49	0.49	0.49
EF	0.41	0.42	0.41	0.41	0.43	0.41	0.41
CLI	0.48	0.48	0.48	0.48	0.48	0.48	0.48
ML	0.5	0.5	0.5	0.5	0.5	0.5	0.5
HUM	0.48	0.48	0.48	0.48	0.48	0.48	0.48
DNS	0.49	0.51	0.51	0.5	0.49	0.49	0.5
JF	0.48	0.53	0.5	0.49	0.49	0.49	0.49
ILS	0.48	0.5	0.52	0.5	0.48	0.48	0.48
ORA	0.48	0.49	0.48	0.49	0.49	0.49	0.48

As these results are hard to read and compare measures and projects to each other. We have chosen to visualize the results with a multiple line graph. Using this graph type it is possible to see how each of the projects score relative to each other. Spikes will indicate that the corresponding measure is a stronger feature than the ones next to it. The results from Table 6.5 are visualized in Figure 6.3.

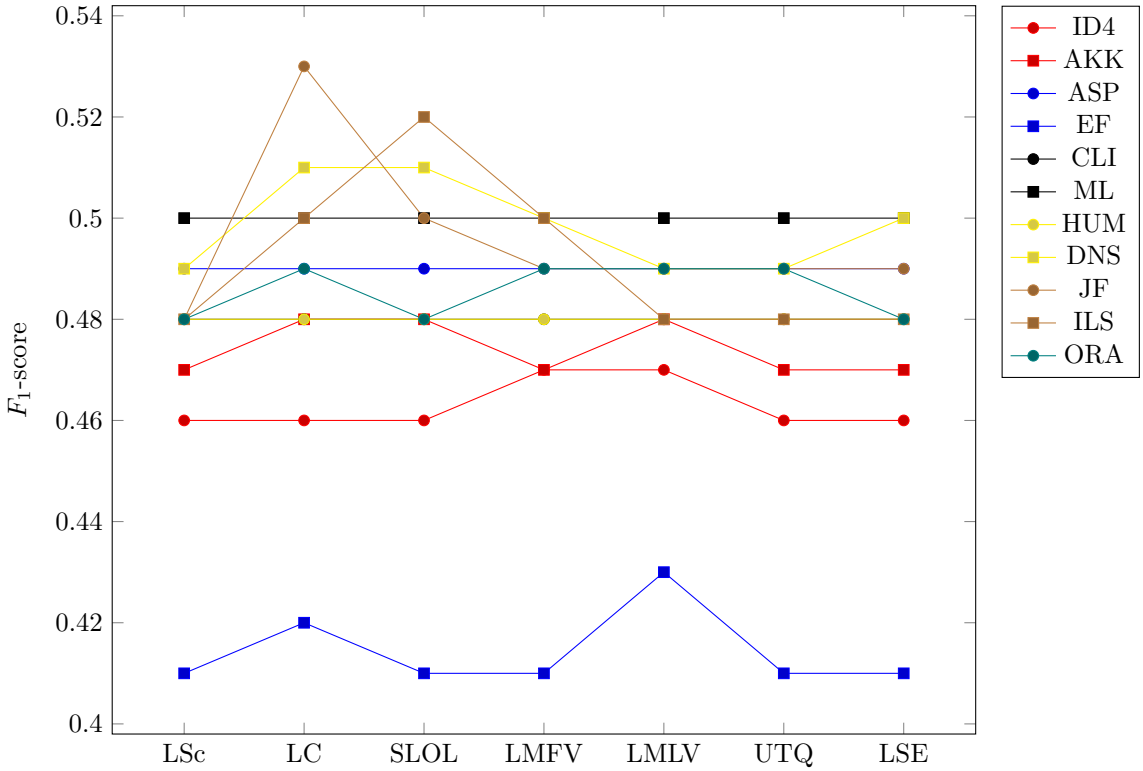

Figure 6.3: F_1-score Prediction model: univariate regression

In Figure 6.3, we see the measures Source Lines of Lambda (SLOL) and Lambda Count (LC) perform well on the projects: ILSpy and JellyFin. The univariate regression models created for the Entity Framework project, perform relatively bad compared to the other projects. Looking at the raw input for the project, we see that only $\frac{1}{5}$ of the classes of the EF project use lambda expressions, this could attribute to the relative low F_1-score. Comparing this to other projects e.g. AKKA.NET the ratio is $\frac{1}{3}$. The fewer usage of lambda expressions could influence the usability of our measures. The Humanizer project seems to score an almost stable 0.48. When looking into the raw output data from our static analysis we see only $\frac{1}{9}$ classes in this project uses lambda expressions. Therefore, this project might not be functional enough for our measures to yield any value.

6.3.2 Multivariate Logistic Regression Prediction Model

To evaluate the value of our candidate measures compared to our baseline, we create a multivariate regression model based on K-Best features for each of the projects. We again evaluate the model by

Table 6.6: Inclusion Candidate Measures K-Best Model

ID4	0.52	0.52
AKK	0.55	0.54
ASP	0.5	0.53
EF	0.58	0.6
CLI	0.5	0.53
ML	0.51	0.5
HUM	0.51	0.55
DNS	0.52	0.53
JF	0.58	0.58
ILS	0.57	0.56
ORA	0.53	0.55

In Table 6.7 is shown how many out of 11 K-best models included the corresponding measure as a feature.

Table 6.7: Inclusion candidate measures K-Best model

Measure	# Models
LSc	3
LC	6
SLOL	9
LMLV	4
LMFV	6
LSE	0
UTQ	0

Most notable is that 9 out of 11 projects include the Source Lines of Lambda (SLOL)-measure as described in Section 4.2. The one project where SLOL was excluded in the K-Best, was the Humanizer project. As described earlier, the project does not use a lot of the FP inspired constructs and therefore, is not suitable for our measures. The measure LSE counting the numbers of side-effects in lambdas and UTQ, counting the number of unterminated collection queries, both do not seem to yield an interesting result. Even though these FP inspired constructs do occur in almost all project, the amount of occurrences is too limited to yield good value.

To do a comparison between our baseline model and the selection of the K-best features model, the projects are plotted in Figure 6.4.

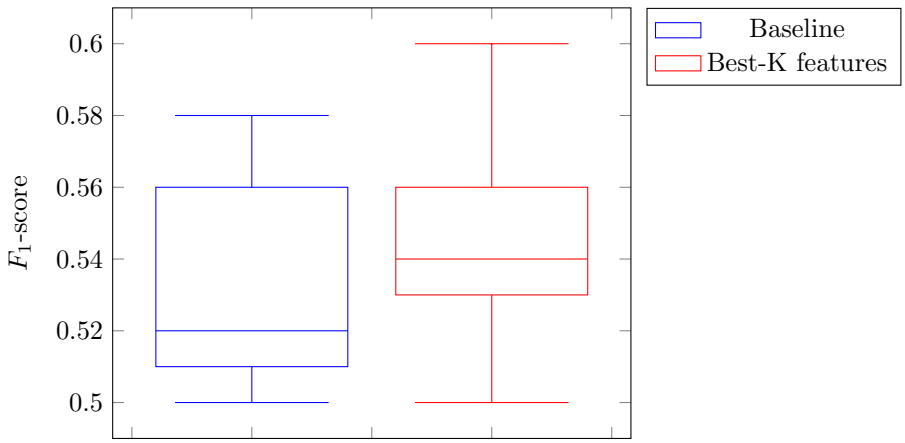

Figure 6.4: F_1-score prediction model: multivariate regression

Looking at Figure 6.4 one can see that the worst-performing project did not gain improvement in performance. However, the first quartile had a performance increase of 0.02. The best performing project also has a 0.02 increase in performance.

6.4 Threats to Validity

Generalizability In this research, 11 open-source projects were analyzed. For the results of the research to be useful, substantiated claims need to be made about the generalizability of the results. Since we only processed 11 projects, we are unable to make any claims regarding that aspect. By analyzing a bigger corpus of projects, one could make an easier distinction on what different type of project our proposed measures yield value.

Bug-fix commit marking We marked commits as bug-fixing commits by analyzing the meta-data of the commits. Where a commit message format can be enforced by an organization, our regular expression is not able to guarantee the absence of false positives or true negatives.

Bug fixes vs Bugs To make an estimation on how error-prone a class is, the assumption was made that bug-fixes made in a class measure the error-proneness of the class. However, there is no way to guarantee that a class which was never updated by a bug-fix is bug-free. Bugs might have not been identified yet, or maybe bugs that were never fixed by a bug-fix commit, were accidentally fixed during a refactoring.

Analyzed classes In our current approach we calculate the metric values only for the classes in the latest version of the project. However, the values for these metrics might differ drastically over different versions of the classes, where classes could have been refactored between the faulty version and the analyzed version.

Scale type For our logistic regression we assume that our scales for the measures are interval based on intuition. However, we are unable to demonstrate that the scale types for our measures are interval. As Briand et al. [1] states, when unsure about the type of the scale, treating such measures as ordinal would result in discarding important information.

Fault types In the current approach for assessing error-proneness in our research, no distinction is made between different bug-types. The problem with the current approach for this aspect is that we also include bugs that were for example caused by a typo. Even though it is hard to argue that our functional inspired constructs attribute to a typo mistake. It would be interesting to argue about each of the fault types how they are or are not impacted our defined constructs. This way would have a stronger case arguing in the validity of the results of the research.

Chapter 7

Related work

Uesbeck et al. [57] did a control experiment where the impact of lambdas in C++ on productivity, compiler errors, percentage of time to fix the compiler errors. The results show that the use of lambdas, as opposed to iterators, on the number of tasks completed was significant. *"The percentage of time spent on fixing compilation errors was 56.37% for the lambda group while it was 44.2% for the control group with 3.5 % of the variance being explained by the group difference."*. Where the groups consisted of developers with different amount of programming experience.
The increased time of fixing compiler error where lambda function was used, which seems likely to be the result of lambda expressions being harder to reason about. Which strengthens our hypothesis.

Finifter [27] shows how verifiable purity can be used to verify high-level security properties. By combining determinism with object-capabilities a new class of languages is described that allows purity in largely imperative programs.

Landkroon [5] shows that metrics from functional programming and object-oriented programming can be combined to evaluate the multi-paradigm programming language Scala.
Srininvasan [58] discusses existing empirical and theoretical methodologies of validating software metrics.
Meneely [59] reveals in a systematic literature review study that the academic literature contains a diverse set of constraints on what a valid metric should satisfy.
Heitlager [8] proposed a maintainability model where the unit complexity of software is mapped to the 'changeability' sub-characteristic. The changeability sub-characteristic is defined in the ISO9126 document and translates to the modifiability sub-characteristic in the ISO25010 document. The maintainability model proposed by Heitlager indicates that Cyclomatic-/McCabe complexity[9] and Coupling give a good indication of the complexity of the unit.

Table 7.1 shows what research is done for code evaluation for patterns originally come from paradigm X but manifest in a language that is categorized as a Y-paradigm language.
It also shows how there are gaps for which areas no research was found, the first gap shows that there are no studies found on how patterns from the object-oriented code that manifests in functional languages can be evaluated. The second gap shows that there are no studies found on how patterns from the functional paradigm that manifest in object-oriented languages can be evaluated.
The purpose of this research is to cover the second gap, which would increase our understanding of what impact the mixing of the two paradigms has on code quality.

	Objected-oriented patterns	Functional patterns
Object-oriented paradigm	[14][29][8]*	
Functional paradigm		[10] [11]*
Multi-paradigm	[5]	[5]

Table 7.1: Literature survey results

* *non-exhaustive*

Chapter 8

Conclusion

With the results obtained in Chapter 6, we can answer our research questions defined in Section 1.1.

RQ1 What new constructs occur, that are neither functional nor object-oriented, when introducing functional programming inspired features to the object-oriented language C#?
We investigated the evolution of the OO language C# and what features inspired by the FP paradigm are added. The development and introduction of the FP inspired features seems to be going rapid and there is no indication of the development slowing down. This new declarative syntax enables more concise code constructs. Therefore, enables the software engineer to write more functionality with less code. However, these constructs are introduced without the constraints that would be present in FP languages. Therefore, impure usages of concepts that were designed pure, exist in the OO language C#. To cover the new type of complexity introduced by these FP inspired constructs and impure usages of these constructs, we defined measures. The defined measures cover the following FP inspired constructs: lambda expression usages and impure usages, in which the expression its evaluation is affected by or affects the outside state. Furthermore, we defined a measure to report the unterminated collection queries.

RQ2 To which extent can we improve our baseline prediction model by adding code metrics covering functional inspired constructs?
The candidate measure Source Lines of Lambda (SLOL) (described in Section 4.2) yielded promising results when used in a univariate prediction model for all of the projects where FP inspired constructs were actively used. To assess if we can improve our baseline model, we swapped out the weaker metrics from the baseline model with stronger metrics based on our set of defined measures. We were able to achieve a marginal improvement (F_1-score 0.0-0.02) with respect to different projects. For some projects, we were able to achieve a small improvement in the prediction model. On the contrary, the projects where a low amount of FP inspired constructs were used, the candidate measures did not yield value.

RQ How does usage of functional programming inspired constructs relate to error-proneness in the object-oriented language C#?
To answer **RQ1** we investigated FP inspired constructs that were added to C#. However, C# did not add the constraints to these constructs that exist in FP languages. Impure usages of these constructs are possible in C#. This can lead to unexpected behaviors. We defined measures that will capture these impure usages. We saw that these constructs occur in almost every project that we analyzed. We did find a correlation between our measures and error-proneness. But the presence of this correlation is too uncertain to make claims about causality.

As described earlier in Section 1 the significant OO languages seem to adopt more features from the FP paradigm. Our hypothesis is that the set of FP inspired features will become bigger and receive a more FP-like syntax. The increase of performance in our prediction models found in answering **RQ2** seems marginal for now. But we hypothesize that their relevance will increase in the future, based on the ongoing evolution of OO languages and the increasing adoption by developers of these inspired features.

8.1 Future work

We see variance in the performance of our measures regarding the different projects. There are a lot of factors that can influence why the measures perform well on some projects and bad on other projects. Digging deeper in the development of the project, to explain (some of) the variance would be interesting as as follow-up research.

In our study, we focus on the OO language C#, where some people might argue that C# is now a multi-paradigm-language. C# was designed as an OO language. The FP in C# came as a bonus of syntactic sugar instead of being chosen for its FP features. Our hypothesis is that in languages that are from origin multi-paradigm, are explicitly chosen for their functional features. Therefore, the FP inspired constructs would be used more FP-aware and that would result in a lower error-proneness regarding the use of FP inspired constructs. This hypothesis would be interesting to look in to.

For this research, we analyzed a collection of 11 C# codebases. It would be interesting to explore how applicable these metrics are on other object-oriented languages. This is interesting because for example not all the OO languages have the same constraints on the use of lambda expressions.

In the OO language Java, an additional constraint is added. All variables that are referenced from a lambda function, that are outside the lambda function its closure, have to be 'final'. Final meaning that the variable may only be initiated and afterward, it may not be updated [60]. This results into a more pure design. However, if this references variable is an object pointer. The object can still be modified even though it can not be reassigned. So this additional purity constraint, is only significant for primitive types.

In the OO language C++, usage of lambda expressions have other constraints. It is only possible to use variables from outside the lambda expression its closure, if they are specified in the capture clause. This capture clause is defined in square brackets before the parameter specification of the lambda expression as seen in Listing 20. When specifying the inclusion of a variable in the lambda expression, it has to be specified if the variable has to be used by reference or by copy. This way, it is easier to reason about what variables are from what closure. An empty capture clause, still does not guarantee purity. This is because imported libraries, such as `DateTime` and `I\O` are still available inside the lambda function its closure.

```
1 [](float a, float b) { return (std::abs(a) < std::abs(b)); }
```

Listing 20: Lambda expression in C++

8.2 Reflection

After investigating FP inspired constructs, I have seen the benefits and the problems that come with them. I can not say if I find FP inspired constructs good or bad. Digging into the code of the projects that I analyzed, I encountered many constructions which I consider anti-patterns. E.g. too complex lambda expressions spanning multiple lines or combining side effects with lazy evaluation. However, I can still say that in most cases the newly introduced syntax provides a more concise and readable manner of writing code. From a personal perspective I would like to see more FP inspired features and a more declarative syntax in the OO languages. However, I would encourage myself and others developers working on the same codebase, to use these FP inspired constructs in the same way, you would use them in a FP language. So keep your lambda functions pure. Lambda functions are often created inline, in e.g. a LINQ-expressions. Therefore, they should be kept simple. If lambda expressions span multiple lines, readability will decrease drastically. In this case the function, should be a named function, where return and parameter types are transparent.

Bibliography

[1] 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.

[2] *PYPL*, http://pypl.github.io/PYPL.html, Accessed: 2019-01-11.

[3] *Java 8 update notes*, https://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html, Accessed: 2019-01-11.

[4] *C# 3 update notes*, https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-version-history, Accessed: 2019-01-23.

[5] E. Landkroon, *Code quality evaluation for the multi-paradigm programming language scala*, 2017.

[6] B. Jaruzelski, *The 2018 GlobalInnovation 1000 study*, https://www.strategyand.pwc.com/innovation1000, Accessed: 2019-08-30.

[7] 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.

[8] I. Heitlager, T. Kuipers, and J. Visser, "A practical model for measuring maintainability", in *Proceedings of the 6th International Conference on Quality of Information and Communications Technology*, IEEE, 2007, pp. 30–39.

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

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

[11] K. Van den Berg, "Software measurement and functional programming", *University of Twente*, 1995.

[12] Info Support, *Over Info Support*, https://www.infosupport.com/info-support-b-v/, Accessed: 2019-08-30.

[13] "Systems and Software Engineering – Systems and Software Quality Requirements and Evaluation (SQuaRE) –System and Software Quality Models", International Organization for Standardization, Geneva, CH, Standard, Mar. 2011.

[14] 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.

[15] S. P. Jones, *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[16] M. Wenzel, *Functional Programming vs. Imperative Programming (C#)*, https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/functional-programming-vs-imperative-programming, Accessed: 2019-07-15.

[17] J. Alama and J. Korbmacher, "The lambda calculus", in *The Stanford Encyclopedia of Philosophy*, E. N. Zalta, Ed., Winter 2018, Metaphysics Research Lab, Stanford University, 2018.

[18] H. P. Barendregt, "Lambda calculi with types", 1992.

[19] M. Torgersen, *Do more with patterns in C# 8.0*, https://devblogs.microsoft.com/dotnet/do-more-with-patterns-in-c-8-0/, Accessed: 2019-07-15.

[20] H. Abelson, G. J. Sussman, and J. Sussman, *Structure and interpretation of computer programs*. Justin Kelly, 1996.

[21] I. D. B. Stark, "Names and higher-order functions", PhD thesis, Citeseer, 1994.

[22] B. Wagner, *Language Integrated Query (LINQ)*, https://github.com/dotnet/cli, Accessed: 2019-07-15, 2017.

[23] *Lazy Initialization*, https://docs.microsoft.com/en-us/dotnet/framework/performance/lazy-initialization, Accessed: 2019-07-15.

[24] *C# 7 update notes*, https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-7, Accessed: 2019-03-06.

[25] P. Hudak, S. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzmán, K. Hammond, J. Hughes, T. Johnsson, *et al.*, "Report on the programming language Haskell: A non-strict, purely functional language version 1.2", *ACM SigPlan notices*, vol. 27, no. 5, pp. 1–164, 1992.

[26] *Haskell Documentation*, https://wiki.haskell.org/Pure, Accessed: 2019-01-24.

[27] M. Finifter, A. Mettler, N. Sastry, and D. Wagner, "Verifiable functional purity in Java", in *Proceedings of the 15th ACM conference on Computer and communications security*, ACM, 2008, pp. 161–174.

[28] 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.

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

[30] D. W. Hosmer Jr, S. Lemeshow, and R. X. Sturdivant, *Applied logistic regression*. John Wiley & Sons, 2013, vol. 398.

[31] F. Lanubile and G. Visaggio, "Evaluating predictive quality models derived from software measures: Lessons learned", *Journal of Systems and Software*, vol. 38, no. 3, pp. 225–234, 1997.

[32] M. Stone, "Cross-validatory choice and assessment of statistical predictions", *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 36, no. 2, pp. 111–133, 1974.

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

[34] M. Sokolova and G. Lapalme, "A systematic analysis of performance measures for classification tasks", *Information Processing & Management*, vol. 45, no. 4, pp. 427–437, 2009.

[35] V. Nguyen, S. Deeds-Rubin, T. Tan, and B. Boehm, "A sloc counting standard", in *Cocomo ii forum*, Citeseer, vol. 2007, 2007, pp. 1–16.

[36] P. Kulikov, *Boolean logical operators (C# reference)*, https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/boolean-logical-operators, Accessed: 2019-07-18.

[37] SonarQube, *Metric Definitions*, https://docs.sonarqube.org/latest/user-guide/metric-definitions/, Accessed: 2019-07-15, 2019.

[38] O. Arafati and D. Riehle, "The comment density of open source software code", in *2009 31st International Conference on Software Engineering-Companion Volume*, IEEE, 2009, pp. 195–198.

[39] A. Rooney, "Object Orientation in Java", *Foundations of Java for ABAP Programmers*, pp. 49–56, 2006.

[40] Microsoft, *Azure DevOps User Guide*, https://docs.microsoft.com/en-us/azure/devops/user-guide/what-happened-vsts, Accessed: 2019-05-01.

[41] ——, *Azure DevOps API documentation*, https://docs.microsoft.com/en-us/rest/api/azure/devops/?view=azure-devops-rest-5.0, Accessed: 2019-05-01.

[42] N. Schonning, *The .NET compiler platform SDK*, https://docs.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/, Accessed: 2019-07-19.

[43] ——, *Get started with semantic analysis*, https://docs.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/get-started/semantic-analysis, Accessed: 2019-07-19.

[44] GitHub, *Closing issues using keywords*, https://help.github.com/en/articles/closing-issues-using-keywords, Accessed: 2019-07-15, 2019.

[45] .NET Foundation, *CLI*, https://github.com/dotnet/cli, 2015.

[46] ——, *Machine learning*, https://github.com/dotnet/machinelearning, 2018.

[47] ——, *IdentityServer4*, https://github.com/IdentityServer/IdentityServer4, 2015.

[48] ——, *Aspnetcore*, https://github.com/aspnet/AspNetCore, 2015.

[49] ——, *EntityFrameworkCore*, https://github.com/aspnet/EntityFrameworkCore, 2014.

[50] AkkaDotNet, *Akka.net*, https://github.com/akkadotnet/akka.net, 2014.

[51] Jellyfin, *Jellyfin*, https://github.com/jellyfin/jellyfin, 2013.

[52] C. Forbes, *OpenRA*, https://github.com/OpenRA/OpenRA, 2009.

[53] 0xd4d, *Dnspy*, https://github.com/0xd4d/dnSpy, 2016.

[54] ICSharpCode, *ILSpy*, https://github.com/icsharpcode/ILSpy, 2009.

[55] M. Khalili, *Humanizer*, https://github.com/Humanizr/Humanizer, 2012.

[56] D. Landman, A. Serebrenik, E. Bouwers, and J. J. Vinju, "Empirical analysis of the relationship between CC and SLOC in a large corpus of Java methods and C functions", *Journal of Software: Evolution and Process*, vol. 28, no. 7, pp. 589–618, 2016.

[57] P. M. Uesbeck, A. Stefik, S. Hanenberg, J. Pedersen, and P. Daleiden, "An empirical study on the impact of c++ lambdas and programmer experience", in *Proceedings of the 38th International Conference on Software Engineering*, ACM, 2016, pp. 760–771.

[58] K. Srinivasan and T. Devi, "Software metrics validation methodologies in software engineering", *International Journal of Software Engineering & Applications*, vol. 5, no. 6, p. 87, 2014.

[59] A. Meneely, B. Smith, and L. Williams, "Validating software metrics: A spectrum of philosophies", *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 21, no. 4, p. 24, 2012.

[60] Oracle, *Java Language Specification*, https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf, Accessed: 2019-08-29.

Acronyms

AST abstract syntax tree. 19, 20, 23, 24
cbo Coupling between Objects. 23
CC Cyclomatic Complexity. 7, 14, 15
FP functional programming. 4–7, 13–15, 30, 37
IDE integrated development environment. 14, 23
LC Lambda Count. 32, 33
LMFV Lambda Local Variable Usages. 32
LMLV Lambda Local Variable Usages. 32
LSc Lambda Score. 19, 32
LSE Lambda Side Effects. 32
NOC Number of Children. 23
OO object-oriented. 4–7, 13, 15, 16, 19, 37, 38
OOP object-oriented programming. 18, 30
R&D Research & Development. 4
RFC Response for Class. 18, 30
SDM semantic data model. 19, 20, 23
SLOC Source Lines of Code. 7, 14, 15, 19
SLOL Source Lines of Lambda. 19, 32–34, 37
UTQ Unterminated Collection Queries. 32

Appendix A

A.1 Candidate Measure Correlation

Table A.1: R^2-values for the IdentityServer4-project

	LSc	LC	SLOL	LMFV	LMLV	UTQ	LSE
LSc	1.0	0.377	0.755	0.413	0.018		
LC	0.377	1.0	0.345	0.285	0.246		
SLOL	0.755	0.345	1.0	0.536	0.034		
LMFV	0.413	0.285	0.536	1.0	0.062		
LMLV	0.018	0.246	0.034	0.062	1.0		
UTQ							
LSE							

Table A.2: R^2-values for the AKKA.NET-project

	LSc	LC	SLOL	LMFV	LMLV	UTQ	LSE
LSc	1.0	0.274	0.381	0.367	0.285	0.013	0.252
LC	0.274	1.0	0.518	0.403	0.408	0.128	0.189
SLOL	0.381	0.518	1.0	0.635	0.855	0.061	0.437
LMFV	0.367	0.403	0.635	1.0	0.48	0.038	0.378
LMLV	0.285	0.408	0.855	0.48	1.0	0.061	0.323
UTQ	0.013	0.128	0.061	0.038	0.061	1.0	0.017
LSE	0.252	0.189	0.437	0.378	0.323	0.017	1.0

Table A.3: R^2-values for the ASP.NET-project

	LSc	LC	SLOL	LMFV	LMLV	UTQ	LSE
LSc	1.0	0.289	0.523	0.08	0.363	0.001	0.049
LC	0.289	1.0	0.464	0.194	0.223	0.055	0.013
SLOL	0.523	0.464	1.0	0.208	0.664	0.003	0.115
LMFV	0.08	0.194	0.208	1.0	0.079	0.003	0.006
LMLV	0.363	0.223	0.664	0.079	1.0	0.0	0.186
UTQ	0.001	0.055	0.003	0.003	0.0	1.0	0.0
LSE	0.049	0.013	0.115	0.006	0.186	0.0	1.0

Table A.4: R^2-values for the EntityFramework-project

	LSc	LC	SLOL	LMFV	LMLV	UTQ	LSE
LSc	1.0	0.224	0.244	0.099	0.07		0.04
LC	0.224	1.0	0.892	0.058	0.012		0.0
SLOL	0.244	0.892	1.0	0.252	0.012		0.003
LMFV	0.099	0.058	0.252	1.0	0.001		0.0
LMLV	0.07	0.012	0.012	0.001	1.0		0.423
UTQ							
LSE	0.04	0.0	0.003	0.0	0.423		1.0

	LSc	LC	SLOL	LMFV	LMLV	UTQ	LSE
LSc	1.0	0.24	0.491	0.185	0.266	0.008	0.04
LC	0.24	1.0	0.384	0.31	0.11	0.166	0.003
SLOL	0.491	0.384	1.0	0.583	0.794	0.031	0.255
LMFV	0.185	0.31	0.583	1.0	0.431	0.023	0.018
LMLV	0.266	0.11	0.794	0.431	1.0	0.003	0.37
UTQ	0.008	0.166	0.031	0.023	0.003	1.0	0.001
LSE	0.04	0.003	0.255	0.018	0.37	0.001	1.0

Table A.5: R^2-values for the CLI-project

Table A.6: R^2-values for the Machine Learning-project

	LSc	LC	SLOL	LMFV	LMLV	UTQ	LSE
LSc	1.0	0.253	0.386	0.231	0.221	0.003	0.145
LC	0.253	1.0	0.539	0.256	0.296	0.013	0.174
SLOL	0.386	0.539	1.0	0.501	0.808	0.003	0.488
LMFV	0.231	0.256	0.501	1.0	0.385	0.002	0.236
LMLV	0.221	0.296	0.808	0.385	1.0	0.001	0.549
UTQ	0.003	0.013	0.003	0.002	0.001	1.0	0.001
LSE	0.145	0.174	0.488	0.236	0.549	0.001	1.0

Table A.7: R^2-values for the Humanizer-project

	LSc	LC	SLOL	LMFV	LMLV	UTQ	LSE
LSc	1.0	0.76	0.863	0.146	0.009	0.057	
LC	0.76	1.0	0.92	0.381	0.114	0.187	
SLOL	0.863	0.92	1.0	0.306	0.09	0.148	
LMFV	0.146	0.381	0.306	1.0	0.0	0.157	
LMLV	0.009	0.114	0.09	0.0	1.0	0.211	
UTQ	0.057	0.187	0.148	0.157	0.211	1.0	
LSE							

Table A.8: R^2-values for the dnSpy-project

	LSc	LC	SLOL	LMFV	LMLV	UTQ	LSE
LSc	1.0	0.22	0.236	0.048	0.089	0.0	0.033
LC	0.22	1.0	0.619	0.264	0.183	0.007	0.063
SLOL	0.236	0.619	1.0	0.357	0.611	0.005	0.417
LMFV	0.048	0.264	0.357	1.0	0.304	0.0	0.155
LMLV	0.089	0.183	0.611	0.304	1.0	0.002	0.44
UTQ	0.0	0.007	0.005	0.0	0.002	1.0	0.002
LSE	0.033	0.063	0.417	0.155	0.44	0.002	1.0

Table A.9: R^2-values for the JellyFin-project

	LSc	LC	SLOL	LMFV	LMLV	UTQ	LSE
LSc	1.0	0.14	0.23	0.1	0.087	0.048	0.021
LC	0.14	1.0	0.62	0.492	0.211	0.439	0.145
SLOL	0.23	0.62	1.0	0.468	0.717	0.232	0.358
LMFV	0.1	0.492	0.468	1.0	0.171	0.201	0.17
LMLV	0.087	0.211	0.717	0.171	1.0	0.068	0.477
UTQ	0.048	0.439	0.232	0.201	0.068	1.0	0.047
LSE	0.021	0.145	0.358	0.17	0.477	0.047	1.0

Table A.10: R^2-values for the ILSpy-project

	LSc	LC	SLOL	LMFV	LMLV	UTQ	LSE
LSc	1.0	0.055	0.433	0.079	0.727	0.003	0.451
LC	0.055	1.0	0.628	0.09	0.062	0.023	0.021
SLOL	0.433	0.628	1.0	0.329	0.479	0.014	0.344
LMFV	0.079	0.09	0.329	1.0	0.141	0.013	0.224
LMLV	0.727	0.062	0.479	0.141	1.0	0.014	0.37
UTQ	0.003	0.023	0.014	0.013	0.014	1.0	0.001
LSE	0.451	0.021	0.344	0.224	0.37	0.001	1.0

Table A.11: R^2-values for the OpenRA-project

	LSc	LC	SLOL	LMFV	LMLV	UTQ	LSE
LSc	1.0	0.264	0.406	0.328	0.367	0.032	0.24
LC	0.264	1.0	0.742	0.714	0.464	0.132	0.345
SLOL	0.406	0.742	1.0	0.835	0.741	0.065	0.54
LMFV	0.328	0.714	0.835	1.0	0.705	0.071	0.493
LMLV	0.367	0.464	0.741	0.705	1.0	0.034	0.397
UTQ	0.032	0.132	0.065	0.071	0.034	1.0	0.026
LSE	0.24	0.345	0.54	0.493	0.397	0.026	1.0

A.2 Candidate Measure with Baseline Metrics Correlation

Table A.12: R^2-values for the IdentityServer4-project

	CC	SLOC	CD	WMC	DIT	NOC	CBO	RFC	LCOM
LSc	0.001	0.002	0.005	0.0	0.007	0.0	0.001	0.008	0.0
LC	0.049	0.073	0.015	0.057	0.016	0.001	0.01	0.117	0.001
SLOL	0.002	0.016	0.011	0.004	0.006	0.0	0.001	0.023	0.0
LMFV	0.008	0.018	0.007	0.01	0.006	0.0	0.0	0.015	0.0
LMLV	0.148	0.099	0.004	0.141	0.005	0.0	0.045	0.099	0.004
UTQ									
LSE									

Table A.13: R^2-values for the AKKA.NET-project

	CC	SLOC	CD	WMC	DIT	NOC	CBO	RFC	LCOM
LSc	0.055	0.066	0.032	0.035	0.076	0.0	0.006	0.089	0.002
LC	0.376	0.415	0.022	0.373	0.046	0.0	0.051	0.565	0.069
SLOL	0.27	0.268	0.018	0.214	0.043	0.0	0.035	0.306	0.015
LMFV	0.201	0.201	0.019	0.142	0.049	0.0	0.026	0.225	0.011
LMLV	0.236	0.227	0.013	0.193	0.033	0.0	0.03	0.278	0.021
UTQ	0.093	0.103	0.002	0.082	0.0	0.0	0.026	0.116	0.005
LSE	0.095	0.11	0.009	0.072	0.024	0.001	0.041	0.119	0.01

Table A.14: R^2-values for the ASP.NET-project

	CC	SLOC	CD	WMC	DIT	NOC	CBO	RFC	LCOM
LSc	0.0	0.001	0.001	0.001	0.001	0.001	0.004	0.007	0.0
LC	0.016	0.029	0.002	0.025	0.002	0.0	0.029	0.113	0.009
SLOL	0.006	0.011	0.0	0.01	0.0	0.0	0.01	0.04	0.001
LMFV	0.002	0.004	0.0	0.004	0.0	0.0	0.004	0.012	0.0
LMLV	0.005	0.007	0.0	0.008	0.0	0.0	0.004	0.021	0.0
UTQ	0.01	0.021	0.0	0.016	0.001	0.0	0.007	0.064	0.049
LSE	0.005	0.006	0.0	0.008	0.0	0.001	0.004	0.007	0.001

Table A.15: R^2-values for the EntityFramework-project

	CC	SLOC	CD	WMC	DIT	NOC	CBO	RFC	LCOM
LSc	0.001	0.012	0.0	0.002	0.006	0.0	0.005	0.032	0.0
LC	0.025	0.087	0.003	0.029	0.001	0.0	0.055	0.163	0.0
SLOL	0.008	0.064	0.003	0.012	0.0	0.0	0.034	0.098	0.
0 LMFV	0.003	0.032	0.002	0.008	0.0	0.0	0.0	0.01	0.0
LMLV	0.069	0.07	0.005	0.062	0.0	0.0	0.002	0.054	0.0
UTQ									
LSE	0.002	0.002	0.0	0.002	0.0	0.0	0.0	0.001	0.0

Table A.16: R^2-values for the CLI-project

	CC	SLOC	CD	WMC	DIT	NOC	CBO	RFC	LCOM
LSc	0.001	0.008	0.017	0.002	0.008	0.001	0.0	0.018	0.001
LC	0.159	0.302	0.021	0.183	0.002	0.001	0.003	0.349	0.167
SLOL	0.078	0.157	0.008	0.082	0.004	0.0	0.002	0.174	0.031
LMFV	0.025	0.069	0.004	0.029	0.002	0.0	0.0	0.078	0.004
LMLV	0.052	0.087	0.003	0.053	0.002	0.0	0.0	0.089	0.011
UTQ	0.089	0.177	0.003	0.108	0.0	0.0	0.0	0.156	0.115
LSE	0.028	0.045	0.0	0.026	0.003	0.0	0.001	0.04	0.009

Table A.17: R^2-values for the Machine Learning-project

	CC	SLOC	CD	WMC	DIT	NOC	CBO	RFC	LCOM
LSc	0.006	0.012	0.0	0.001	0.001	0.004	0.0	0.022	0.001
LC	0.312	0.368	0.003	0.13	0.0	0.002	0.077	0.432	0.152
SLOL	0.169	0.223	0.005	0.056	0.001	0.002	0.048	0.281	0.085
LMFV	0.089	0.135	0.007	0.013	0.005	0.001	0.027	0.167	0.018
LMLV	0.121	0.149	0.008	0.034	0.003	0.002	0.031	0.188	0.056
UTQ	0.001	0.001	0.0	0.001	0.0	0.0	0.0	0.002	0.0
LSE	0.07	0.078	0.003	0.018	0.0	0.001	0.012	0.09	0.041

Table A.18: R^2-values for the Humanizer-project

	CC	SLOC	CD	WMC	DIT	NOC	CBO	RFC	LCOM
LSc	0.003	0.002	0.0	0.002	0.005	0.001	0.005	0.005	0.0
LC	0.002	0.0	0.001	0.001	0.002	0.001	0.004	0.006	0.0
SLOL	0.003	0.0	0.0	0.002	0.005	0.001	0.004	0.007	0.0
LMFV	0.001	0.0	0.01	0.002	0.0	0.0	0.001	0.001	0.001
LMLV	0.032	0.002	0.002	0.02	0.0	0.001	0.001	0.0	0.001
UTQ	0.0	0.0	0.0	0.0	0.021	0.0	0.003	0.0	0.001
LSE									

Table A.19: R^2-values for the dnSpy-project

	CC	SLOC	CD	WMC	DIT	NOC	CBO	RFC	LCOM
LSc	0.0	0.001	0.013	0.0	0.009	0.001	0.0	0.003	0.0
LC	0.085	0.077	0.009	0.084	0.001	0.0	0.014	0.136	0.046
SLOL	0.106	0.097	0.008	0.11	0.0	0.0	0.017	0.156	0.05
LMFV	0.012	0.014	0.002	0.012	0.0	0.0	0.003	0.02	0.004
LMLV	0.054	0.049	0.002	0.061	0.0	0.0	0.008	0.076	0.018
UTQ	0.007	0.003	0.0	0.004	0.0	0.0	0.0	0.003	0.0
LSE	0.037	0.035	0.001	0.039	0.001	0.0	0.002	0.05	0.016

Table A.20: R^2-values for the Jellyfin-project

	CC	SLOC	CD	WMC	DIT	NOC	CBO	RFC	LCOM
LSc	0.022	0.034	0.005	0.029	0.002	0.0	0.016	0.048	0.012
LC	0.364	0.513	0.006	0.443	0.0	0.008	0.136	0.605	0.514
SLOL	0.468	0.606	0.004	0.518	0.0	0.005	0.083	0.675	0.5
LMFV	0.199	0.289	0.002	0.242	0.0	0.0	0.035	0.347	0.313
LMLV	0.377	0.46	0.002	0.381	0.002	0.001	0.027	0.505	0.282
UTQ	0.132	0.211	0.003	0.169	0.0	0.009	0.1	0.239	0.209
LSE	0.232	0.281	0.0	0.243	0.001	0.002	0.017	0.323	0.262

Table A.21: R^2-values for the ILSpy-project

	CC	SLOC	CD	WMC	DIT	NOC	CBO	RFC	LCOM
LSc	0.0	0.0	0.002	0.0	0.0	0.001	0.002	0.0	0.0
LC	0.094	0.133	0.002	0.079	0.003	0.0	0.004	0.095	0.004
SLOL	0.055	0.076	0.0	0.047	0.001	0.001	0.001	0.058	0.002
LMFV	0.067	0.045	0.002	0.066	0.001	0.0	0.002	0.038	0.001
LMLV	0.019	0.016	0.0	0.019	0.0	0.0	0.0	0.018	0.0
UTQ	0.065	0.046	0.005	0.066	0.001	0.0	0.0	0.03	0.002
LSE	0.001	0.001	0.001	0.001	0.003	0.0	0.0	0.001	0.0

Table A.22: R^2-values for the OpenRA-project

	CC	SLOC	CD	WMC	DIT	NOC	CBO	RFC	LCOM
LSc	0.051	0.072	0.008	0.026	0.017	0.0	0.023	0.085	0.0
LC	0.29	0.376	0.006	0.227	0.001	0.0	0.076	0.392	0.021
SLOL	0.221	0.321	0.005	0.156	0.003	0.0	0.047	0.31	0.008
LMFV	0.216	0.277	0.003	0.149	0.004	0.0	0.042	0.275	0.004
LMLV	0.113	0.189	0.002	0.07	0.004	0.0	0.026	0.186	0.002
UTQ	0.204	0.175	0.017	0.21	0.002	0.0	0.049	0.226	0.032
LSE	0.084	0.119	0.0	0.037	0.002	0.0	0.01	0.1	0.001