Radboud University Nijmegen

Institute for Computing and Information Sciences

Functional purity as a code quality metric in multi-paradigm languages

Master thesis Software Science

Author:
Bjorn Jacobs

University Supervisor:
Dr C.L.M. Kop

Supervisor Info Support	Jan-Jelle Kester MSc.
Process supervisor	Marieke Keurntjes
Student number	s1030650
Start date	31-01-2022
End date	01-07-2022
Version	1.6

August 2022

Abstract

In software engineering, there is a focus on creating quality code. However, what exactly quality code is and how this is measured is not a trivial question. There are software metrics that try to capture this quality. These metrics can be programming language agnostic or focused on a specific programming paradigm like object-oriented or functional. In recent years more traditional object-oriented languages introduced more functional features. Recent research evaluated how well existing object-oriented metrics work on this new multi-paradigm code. Furthermore, new metrics have been defined with a focus on multi-paradigm code. This thesis will continue with this research by using one of the fundamental principles of functional programming, namely functional purity, as a metric. To achieve this we need a way to calculate a purity metric from csharp code. For this, we combined multiple methods from different studies to create a purity metric that captures a function's purity. We evaluated this purity metric against existing object-oriented and functional metrics. In our testing, it performed better at predicting error-prone code than existing object oriented or functional metrics.

Acknowledgements

First of all, I am very grateful to my supervisor at Info Support Jan-Jelle for providing the starting point, guidance and insightful discussions that were the basis for this thesis. Furthermore, I would like to thank my professor Cynthia for the guidance and feedback that ensured it was up to standards.

Secondly, I want to thank Info Support for providing me with the opportunity to make use of their knowledge and resources during my thesis. In particular Marieke for providing me with guidance in the non-technical aspects and ensuring I had all the resources I needed to complete my thesis.

Tenslotte wil ik mijn ouders bedanken voor hun steun gedurende mijn gehele academische carrière.

$\sim Bjorn$

Contents

1 Introduction

In software engineering, there is a focus on creating quality code that is reliable and maintainable. There are several steps a development team can take to achieve this goal. One of these steps is using code quality metrics. These aim to give an objective evaluation of the code. These metrics are available for most popular object-oriented languages like csharp [4], Java [10] and functional languages like Haskell [33] [34]. These code quality metrics are often available in tools like IDEs and SonarQube.

1.1 Problem statement

In previous years, OOP programming languages like csharp introduced more functional features with each release. As a result, there is an increase in mixed paradigm code. Even though there are metrics for object-oriented and functional paradigms, these fall short when the two paradigms are combined in a single code base [39].

In previous research, there have been attempts to measure the effectiveness of existing metrics in scenarios where functional code is used in an object-oriented language [20]. Further research sought to define new metrics that focus on a combination of OOP and FP to improve on existing metrics [39] [19]. These studies were conducted on the programming languages csharp and Scala respectively.

1.2 Research goal

This thesis aims to further research code quality metrics in multi-paradigm languages. We do this in two ways. The first is to define a novel metric that is based on the idea of a pure function. Functional purity is one of the essential principles of functional programming. A function is pure when it is deterministic and side effect free. In pure functional languages, this is enforced by the compiler. However, in multi-paradigm languages, this is not enforced at any stage. Even though an attribute in csharp can be used to mark if a method is pure, this attribute is an artefact of the code contracts framework [23] and is not enforced by any analysis tool [24]. As a side effect of this approach, we can further validate the metrics aimed at multi-paradigm code in csharp from previous studies.

1.2.1 Research questions

In this study, we will define new metrics that are based on the functional purity of multi-paradigm code and compare these to previous code quality metrics. As a baseline model, we will use the results from previous research on the topic by Zuilof [39] and Konings [19]. To achieve the research goal we will answer the following questions:

RQ: To what extent can functional purity be used as a code quality metric in an object-oriented language like csharp?

- RQ1: How can the concept of functional purity be used as a software quality metric in an object-oriented language?
- RQ2: How well do the purity metrics predict error-proneness?
- RQ3: How well does the purity metric perform in the different types of functions in csharp?
- RQ4: To what extent does combining existing metrics with the purity metric improve the error-proneness prediction?

1.3 Approach

To introduce a metric that is based on functional purity, we first have to know the purity of a function in an object-oriented language. Previous research focuses on statically analysing code to determine if it is pure [31] [28]. There are different ways to achieve the same analysis. From a high-level analysis of the code itself to a low-level analysis of the generated byte code. Our approach uses the Roslyn framework and is aimed at a higher-level analysis. From this analysis, we know to what degree a function is pure. We do this by capturing what and how much impure behaviour a function exhibits.

When we have defined our metric, we need to validate it. We do this by calculating the metrics for error-prone and regular code. When we have these metrics, we define a model that aims to predict how error-prone a function is given a set of metrics. This way, we can compare how suitable different metrics are at predicting error-prone code.

1.4 Contributions

The contributions of this thesis are, firstly, an algorithm and implementation that is able to give a detailed overview of how pure each function is. This implementation uses the Roslyn compiler platform and gives feedback to the programmer on the purity of a function. Furthermore, we provide a proof of concept implementation of how this analysis can be used in build tools.

Secondly, we provide an analysis of how effective the purity of function can be used as an error-proneness indicator. This is done by finding a correlation between the degree of purity of a function and if the function has received bug fixes in the past.

Lastly, we provide a model that predicts how error-prone a function is based on the purity of the function. Furthermore, the dataset and the code that we used to create this model. As well as all the other code that was used throughout this study is available on Github at:
https://github.com/bjornjacobs/PurityCodeQualityMetrics

1.5 Outline

We start this thesis with the preliminary sections that give an introduction to code quality metrics and function concepts in csharp. Next, we go over how our purity algorithm works. After that, we have the methods and validation sections. Next, we present our results. Then we have a section that gives the details of our implementation. And lastly, we give our conclusion where we answer the research questions.

2 Software metrics

As the name indicates, software metrics attempt to quantify some aspects of a software project. One of the earliest metrics is from Halstead in 1977 [13] which measures how many operators and operands a program contains and deduces metrics from that. For example, what the program length and complexity is. Although, at the time, these metrics were innovative, they are not widely used in practice. On the other hand, at the same time, Mccabe introduced cyclomatic complexity [22] which is still widely used. Over the years, other metrics have been developed that are aimed at more specific paradigms.

2.1 Product and process metrics

In the effort of improving the quality of a piece of software different metrics can be used. These are often categorized into two types. Firstly, process metrics, these concern the process of making a piece of software and measure, among others, the number of features delivered and the amount of time expended by employees. Secondly, product metrics, these measure various aspects of a piece of software like the quality and the size of the product. In this thesis, we will be focussing on the latter. While product metrics can measure any aspect of a software product that is quantifiable, in this thesis we focus on software quality. These metrics try to quantify the maintainability and error-proneness of the code.

2.1.1 Existing standard

The ISO 25010 standard [16] defines a model for software product quality. The model includes characteristics from among others: functionality, security, maintainability, reliability, etc. This model is a guideline that can be used for any software project. However, there are no specific measurements provided which one can use to test if their software project conforms to these guidelines.

2.2 Quality Metrics

To quantify code quality, one would define metrics that objectively review the code. Different metric suites have been proposed over the years. These consist of programming language and paradigm agnostic metrics such as lines of source code and comment density. However, most of the time, metrics are aimed at paradigm-specific constructs. For object-oriented programming, one of the most widely used suites is from Chidamber et al. [5]. This, among others, contains depth of inheritance (DIT) tree and coupling between objects (CBO). For functional programming, metrics have been defined and evaluated by Ryder et al. [34]. Different metrics for a combination of OOP and FP structures have been defined and validated by Zuilhof [39] and Konings [19]. These quality metrics expand on existing work by tackling functional features in OOP or mixed paradigm languages. Later in this section we introduce all metrics that are relevant. In this thesis, we are expanding on this research by introducing a novel metric based on functional purity in the object-oriented language csharp.

Agnostic metrics

- **Cyclomatic complexity (CC):** Measures the number of linearly-independent paths through a program module [22].
- **Source lines of code (SLOC):** The number of lines the source code spans. Excluding comments and whitespace

- **Comment density (CD):** The ratio of comment over the total lines of code.

Object oriented metrics

- **Weighted methods per class(WMC):** Summation of a metric in all methods in a class. Suggested is to use Cyclomatic complexity [5].
- **Depth of inheritance tree(DIT):** Depth of inheritance of the class. In cases involving multiple inheritance, the DIT will be the maximum length from the node to the root of the tree [5].
- **Response for class (RFC):** The cardinality of the response set for class. The response set is a set of methods that can potentially be executed in response to a message received. [5].
- **Number of children of class (NOC):** The number of children that inherit from this class.
- **Coupling between object classes (CBO):** is a count of the number of other classes to which it is coupled [5].
- **Lack of cohesion of methods (LCOM):** The goal of this measure is to quantify the (lack of) coupling between methods and instance variables in a class. The definition described by Chidamber et. al. [5] is:

  > Consider a class C_1 with n methods used by method M_I. $M_1, M_2, ..., M_n$.
  > Let $\{I_n\}$ = set of instance variables used by method M_i.
  > There are r such sets $\{I_1\}, \cdots, \{I_n\}$. Let $P = \{(I_i, I_j) \mid I_i \cap\ I_j = \emptyset\}$ and $Q = \{(I_i, I_j) \mid I_i \cap I_j \neq \emptyset\}$. If all n sets $\{I_1\}, \cdots, \{I_n\}$ are $\emptyset$ then let $P = \emptyset$.

$$\begin{aligned} \text{LCOM} &= |P| - |Q|, \text{ if } |P| > |Q| \\ &= 0 \text{ otherwise} \end{aligned}$$

Functional metrics

- **Number of lambda functions used (LC):** The number of lambda functions [39].
- **Source lines of lambda (SLOL):** Counts the number of lines containing lambda functions [39]..
- **Lambda score (LSc):** The lambda score is the ratio of lines containing lambda functions to the number of source codelines. [39]
- **Number of lambda functions using mutable field variables in a class (LMFV):** Counts the number of lambda functions using variables defined outside the method scope [39]
- **Lambda Local Variable Usages (LMLV):** Counts the number of lambda functions using variables defined inside the method scope (but outside the lambda itself) [39]
- **Number Of Lambda Functions With Side Effects Used In A Class (LSE):** Counts the number of lambda functions using side-effects. The usage of side-effects means the lambdas are not pure. [39]

- **Number Of Unterminated Collection Queries In A Class (UTQ):** An unterminated collection is an enumerable where the iterator is not yet used [39].

2.3 Challenges

Formulating new metrics can be trivial but validating if they provide value in the development process is more challenging. Here we face the problem that we have to know what code is error-prone and what code is 'good' code. With such a dataset we can define a model using one or more metrics that are derived from this code. Consequently, different metrics can be validated and compared based on the model's performance. However, this requires annotated datasets which are harder to create when the metric tries to measure concepts like code quality and error-proneness. Recent studies [20][39][19] used git repositories with issue trackers as a way to link bug fixes with code changes. One can consider several factors to determine if an issue is a bug fix, for example the title and the tags. After collecting a set of bug fix commits, we can use the git change log to see what code has been removed and what code has been added. The assumption is made that the removed code is error-prone and the added code is 'good'.

3 Functional purity

In this section, we will introduce the idea of functional purity and how it relates to csharp. To do this, we first go over what functional purity is and what functional concepts are implemented in csharp. Lastly, we give an overview of the challenges for determining purity in an object-oriented language like csharp.

3.1 Pure functions

A pure function is a function that adheres to two constraints. Firstly, it has no side effects, which means that it does not alter any state outside the function scope. This can be altering local fields, writing to a global variable or even outputting something to the screen. Secondly, the function needs to be deterministic, which means it always evaluates to the same output for the same input. In practice, this means only reading data from function parameters and not from fields or properties. Moreover, purity also propagates trough the call graph. This means that when a pure function calls an impure function, it also becomes impure.

In functional languages like Haskell, functional purity is ensured by the compiler. However, in OOP languages like csharp, the compiler does not validate the purity of a function. One of the advantages of a pure function is that it is more straightforward to reason about because there are no external factors that determine how a function behaves. This has the benefit that it makes testing easy and allows for delayed execution without any problems. Furthermore, a pure function can be parallelized without worrying about the order of execution. However, making a whole OOP application functionally pure is not practical as one would want to have side effects in a program. For example, outputting some data to the screen or non-deterministic behaviour like reading data from a database. Moreover, making a fully pure program in an object-oriented language would mean not using any of the design principles that make a language object-oriented.

3.1.1 Examples of pure and impure functions

In this subsection, we will go over some examples of pure and impure functions in csharp. In listing 1, two pure functions are shown. They are pure because only parameters are accessed, and no state outside of the function is altered. These functions can be executed however often and in whatever order without changing the output.

```
int Addition(int x, int y)
{
    return x + y;
}

bool IsOlder(Person a, Person b)
{
    return a.Age > b.Age;
}
```

Listing 1: Example of pure functions

In listing 2, two non-deterministic functions are shown. In the first function, the random class is used, which is non-deterministic if no seed is provided. The second function reads from state outside of the function. In this case that state is the file system.

```
int RollDice()
{
    return new Random().Next(6);
}

int CountErrorsInLog(string logFilePath)
{
    string[] logs = File.ReadAllLines(logFilePath);
    return logs.Count(x => x.StartsWith("ERROR");
}
```

Listing 2: Example of non-deterministic functions

In listing 3 two functions with side effects are shown. These functions both alter state in some way. The first function alters the state of the parameters and a local property. The second function changes the state of the file system by writing to a file.

```
void AddPlayer(Player p, GameState s)
{
    s.Players.Add(p);
    PlayerCount++;
}

int Log(string file, int level, string text)
{
    File.AppendToFile(file, level + ": " + text);
}
```

Listing 3: Example of functions with side effects

3.2 Functional concepts in csharp

Csharp supports various functional features like first-class, higher-order and lambda functions. In newer versions of csharp pattern matching and immutable types (records) are also supported. Probably the most used feature that is inspired by functional programming is the LINQ library. This provides functions that are often used in functional programming, albeit with a different name. For example, select (map), where (filter), aggregate (fold), etc. In all these functions, the programmer passes a function object as a parameter. These operations are also lazy, meaning they aren't executed until the actual values are requested. This has the advantage that no calculation is done unless it is needed, but this can lead to unexpected behaviour if the function that is passed as an argument is not pure.

```
List<int> odd = naturalNumbers.Where(x => x % 2 == 0)
            .Select(x => x + 1)
            .ToList(); //Values are materialized
```

Listing 4: Example of LINQ functions in csharp

3.3 Measuring purity in object-oriented languages

In functional languages like Haskell, it is impossible to create side effects as no variables outside of functions exist. Furthermore, no reference types are exposed to the programmer. However, object-oriented languages are designed to use fields and reference types. Consequently, this is the main challenge one faces when determining the purity of a function in an object-oriented language.

Various solutions try to solve the problem. Pearce (Jpure) [32] defines the problem at a high level and checks the intermediate language (IL) for reads and assignments. Jpure exists in two parts. The first is a purity inference algorithm, and the second is a purity checker. The purity inference is a slower method that can automatically assign a purity level to a function. The purity checker is a fast method that can check if an entire program is correctly annotated. The advantage of this approach is that the inference can happen once and is valid until a function changes. After that, it is fast to check the purity of a program as only the changed functions need to be inferred again.

Another solution proposed by Salcianu and Rinard [35], is adapted from pointer and escape analysis on byte code. In this research, the analyser examines the generated byte code. Using this method a function is pure if it does not mutate any memory location that exists in the program state right before the invocation of the function. The advantage of this method is that it is more generalisable as it does not have to deal with specific code structures specific to a programming language. Furthermore, this method is easy to reason about as it eliminates edge cases that higher-level code constructs may introduce.

One more approach is to define and check everything at a high level. However, checking purity at programming language level does have its disadvantages. Namely, the edge cases have to be handled, and the approach is specific to the constructs of a programming language. An example of such a solution is from Österberg (CsPurity) [31]. In their thesis, a technique and implementation for csharp are given. This method works by following a ten-step checklist, see appendix A, that handles the cases that make functions impure. The downside is that this method is more work to implement and less generalisable compared to the previous described methods. However, as a result, the algorithm has more information about what made the function impure and can assign different purity levels for different cases.

3.3.1 Purity levels

The research and implementation from Österberg [31] is a starting point for our novel implementation. They define a way to measure purity in csharp using a technique defined at a programming language level. Österberg also provides a partial implementation that is based on the checklist. However, the main reason why this method was chosen is that it can assign each function with a different purity level. The different purity level that CsPurity uses can be found in table 1. A function gets assigned one purity level. If multiple levels are applicable the highest one gets selected.

1	Pure	No side effects and deterministic
2	Impure throws exception	Throws and exception in function
3	Parametrically impure	Modifies reference type that is passed a parameter
4	Locally Impure	Reads or writes to local state
5	Impure	Reads or writes from global state
6	Unknown	Program can't determine the purity of all dependencies

Table 1: Purity levels as defined by Österberg

3.3.2 Non-Fresh objects

In object-oriented languages like csharp, objects/ classes are reference types. This means that modifying a field or property on these can be considered a side effect as it affects part of the system outside the current function. However, an object can be 'fresh', meaning it was just created. In reality, this means that the only reference to that object exists in that function scope. Consequently, a 'fresh' reference type can be modified without a side effect.

The JPure purity system [32] handles this by annotating functions with the @Fresh Java annotation, meaning that the object returned is fresh. Moreover, all objects that the constructor of a class returns are also considered 'fresh'. The next step is to use data flow analysis to check if a local variable in a function contains a fresh or non-fresh object.

3.4 Purity metric

When calculating purity in an object-oriented language, an impure function is easy to make. Consequently, all functions that depend on it are also impure. As a result, most of the methods in a program are marked as impure. This limits how effective the concept of purity can be used as a metric. To solve this problem, we define a novel method that does not give functions a direct rating. Instead, for each function we record its 'purity violations', see table 2 for the list. This allows for a flexible metric that gives more insight into the purity of each function. A function in csharp can be a local function, class method or a lambda function.

Research from Landkroon [20] and Zuilhof [39] already focussed on side effects in lambda functions. However, these metrics are very minimal in terms of analysis. Both only check if a lambda function reads to or assigns to a non-fresh variable and cannot check dependencies. Our purity metric method also analyses calls, meaning that it knows when a lambda method calls a non-pure method. Furthermore, the purity metric is not limited to lambda functions but uses every form of function/method in csharp.

The metric can be calculated in different ways; see section 6.2. However, they are based on the same data. Namely, a vector of 2-tuples (d, v) where d is the distance in the call graph and v is the violation type. By saving the distance, the metric can differentiate between violations committed by the function and violations of its dependencies.

Reads Local State	A local field or property is read from.
Writes local state	A local field or property is written to.
Reads global state	A static variable is read from. This also includes reading from persistent storage. But Constant values e.g. readonly or const value types are excluded.
Writes global state	A static variable is written to. This also includes writing to persistent storage.
Modifies Parameters	A parameter is modified, only applicable if the parameter is a reference type.
Modifies non-fresh object	Modifies an object that exists in the scope of the method but is not fresh. Meaning that there exists a reference to the object outside the function.

Table 2: Novel purity violations

3.5 High level design

Österberg provided a partial implementation, called CsPurity, that follows the checklist he defined. However, the only parts that are implemented are steps (3) reading/ writing to static fields, (9) throwing an exception and (10) checking pre-marked impure methods. Furthermore, only the Abstract Syntax Tree (AST) is analysed, meaning that it cannot properly construct a dependency graph as this does not contain the semantic meaning of nodes that is needed. Furthermore, the CsPurity implementation eagerly marks functions as impure when one step of the checklist detects impure behaviour.

When designing our implementation, we made sure that the output of our analyser contains all the information we need to determine the purity metric. For this we took inspiration from the JPure design, which splits the purity inference and purity calculation into two parts. In our design, we also split the analysis into two parts. The first part analyses each function separately and saves all the direct violations it commits and all the functions it depends on. The second part uses this information to calculate the indirect violations and output the vector of 2-tuples with the distance and type for each violation. This has the advantage that we know why a function is impure. Moreover, this also allows us to detect and manually input missing data, see section 4.3.1 for more information.

3.5.1 Violations example

In figure 1 we see the violations of an example program and in table 3 what the output would be of our implementation for this example program.

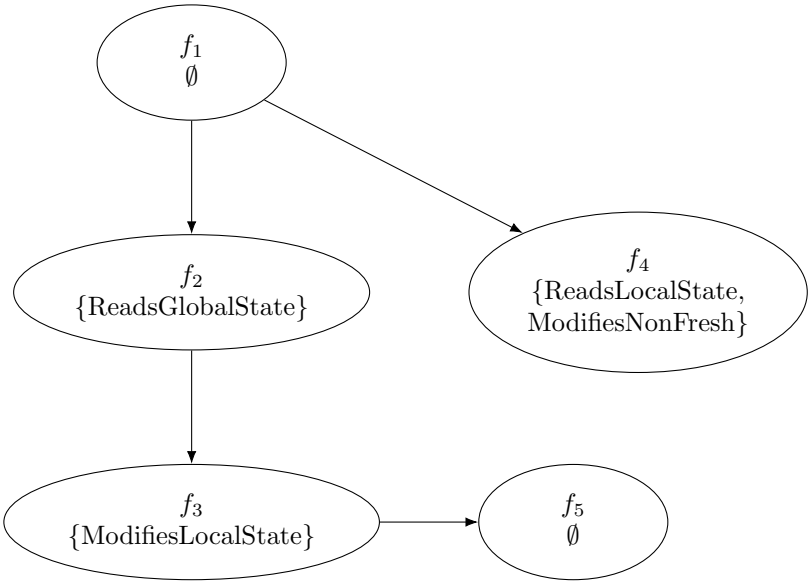

Figure 1: Example of call graph outputted by step 1. Each node represents a function and contains its direct violations

Function	**Rating**	**Violations with distance**
f_1	Impure	(ReadsGlobalState, 2) (ModifiesLocalState, 3) (ReadsLocalStat, 2 (ModifiesNonFresh, 2))
f_2	Impure	(ReadGlobalSate, 1) (ModifiesLocalState, 2)
f_3	LocallyImpure	(ModifiesLocalSate, 1)
f_4	LocallyImpure	(ReadsLocalState, 1) (ModifiesNonFresh, 1)
f_5	Pure	-

Table 3: Output of step 2 based on the example given in figure 1

4 Methods

In this section we will go over the methods we use to collect, validate and evaluate our data. We start off by introducing the validation methodology. In the following sections we are going to look at how we implement the requirements for this validation methodology.

4.1 Briand's methodology

A frequently used methodology to validate software metrics is Briand's empirical validation method [2]. This tries to answer the following question: "is the measure useful in a particular development environment, considering a given purpose in defining the measure and a modeller's viewpoint?". In this question, a measure is useful when it relates a measure of an external attribute to the object of the study. In this thesis this is: relating the software metrics, e.g. the purity of a function, to the error-proneness of a piece of code. Furthermore, the purpose of defining the measure and modeller's viewpoint is, in our case, to reduce the number of errors in the program by detecting code that has a high chance of causing an error. We do this with the purpose of validating our metrics (internal attributes). Because if we do not do this, we can't say that there is a relationship between a software metric and the error-proneness we want to predict. Briand's validation methodology defines the following requirements to be met.

1. A collection of the software metrics for a software product is needed. These are the metrics that are going to be tested.
2. Identification of what parts of software product are error-prone. Furthermore, we have to decide in what detail we want to identify error-prone code. This can be a whole file, classes, methods, etc. This is used as a dataset to find a relation between a metric and code that is marked as error-prone.
3. A suitable analysis tool to find a relationship between our metrics and if a piece of code is error-prone.

4.2 Landkroon's methodology

In Briand's methodology, error-prone classes are marked when a fault is found in them at any point in time. This results in a data set where each class has a count of how many errors were constituted in that piece of code. After this, the latest version of the code is used to calculate the code metrics. The critique Landkroon [20] has on this is that due to bug fixes or refactoring, an entire class can change after a bug is fixed but is still marked as error-prone. This means that the code metrics are calculated on entirely different code than when the issue was found. Moreover, this shortcoming is more noticeable as projects get a longer time span. Because we do our analysis on open source projects that have been in development for years we chose to use Landkroon's methodology.

Landkroon adapted Briand's methodology to analyse the code at the point right before a bug is fixed. This way we have code metrics for error-prone code when the bug was still in the code. Finally, all the code in the latest version of the code base is considered 'good' code.

4.2.1 Algorithm implementation

The algorithm's pseudo-code can be seen in algorithm 1. In this subsection, we will give a short textual representation. First, it collects all the commits and issues from

the project. Next, it finds out which issues can be classified as a bug and finds the corresponding commits. Then it defines an empty list where we will store all the metrics. When it has everything set up, it can loop through each commit linked to a bug fix and find out which methods changed in the bug fix. When it has that information, it checks out the commit and calculates the metrics for the changed functions. It then does the same for each parent commit. Now it has the metrics for a function before and after the bug fix. Then, it adds these metrics to the metric list. Finally, it saves the metrics to the disk so we can analyse them later.

Algorithm 1 Implementation of Landkroon method

```
commits                                          ▷ Loaded from git repository
issues                                           ▷ Loaded from github issues api
bugfixes ← isBugFix(issues)   ▷ Determined by looking at the bug labels and title
bugfixCommits ← commits ∩ bugfixes   ▷ Every commit that has an associated fix
metrics ← []
for b ∈ bugfixCommits do
    for p ∈ parents(b) do
        relevantfunc ← diffs(b, p)                ▷ Functions that changed in commit
        oldmetrics ← calc_metrics(p, relevantfunc)   ▷ Metrics where bug is present
        newmetrics ← calc_metrics(b, relevantfunc)   ▷ Metrics where bug is fixed
        merged ← merge(oldmetrics, newmetrics)       ▷ Merge metrics per function
        add(metrics, merged)                  ▷ Add the calculated metrics to the list
    end for
end for
writeToDisk(metrics)
```

4.3 Collecting data

As mentioned in section 2.3 we are going to use git repositories and issue trackers to detect error-prone code. Issues in these trackers contain tags that give an indication of whether it is a bug report. When an issue is solved the code changes are linked to the issue in the form of a pull request. The benefit of using git is that we can go back in history to the code before and after the bug fix. Furthermore, we know precisely what lines of code have been changed. Meaning that we can choose to only analyse that code. This has the advantage that we can increase the accuracy of the data compared to analysing the whole program before and after the bug fix.

The data that we use is acquired from public GitHub repositories, see figure 4 for the list. For each repository, we automatically collect the following information using a git client in csharp; (1) git commit history; this is used to go back to the code snapshot before and after a bug fix. This data is retrieved by cloning the git repository from GitHub. And (2) GitHub issues; this is used to determine which commits are bug fixes. This data is retrieved by mining the GitHub API. To determine if an issue is a bug fix, we look at the title and the labels of the issue. If these contain a hint that the issue is about a bug we include it in the bug fixes list.

Lastly, we determine what code is changed by looking at the git diff log. These contain information about which code lines were removed and which code is added. We now know what lines are changed in what files. We extract this information from the diff log using a regular expression and then cross-reference these lines with code to find out which methods were changed. In order to do this, we use a local git repository where we check out the commit we are currently analysing.

In most code quality research on OOP metrics, there is a focus on classes [5] [39]. We choose to focus on methods. This is done because our purity is calculated on a per-method basis. This also means that we look at local and lambda functions as separate functions. For this reason, we choose to collect data on this level.

Name	# Bug fixes	URL
ML for .NET	45	https://github.com/dotnet/machinelearning [26]
Identity server	59	https://github.com/IdentityServer/IdentityServer4 [6]
AKKA.NET	240	https://github.com/akkadotnet/akka.net [1]
Jellyfin	199	https://github.com/jellyfin/jellyfin [17]
OpenRA	227	https://github.com/OpenRA/OpenRA [30]
ILSpy	374	https://github.com/icsharpcode/ILSpy [15]
Humanizer	27	https://github.com/Humanizr/Humanizer [14]
MoreLinq	69	https://github.com/morelinq/MoreLINQ [27]
Reactive	13	https://github.com/dotnet/reactive [25]
Shadowsocks	30	https://github.com/shadowsocks/shadowsocks-windows [37]
Resharper	41	https://github.com/JetBrains/resharper-unity [18]
Roslyn	2409	https://github.com/dotnet/roslyn [7]

Table 4: Open source projects used. Number of bug fixes means the amount of issues found that can be labeled as a bug. Commit hash and date of retrieving issues can be found in appendix B

4.3.1 Unknown data

When determining the purity of a function we also analyse the dependencies. Sometimes these dependencies can't be analysed. This can be because the function is from an outside dependency like a nuget-package or the standard library. Furthermore, sometimes the compilation has an error and the compiler can't determine which method is being referenced.

To solve this problem, we input this data by hand using the interface shown in figure 2. This allows us to manually input what violations a function has and whether the return value is fresh. This judgement is then saved to a database for future analysis. This partially solves the problem of unknown methods and because we save the judgments for the future we have to manually input less data as we analyse more code.

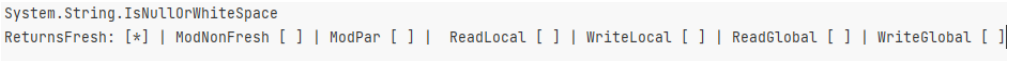

Figure 2: Input screen unknown function

The downside of this method is that a human has to guess what violations a function commits based on the function name. For the standard library and other dependencies the programmer is familiar with, we expect that the results are fairly precise. But for unknown code we expect that the programmer doesn't know any more than the machine. When collecting the actual data we found out that marking every method is too time consuming. Moreover, for most domain specific methods we could not deduce from the method signature alone what possible violations it contained. For this reason we opted to only mark methods from the standard library by hand. e.g. methods that are in the namespace 'system.*'. Other external dependency are marked as having no violations. This problem could be solved in the future by analysing the byte code.

4.4 Analysis method

From the collected data, we know what the properties of a function are, e.g. the code metrics and purity violations, and if the function is considered error-prone. Our goal is to predict whether a function is error-prone based on its properties. To do this, we need to define a model that gives such a prediction. Like previous studies [3][20][39][19] before us, we will use logistic regression. This will train a model to predict a dependent variable (if a function is error-prone). using one or more independent variables. In our case these independent variables are the code metrics and purity violations. This training works by inputting the cases where the dependent variable is known. From these cases a model is built where the independent variables are the input and the output is a prediction if the dependent variable is true or false. Because our purity metric and baseline models consist of multiple parameters, e.g. the different purity violations, we will have multiple independent variables. This means that we will have to use multivariate logistic regression, see figure 3.

$$P(faulty = 1) = \frac{e^{\beta_0+\beta_1 X_1+\beta_2 X_2+\ldots+\beta_i X_i}}{1 + e^{\beta_0+\beta_1 X_1+\beta_2 X_2+\ldots+\beta_i X_i}} \quad (1)$$

Figure 3: Logistical regression equation.

4.5 Data split

When creating our models we have to split the data into training, validation and test sets. The training set is used to fit the model, the validation set is used to tune the hyper parameters which in our case is the purity function. And the test set which is used to measure the performance of the model. To split this data we aimed for 70% training data and 15% for validation and test data. Moreover, the data for one project can only exist in either the training, validation or test dataset. This makes sure the test result are generalizable. The data split is shown in table 5. Note that the Roslyn project had 145.956 data points but a sample of 40.000 was taken. This was done to not overfit the model on Roslyn samples. This split resulted in 67% training 15% validation and 18% test data with each four projects.

Project	**Total**	**Faulty**	**Non-Faulty**	**Dataset**
Roslyn	40.000	797	39.203	Train
machinelearning	22.908	1.485	21.423	Test
jellyfin	9.087	1.576	7.511	Test
akka.net	40.278	25.264	15.014	Train
IdentityServer4	1.632	568	1.064	Test
ILSpy	18868	7.985	10.883	Train
MoreLINQ	3.860	1.143	2.717	Test
Humanizer	2.838	90	2.748	Validation
reactive	35.428	238	35.190	Train
OpenRA	24.682	11.705	12.977	Validation
shadowsocks-windows	1.032	490	542	Validation
resharper-unity	285	218	67	Validation

Table 5: Data split

4.6 Baseline models

If we want to know if our metric is an improvement over already existing metrics we need a baseline. Previous research from Zuilhof [39] already defined a model for predicting error proneness. This model is defined using multiple metrics and multivariate regression. In our study, we will compare the results to this model and the most common object oriented metrics to determine if we made an improvement. The baseline object oriented and function models contain the following metric respectively.

Object-oriented model

- Weighted Methods Per Class (WMC)
- Cyclomatic Complexity (CC)
- Depth Of Inheritance Tree (DIT)
- Lack Of Cohesion Of Methods (LCOM)
- Response For A Class (RFC)
- Comment Density (CD)

Functional model (Zuilhof)

- Lambda score (LSc)
- Number of lambda functions used (LC)
- Source lines of lambda (SLOL)
- Lambda Local Variable Usages (LMLV)
- Number of lambda functions using mutable field variables in a class (LMFV)

4.7 Novel models

In this study we will define multiple models. The first model uses the different violations to determine the best purity metric. This metric is defined using a combination of the violation count, total lines of source code and distance from the current function a violation was established, as defined in section 3.4. When this metric is defined, we will use multivariate regression to create a prediction model. In this model, we can analyse the different coefficients to determine if a type of violation has a positive or negative correlation with predicting error-proneness. Furthermore, because each input in this model is a violation count we can directly compare how strongly corralled a metric is by looking at the coefficients in the model.

The other models are a combination of our purity metric with the existing functional model Zuilhof [39] and the object-oriented model from Chidamber et al. [5]. However, in these models we can not use the coefficients to directly compare the correlation between the metric and the error-proneness prediction. This is because the metrics count different things. For example, lines of source will almost always give a higher value than the number of lambda functions in a method.

Furthermore, to answer research question 3, we will evaluate our purity model per function type. Meaning that we will group the dataset per function type, e.g. lambda, local function and class methods, and evaluate them separately.

4.8 Evaluation

When we have trained our models, we want to measure the performance of each to determine if, and how much of an improvement we made. We use the following metrics to determine the performance of the model. We do this per open source project as well as for all the projects combined. In the evaluation we have four categories that a result can be. (1) true positive (TP) is when a function that is marked as error-prone code is error-prone code, (2) false positive (FP) is when good code is marked as error-prone code, (3) true negative is when good code is marked as good code and (4) false negative (FN) is when error-prone code is marked as good code. See the confusion matrix in figure 4 for a visualisation.

		Predicted label	
		0	1
Actual label	0	True negative (TN)	False positive (FP)
	1	False negative (FN)	True positive (TP)

Figure 4: Confusion matrix

Precision

The precision measures the ratio of true positives over false positives in combination with true positives. In our case this measures how much of the code that our model determines is error-prone is actually error-prone. In other words, the precision is 1 if every code that is marked as error-prone is actually error-prone.

$$Precision = \frac{TP}{TP + FP} \tag{2}$$

Recall

The recall measures the ratio of true positives over true positives in combination with false negatives. In our case this measures how much of the error-prone code our model can retrieve. In other words, the recall is 1 if every error-prone method is retrieved.

$$Recall = \frac{TP}{TP + FN} \tag{3}$$

F-measure

To get one score we can compare between models. For this we need some way to combine precision and recall. One could take the mean of the two but this would result in a relatively high score if one of the two is high and the other is 0. Because our dataset is imbalanced with more good examples than error-prone examples this is easy to achieve by marking every function as not error-prone. The F-measure is a popular metric to use when there is such an imbalance problem [11]. It uses precision and recall, but both need to have a high score to get a high f-score.

$$F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{4}$$

4.9 Validation

In the first part of this section, we talked about *how* the study is conducted. In this subsection, we will give more information about *why* things are done this way. Furthermore, we talk about the threats to validity and how we minimise them.

4.9.1 Data

For the validation of the code metrics, we need a good dataset. However, such a dataset is hard to come by as we need to know what code is classified as error-prone and have the original code to calculate the metrics. Moreover, we are even more limited because we are investigating a more recent trend in a specific programming language. As a result, we had to create a new dataset. The approach for this is to use git repositories and issue trackers. However, the assumption we have to make is that all code changed in a 'bug-fix' commit is error-prone. Furthermore, this process of detecting bug-fix commits is automatic and may include code that is not necessarily error-prone. To minimise this impact, we try to be as accurate as possible in our data collection. We do this by, firstly, using Landkroon's methodology instead of Briand's as this gives us the code at the time of the bug and not that of the final version. And secondly, by only including functions that contain a change instead of the whole file.

Collection

To be able to use the data, we have to collect it. The first question was what projects do we use? For this, we looked at projects selected by previous studies and how they were selected. Here the following requirements were used.

1. The project contains a significant amount of csharp code. In the case of Zuilhof, this was at least 100 csharp classes.
2. The project actively works with the Github issue tracker. This means that the workflow of the project includes using the issue tracker and linking commits/pull requests with issues. We cannot link bug fixes with actual code if this is not done. As a result, we cannot detect what code is error-prone.
3. The project contains a semi-balanced dataset. Meaning that the project contains enough error-prone code that it is statistically significant.

Because our research also focuses on csharp, we decided to use the same projects. Moreover, using the same projects also allows us to directly compare our results with Zuilhof's results. However, some projects were replaced because they are archived or not under active development.

4.9.2 Imbalanced data set

In our final dataset, we have two types of data. The first is a collection of functions analysed as they are in the final code. The second is functions that have changed in a bug-fix. For our data set to be useful, we must have enough functions in each category. Our final dataset had 52,974 error-prone and 233,260 regular functions. As a result, around one in five functions were marked as error-prone. This is acceptable as we can get statistically relevant results in later analysis.

4.9.3 Validating automatic analyser

Analysing csharp syntax tree for violations is complex and contains a lot of edge cases. This means that we can miss violations or wrongly identify a purity violation. The way our algorithm is implemented is still limited. Mainly that it doesn't fully take control flow into account, only a very limited check to see if a local variable is fresh.

Even though our implementation is not perfect, we believe that it covers the most common cases. To ensure that all these cases are correctly covered, we create automatic test cases that ensure that the analyser finds all and only the violations in a function. A test case is created by annotating a csharp method with an attribute that defines what violations exist in that method. For the implementation details see section 5.2.2

5 Implementation

This section discusses the implementation details and the gap between the theory and the actual implementation. Furthermore, we will go in-depth and provide the details for repeatability. All the code and datasets can be found on Github https://github.com/bjornjacobs/PurityCodeQualityMetrics

5.1 Dependencies

In the implementation, we used the following dependencies. We briefly go over each dependency and why we use it.

- **Roslyn:** is the open-source implementation of both the CSharp and Visual Basic compilers with an API surface for building code analysis tools [7]. We use Roslyn for all our code analyses. It enables us to parse and walk the syntax tree. However, more importantly, it also allows us to do semantic analysis. This is important because it enables us to determine the purity violations, as seen in subsection 5.2.
- **LibGit2Sharp:** brings all the might and speed of libgit2, a native Git implementation, to the managed world of .NET [21]. We use LibGit2Sharp for interacting with the local git repositories. This library allows us to read commits, extract information from them, and run git commands like 'reset' and 'checkout'.
- **Zuilhof:** This thesis builds on previous research from Zuilhof [39]. In their study, they also focused on software metrics in csharp. Since they made their code open-source, we used part of that code for the following purposes. Firstly, to calculate the OOP metrics. Secondly, to connect to GitHub and mine the data we needed. And lastly, using his git code as a basis for our implementation. We copied the code from their GitHub and adapted it to work in our implementation.
- **SkLearn:** is a Python module for machine learning built on top of SciPy [36]. We use this library for k-fold cross-validation and fitting the data to a logistical regression model.

5.2 Purity analysis

The purity analyser is responsible for determining a function's violations and constructing the dependency graph. Furthermore, it flattens the dependency graph and calculates the actual violations for each function.

In the implementation, we split into two different classes. The first is the 'PurityAnalyser' which is responsible for the first two steps. The second is the 'PurityCalculator', responsible for the last step. The purity analyser works at a high level by executing the following steps; each of these steps will get their own sub-section where we will further explain the details.

1. Use Roslyn to extract all the syntax trees from a csharp project.
2. Retrieve all the methods, lambdas and local functions from the syntax trees.
3. Determine the direct violations per function.
4. Determine if the return value of each function is fresh.
5. Retrieve all dependencies for each function.
6. Collapse the dependency graph to calculate the indirect violations of each function.

5.2.1 Step 1 & 2: retrieval of functions

We want to extract all functions from a csharp project in our analysis. We can generate a syntax tree from a csharp file using Roslyn. From this syntax-tree we extract all children of type 'MethodDeclarationSyntax', 'LocalFunctionStatementSyntax' and 'LambdaExpressionSyntax'. This gives us all functions. However, lambda and local function syntax usually are children of method declaration syntax. To prevent lambda and local functions from being considered in the body of a method, we define a new extension method that filters out all the elements that are not direct children of that method. See listing 5 for an example. Here we can see that the body of the lambda is inside the body of the method. The extension method filters out everything between .Where(...) as this is is part of the lambda function.

```
public class ExampleClass
{
    private int threshold = 3;

    public int MethodWithLambda(List<int> numbers)
    {                                                   %Method body
        var average = numbers.Where(x => {              %Method and Lambda body
            return x > threshold;                       %Lambda body
        })                                              %Method and Lambda body
        .Average();                                     %Method body
        return average;                                 %Method body
    }
}
```

Listing 5: Example of overlap in function bodies

5.2.2 Step 3: direct violations

When we have our functions and their corresponding syntax tree, we want to know what violations each function commits. We have defined six violations, and each occurs when different requirements are met. See table 6 for an overview. To check these requirements, we extract all identifier ('IdentifierNameSyntax') objects from the syntax tree. In csharp identifiers can chain, for example 'field.Prop1.X', one can choose to take all these variable accesses as violations. However, we keep only the violations for the top-level identifiers because scoping information may be unknown for the identifiers further in the chain. We then use Roslyn to extract the matching symbols for each identifier giving us semantic information. When we have all this information, we check the properties of an identifier and determine if it matches a violation as defined in section 2. A partial list of the properties checked can be found in table 6, in this table we show the requirements that have to met for a violation to be committed. The full code can be found in the class 'IdentifierViolationPolicy'.

Violation	Read	Write	this.*	Para	Ref	Const
Reads local state	Y	N	Y	-	-	N
Modifies local state	N	Y	Y	-	-	-
Reads global state	Y	N	N	N	-	N
Modifies global state	N	Y	N	N	-	-
Modifies parameters	N	Y	N	Y	Y	-
Modifies non fresh	N	Y	N	N	Y	-

Table 6: Properties checked to determine each purity violation. A dash means that it is not applicable (n/a).

To test this we have integration tests using a custom csharp attribute. A csharp attribute is everything between [...] and can be used to annotate a function with extra information, see listing 6 for the [ViolationsTest(..)] attribute. The ViolationsTest attribute contains all the violations a function should contain. When the tests are run all the functions with this attribute are retrieved and validated if and only if all the violations defined in the attribute are committed by the function.

```
[ViolationsTest(PurityViolation.ReadsLocalState, PurityViolation.ModifiesLocalState)]
public int LocallyImpureTest()
{
    _member = 1;
    return _member;
}
```

Listing 6: Example of automatic test case. the attribute defines what violations the function contains

5.2.3 Step 4: fresh objects

The violation 'modifies non-fresh object' is a particular case as it can not always directly be determined. This is because we need to know if local objects are fresh, which is only the case if all the possible objects assigned to a variable are from a fresh source. i.e. a class constructor or a function that returns a fresh object. However, we do not know if a function returns a fresh object at this step. As a result, we can not determine whether the modified non-fresh violation is committed.

The solution is to check this at step 6, when all indirect violations are determined. Nevertheless, at this step, we need to save if this function returns a fresh object and what functions are possible non-fresh violations if they do not return a fresh object.

```
public List<string> ReadLogs(string file)
{
    if(File.Exists(file)){
        return new List<string>(); //A new object is always fresh
    }
    var logs = File.ReadAllLines(file); //Read All lines returns fresh object

    return logs; //Checked if all assignments to logs are fresh
}
```

Listing 7: Example of a function that return a fresh object

5.2.4 Step 5: dependency graph

To determine the indirect violations, e.g. the violations propagated from a function's dependencies, we need to know which functions depend on what other functions. This can be represented in a dependency graph. In this graph, the nodes are functions and the vertices represent a dependency. We find all 'InvocationExpressionSyntax' and 'LambdaExpressionSyntax' in a function's body to build this graph. To connect each function in the later stage, we give each function a unique name. This is constructed in the following format:

'namespace.classname.function_name.<<function_type>>.arguments_types'

In csharp a function with the same name can be overloaded with different argument types. To make sure these functions are unique, we include the argument types. Furthermore, lambdas are also called anonymous functions as they do not have a name. To work around this we give it the name of the parent in combination with a number.

5.2.5 Step 6: indirect violations

Now that we have the direct violations for each function and the call graph, we need to calculate the indirect violations and the distance to the violation. The challenge is to do this in an order that we only have to process each function once. Furthermore, we also need to handle recursion/ cycles in the call graph, as this was one of the problems that Österberg [31] had left for future research.

Our solution to this problem uses Tarjan's algorithm [38]. This algorithm finds all strongly connected components in a graph. In other words, it finds all functions in the call graph that contain mutual recursion. These are called strongly connected components. A strongly connected component is processed as a single unit and given the same violations.

Furthermore, the algorithm outputs the strongly connected components in reverse topological order. When we process the components in this order all the dependencies are either in this component or are already processed. These properties make the implementation simple and $O(n)$ time complexity.

However, to find the distances between nodes in a single component, the Floyd Warshall algorithm is used. This makes the implementation $O(n^3)$ where n is the size of the component. Nevertheless, we do not expect this to be a problem in practice as strongly connected components are usually small.

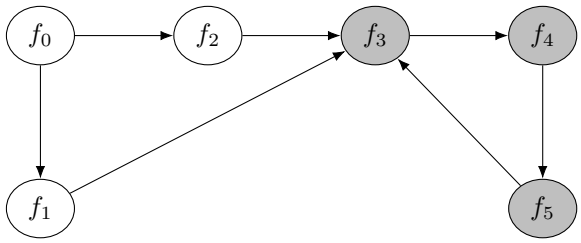

Figure 5: Example of strongly connected component (grey)

5.3 Landkroon implementation

When implementing Landkroon's method, we must traverse a git repository and checkout each commit that is linked to a bug fix. Next, we have to parse the change log

and determine which methods changed. After that, we want to calculate the metrics for these methods. Finally, we want to match the before and after metrics for each function.

The first task is to scrape all the issues from the GitHub API. For this, we use Zuilhof's code. This gets all the issues from a repository. We then cache these issues locally. After this, we want to filter out all issues related to a bug. For this, we match the issue message and labels with a bug label that is defined per project. Lastly, we use the issue number to match it to a commit.

We now have a list of commits that are linked to bug fixes. However, we want to calculate the metrics *before* and *after* the bug fix. We can use git to first checkout the commit that is linked to the bug fix. This will be the *after* code metrics. As a commit can have multiple parents, we need to checkout each of the parents. These are the *before* metrics. In figure 7 we show an example of how this git structure is build.

However, when calculating the metrics we only want to include the functions that were changed. For this information, we can use the 'git diff' function. This will give us what lines have been added and which are removed in what files. We then parse this information using a regular expression. When calculating the metrics we only include functions that match the lines that were changed.

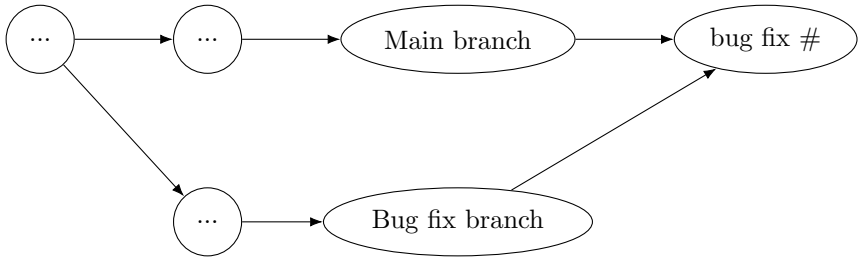

Figure 6: Merge git tree

5.4 Data flow

We do not want to calculate the metrics each time we want to generate a model. To prevent this, we output the data in an intermediate format with all the information that may be needed in the next stage. The format is shown in table 7. This data is then stored and used in later stages to generate the data for the models and other figures of the results sections.

To go from csharp code to a regression model we need to take several steps. The first is generating the metrics per csharp function and converting it to JSON, here all the information that we have is saved. The second step is importing it again in another csharp project. In this project we select the data we want and output it to a comma-separated (.csv) file. Then use Python for training the models. For this training, we use SKLearn. This provides functions for stratified learning and fitting the data to a logistical regression model.

Name	Type	Description
Function Name	string	The full name of the function as described in section 5.2.4
Commit Hash	string	The commit this data point originates from
Method Type	enum	Method, Local or lambda
Dependency Count	int	The amount of functions this function depends on
Total Lines Of Source Code	int	Total lines of source of this function and its dependencies
Metrics	Dictionary<Measure, int>	OOP and FP metrics
Violations	List<ViolationWithDistance>	The purity violations with distance

Table 7: Data model output

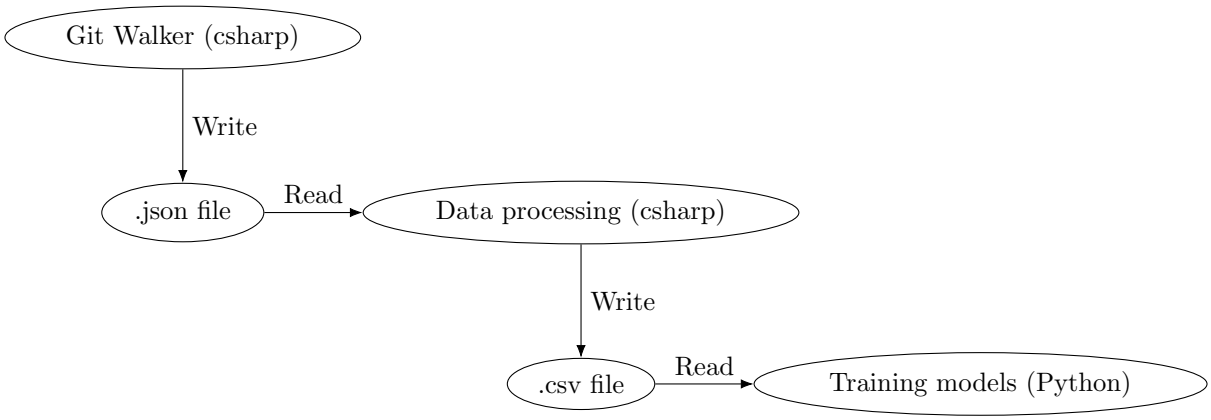

Figure 7: Data flow application

6 Results

In this section, we will discuss the results of our analysis. This can be divided into two parts. The first part goes over the validation results as defined in section 4.9. The second part goes over performance results for the different purity metrics and compares the purity metric to existing metrics. In these metrics we show the results per project and **overall**. It is important to note that the **overall** results are not the averages of the individual project. Instead, all the data is combined into one dataset and evaluated. This is done by loading the data from all projects and outputting it to one file that is used to train the model.

6.1 Validation results

In this section, we will give an overview of the validation results. We first review how the purity metric changes before and after a bug and then go over the purity violation distribution in error-prone and good code.

6.1.1 Change in purity violations after bug fixes

In the histogram, in figure 8 we show the changes in violations after a bug fix. For this, we use the normalised violations count *nvc* as defined below, where V are the violations and LSC are the total lines of source code the function spans. This metric was chosen because we want to measure how much of the changed code is impure.

$$nvc = \frac{|V|}{LSC} \tag{5}$$

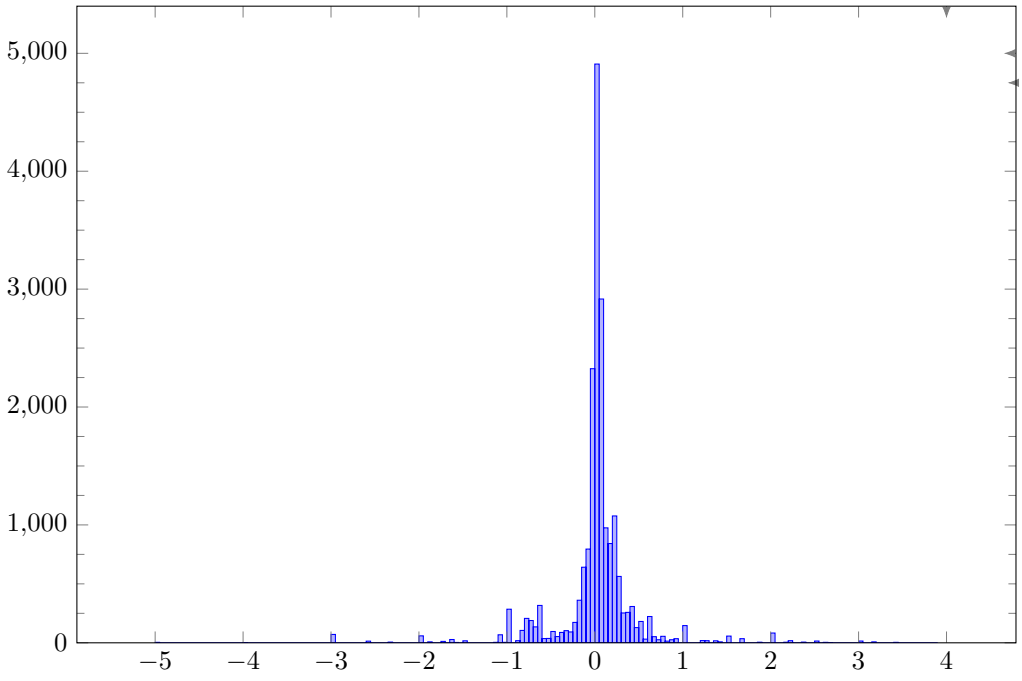

Figure 8: Histogram delta normalized purity violations after bug fixes

On average, the normalised purity violation count increased by 0,01517. This means that in our data set, the number of purity violations per line of source code increased when comparing the code before and after a bug fix. This means that, on average, more impure code was introduced with a bug fix.

6.1.2 Purity violation distribution

In the figures 9 and 10 we see the distribution of the different types of purity violations per project. Furthermore, in table 8 we can see the changes on average per violation type. Here we can see the read-local violation reduced while write-local and read-global violations increased. This is an indicator that the write-local and read-global violations could be more strongly correlated with error-prone code.

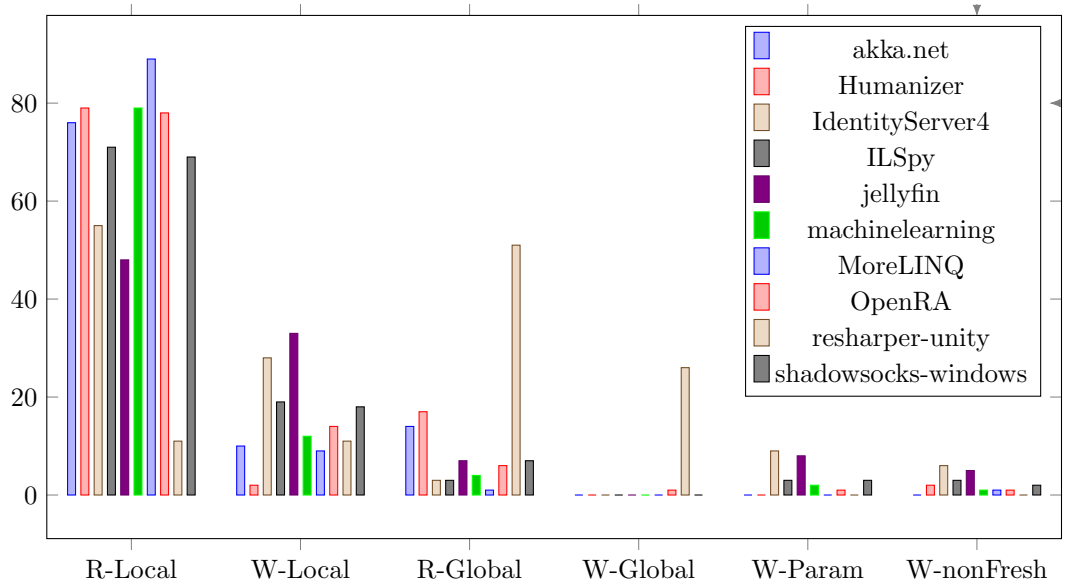

Figure 9: Purity Violation distribution for 'good' code in different projects (in percent)

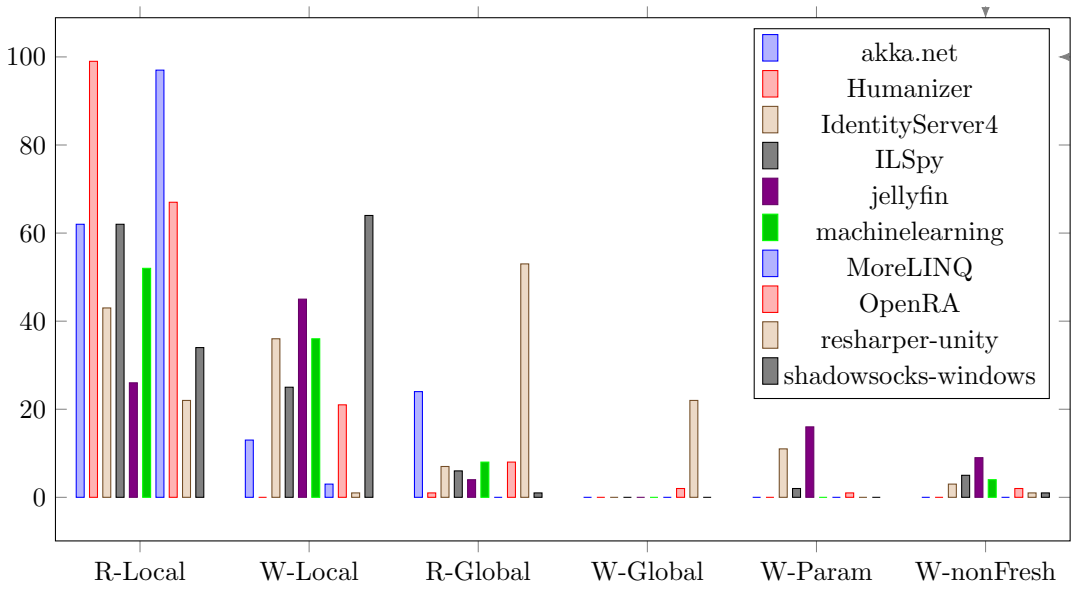

Figure 10: Purity Violation distribution for error-prone code in different projects (in percent)

Violation	**Final**	**Error-prone**	**Δ**
Read Local	73%	56%	-17%
Write Local	15%	25%	+10%
Read Global	7%	14%	+7%
Write Global	0%	0%	0%
Write Parameter	2%	3%	+1%
Write non fresh	2%	2%	0%

Table 8: Change in purity violation distribution and delta between final version of the code and the error-prone code. Here the final version is all the code that is in the repository at the moment we retrieved it. And the error-prone code is code that was changed in a bug fix

6.2 Purity metric

The purity metric is defined using logistic regression, as described in section 4.4. However, we do not yet have a concrete metric to evaluate. In this subsection, we will define different transformation functions that are used as a hyper parameter in our model and measure which one performs the best. This transformation function will be used in the remaining comparisons.

We define our purity metric per function. The main input for the purity metric is a vector of 2-tuples that contain the violation type and distance. For more information see section 3.4. However, for the training and validation of our model we need a vector of numbers where each element represents a violation type. e.g. one number for the local reads, local writes, etc. To accomplish this we first group the vector of violations per violation type. This will give us a vector for each violation type, it is possible that we have an empty vector for some violation types. Now we can define the purity metric on the vectors per violation type.

We have defined transformation functions that can be found in the formulas below.

Here we will first give a textual introduction. P_1 uses the number of violations without taking anything else into account, P_2 uses the total lines of source code to normalise the violation count over the size of the functions, P_3 uses the distance of each violation to take into account if the violation happened in this function or if it is further away in the dependency graph, P_4 is a combination of P_2 and P_3. Finally, P_{lite} only uses the direct violations of a function as this reduces the amount of computation one would have to do, hence the *lite*.

$$P_1(\vec{v}) = |\vec{v}| \tag{6}$$

$$P_2(\vec{v}) = \frac{|\vec{v}|}{TLSc} \tag{7}$$

$$P_3(\vec{v}) = \sum_{n=1}^{|\vec{v}|} \frac{1}{v_d} \tag{8}$$

$$P_4(\vec{v}) = \frac{P_3(\vec{v})}{TLSc} \tag{9}$$

$$P_{lite}(\vec{v}) = \sum_{n=1}^{|v|} \begin{cases} 1 & \text{if } vd_i = 0 \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

6.2.1 Purity metric evaluation

To select which transformation function is the best we use each as a hyper parameter in our logistical regression model. To do this we use sklearn and the training dataset to fit the the coefficients in each model. Then we use the evaluation dataset to evaluate the performance of each model. In table 9 we can see the precision, recall and f-score for each transformation function. Here we can see that P_3 has the highest overall highest precision (0,59), recall (0,36) and f-score (0,36). As a result, we will use P_3 for future comparisons against the other models. Notable is that the P_{lite} metric has the second highest F_1 score (0,42).

Metric	**Precision**	**Recall**	$\mathbf{F_1}$
P_1	0,58	0,20	0,30
P_2	0,56	0,18	0,28
P_3	0,59	0,36	0,45
P_4	0,59	0,27	0,37
P_{lite}	0,53	0,35	0,42

Table 9: Precision, recall and f-score per transformation function (validation data set)

6.2.2 Evaluate model weights

To get an insight into which violation type our model associates with fault prone functions we can extract the coefficients from the best performing trained model. A multivariate logistic regression model finds (learns) a coefficient (weight) to assign to each

parameter to create the best fit for the given data. These coefficients will tell us what weight the model has given to each violation type. The model weights for the P_3 purity metric can be found in table 10. Here we can see that all weights are positive meaning that an increase in any violation increases the likelihood that the model determines that a function is error-prone. These results do not align exactly what we found in table 8 as 'read local state' violations decreased by 17% when comparing non-error-prone code to error-prone, while the 'read local state' violation still a positive weight in the model. This can be explained because, table 8 measures the total volume change of purity violations in all error-prone and non-error-prone code, while the numbers in table 10 represent the weights the model uses to determine per function if it is error prone. This means that the fit function used to train the model determined that having 'read local violations' still has a positive weight even though the error-prone function contains relatively fewer of those violations.

Violation	**Coefficient**
Read local state	0,0883423
Modifies local state	0,16542579
Read global state	0,26670856
Modifies global state	0,446193
Write to parameter	0,24868768
Write to non-fresh	0,45293643

Table 10: Model weights for P_3 purity metric logistical regression model

6.2.3 Function type analysis

When analysing functions, we can distinguish between three types. (1) methods, (2) local functions and (3) lambda functions. This section will investigate whether our model from section 6.2 scores better for a particular function type, these results can be found in table 11. Furthermore, we trained a specific model for each function type to investigate how this affects the results. These results can be found in table 12.

In table 11 we can see that Lambda and Local functions perform badly when using the general model and the method score is in line with the results from section 6.2. One reason for this could be the imbalanced set, 235.246 Method, 46.698 Lambda and 4.290 Local. To further investigate this we also trained a model per function type and these show better result for lambdas (0,03 to 0,19), local (0,05 to 0,21) and for methods (0,30 to 0,42).

Metric	**Precision**	**Recall**	$\mathbf{F_1}$
Lambda	0,07	0,02	0,03
Local	0,02	0,03	0,05
Methods	0,26	0,37	0,30

Table 11: Precision, recall and f-score per construct (test data set). Model trained on general data set and scored per construct.

Metric	Precision	Recall	$\mathbf{F_1}$
Lambda	0,53	0,12	0,19
Local	0,30	0,16	0,21
Methods	0,39	0,46	0,42

Table 12: Precision, recall and f-score per construct (test data set). Model trained and tested on construct

6.3 Model comparison

This section will compare the baseline FP and OOP models against our novel purity metric. For the baseline FP model, we use the metrics defined by Zuilhof [39]. For the baseline OOP model, we use the metrics defined in section 4.6. The results are shown in table 13. Here the precision, recall and f-score per model are shown.

Here we see that the purity metric has the highest precision (0,22) and F_1 score (0,27). Furthermore the functional and object oriented models are close with an F_1 score of 0,22 and 0,21 respectively. However, the recall is higher in the object-oriented model (0,38). Note that the score from the purity (P_3) differs from table 9 because we used the test and not the validation set as we did in section 3.4 with table 9.

Metric	Precision	Recall	$\mathbf{F_1}$
Purity	0,22	0,34	0,27
Functional	0,17	0,32	0,22
Object-Oriented	0,15	0,38	0,21

Table 13: Precision, recall and f-score per model (test data set)

6.3.1 Model Combination

This section will go over how different combinations of models compare to each other. The models are combined by using the combining the metrics from each model into one data point. In table 14 we see that there only is a small improvement (0,28) over the best performing single model (0,27) by combining all three models.

Metric	Precision	Recall	$\mathbf{F_1}$
Purity	0,22	0,34	0,27
Functional	0,17	0,32	0,22
Object-Oriented	0,15	0,38	0,21
Purity & object-Oriented	0,17	0,34	0,23
Functional & object-oriented	0,18	0,53	0,27
Purity & functional	0,19	0,48	0,27
Purity & functional & object-oriented	0.19	0.53	0.28

Table 14: Precision, recall and f-score per model and per combination (test data set)

7 Related work

7.1 Purity in object-oriented languages

There have been multiple studies on this topic, each with different strategies. Finifer et al. [8] limit the use of Java to a subset called Joe-E which makes it trivial to validate purity for functions written in this language. The disadvantage of this approach is that it requires the code to be written in Joe-E, meaning that existing Java code can not be analysed.

Another Java implementation developed by Pearce [32] called Jpure can analyse regular Java code. The implementation consists of a purity inference and checker algorithm enabling partial code analysis. Furthermore, they use the idea of a freshness which tracks if an object (reference type) is newly (freshly) allocated in the current scope.

Nicolay et al. [29] focussed on Javascript and used a combination of pushdown flow, control flow and value flow analysis to determine the purity of a function. To do this, they create a flow graph that can be analysed. To create this flow graph, they defined an abstract machine with its own input language based on Javascript to simplify the analysis.

Recent research from Österberg [31] developed an implementation for csharp. This approach is based on different aspects of the other studies mentioned. The result is a code analysis checklist which outputs one of five purity levels *pure, locally impure, parametrically impure and unknown* or *impure*. Österberg also provides a partial implementation of this checklist called CsPurity. This partial implementation was developed using Roslyn and was the basis for the implementation in this thesis.

7.2 Code quality metrics in multi-paradigm languages

Early research from, among others, Briand et al. [3], Fioravanti et al. [9], Gyimóthy et al. [12] focussed on defining and validating metrics from object-oriented code. In these studies, a large number of object-oriented metrics were examined. For example, in the validation study from Fioravanti et al. 200 metrics were considered to create a model to predict error-proneness. Eventually, only 12 metrics were used in the final model. However, each of these studies tested on limited datasets from one source. For example, research from Gyimóthy used new projects that were developed by a group of three students over a relatively short period of time.

Recent research from Landkroon [20], Zuilhof [39] and Konings [19] focussed on more recent developments in programming languages that were a combination of object-oriented and functional code is used. This research tested object-oriented metrics in multi-paradigm code, and new metrics that try to capture multi-paradigm constructs were defined and validated. Landkroon and Konings focused on Scala, where they determined if a piece of code is more object-oriented or functional. They then use this to make a more accurate error-prediction model for Scala. Zuilhof focussed on csharp, where he introduced new metrics that focussed on development in the language that make it more functional. These studies use git repositories and issue trackers to create more real-life datasets from longer-running professional development projects. Furthermore, Landkroon adapted the technique for Briand [2] et al. that aims to more accurately use error-prone code to build the dataset.

8 Conclusion

8.1 Research questions

In this section we will discuss the research questions defined in section 1.1.

RQ1: How can the concept of functional purity be used as a software quality metric in an object-oriented language?

In this study, we based our purity metric on purity violations, see section 2. Now that we have these violations, we can use lines of source code and dependency count combined with logistical regression to define a purity metric model. In section 6.2 we defined and experimented with different ways to define a purity metric. We found that using the distance to weigh each violation (P_3) gave the best results in the general case. Furthermore, only using the direct violations (P_{lite}) also gave acceptable results while greatly reducing the complexity of the analysis.

RQ2: How well do the purity metrics predict error-proneness?

To answer this question, we compared the best-performing purity metric against models derived from existing object-oriented and functional metrics. In this comparison, our purity metric had an f-score of 0,27 and thus performed slightly better than the functional and object-oriented models, which both had an f-score 0,22 and 0,21 respectively.

RQ3: How well does the purity metric perform in the different types of functions in csharp?

From previous research from Zuilhof [39] and Konings [19], we expected that our purity metric would perform best in Lambda methods. Nevertheless, this was not the case in our results. The best performing function construct was methods with an f-score of 0,30 compared to 0,03 and 0,05 for local and lambda functions. This could be explained by the high number of methods on our training dataset. Because we know which function type we want to analyse we can train a specific model for each function type. When we do this the results for each function type increase. For lambda functions the f-score went up from 0,03 to 0,19 for local functions from 0,05 to 0,21 and for methods from 0,31 to 0,42. In conclusion our purity metric works best on methods and when we train a specific model for each function type.

RQ4: To what extent does combining existing metrics with the purity metric improve the error-proneness prediction?

To answer this question, we created four models that combined functional, object oriented and the P_3 purity metric. When evaluating these combined models, there was no notable increase in performance in comparison to the best performing single model (the purity metric) by itself. Only by combining all three models we saw a sligh increase in the f-score from 0,27 to 0,28.

RQ: To what extent can functional purity be used as a code quality metric in an object-oriented language like csharp?

Within object-oriented languages, there exists impure behaviour by design. However, how impure a function is can be quantified by measuring what kind of and how many impure actions it contains. By using these measurements, a purity metric can be calculated. Moreover, when evaluating the error-proneness predictions, the purity metric

scored better than existing object-oriented and functional metrics. Even though these results seem promising these are derived from a limited data set.

8.2 Discussion

In this thesis, we tried to answer the question of if the concept of functional purity can be used as an indicator for error-prone code. To do this we used open-source projects and their issues trackers as a dataset. For this to work we made the assumption that the code that is changed in a 'bug fix' is error-prone code. While this may not create a perfect dataset, it is, in our opinion, the best way to do this as it gives access to long-running, professionally developed projects with relatively little effort. Moreover, the data that we analyse can be as recent as bug fixes that were made on the day we analysed the data. Our approach was a quantitative one without any human filtering. One could also take a more qualitative approach by selecting a project where an expert can go through the history and mark any pieces of code that were the source of a bug. However, this has the disadvantage of being subject to the bias of the expert as well as being labour intensive, thus generating fewer examples that can be used in the regression model.

Nextly, our purity analysis is not a big leap forward. However, it does combine and improve on previous studies. The main contributions are that it gives a better insight into why a function is impure by knowing what impure behaviour a function has and where it originated. This enables more opportunities to apply the results. Furthermore, it solved the future work problem from Österberg where it was not possible to correctly analyse recursive programs.

Finally, the results are promising as there was a correlation between impure behaviour and error-proneness. However, we still lack knowledge to know what exactly to do with this information. One could take our model and let it loose on a csharp code base to predict what functions are error-prone. However, the false-positive rate would be too high to be useful. Still, one could further analyse what purity characteristics error-prone functions usually have, see future work section 8.3.4.

8.2.1 Practical implementation

To use this in practice, we have different options. One could integrate the purity analyser into a build pipeline or an IDE. This could enforce, for example, specific namespaces or methods to be pure. As a proof of concept, we wrote a visual studio plugin that uses the purity analyser to validate the [Pure] attribute. This could easily be done as Visual Studio uses Roslyn for its extensions.

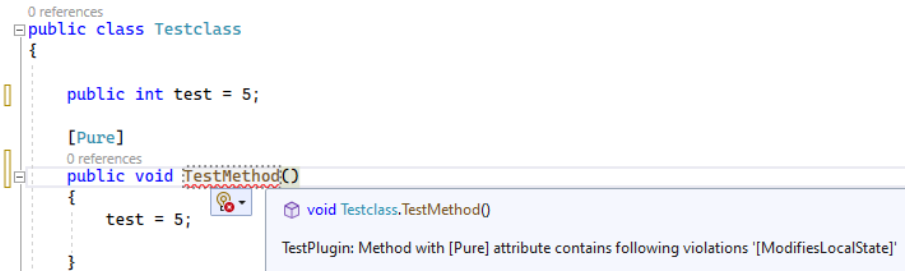

Figure 11: Proof of concept plugin for Visual Studio 2022

8.3 Future work

8.3.1 Analysis standard library

When determining the purity violations, there have been a lot of cases where we could not analyse the function because the dependency is outside of the source code. We create a workaround for this to get acceptable data. See section 4.3.1. However, Microsoft provides the source code of the .NET standard library at https://referencesource.microsoft.com/#mscorlib. Future research can try to use this source code to create a map of which violations each function commits to increase the accuracy of the violation algorithm.

8.3.2 Byte code analysis

To go even further to solve the unknown code problem, one could analyse the byte code when the source code is not available. To do this, one would need to adapt our analysis to work on byte code (IR code). This would require extra work as Roslyn can not be used. As a starting point, one could look to Pearce's [32] JPure implementation for inspiration.

8.3.3 Data flow analysis for freshness

In the implementation of our thesis, we can determine the freshness of an object. However, the algorithm can in some cases over eagerly mark something as non-fresh while it is fresh when looking at the control flow. This is because it does not take data flow into account. Future research could look at incorporating data flow analysis into our implementation. For this one could start by using the research from Nicolay et al. [29].

8.3.4 Applying the purity metric

As shown in a proof of concept described in section 8.2.1 we can use the purity analysis in the csharp build tools. However, what feedback we want to give to the developer is not known. In a more applied research, one could further develop a Roslyn analyser and find out what feedback is useful to give to the developer at what stage of the development process. Furthermore, it would be interesting to apply the purity metric in some form in practice.

References

[1] akkadotnet. *Akka.NET*. https://github.com/akkadotnet/akka.net. 2022.

[2] Lionel Briand and Khaled El Emam. "Theoretical and Empirical Validation of Software Product Measures 1". 1995. URL: http://www.elet.polimi.it/~morasca.

[3] Lionel C. Briand, Walcelio L. Melo, and Jürgen Wüst. "Assessing the applicability of fault-proneness models across object-oriented software projects". In: *IEEE Transactions on Software Engineering* 28 (7 July 2002). Focusses on Java¡br/¿, pp. 706–720. ISSN: 00985589. DOI: 10.1109/TSE.2002.1019484.

[4] Arti Chhikara and Sujata Khatri. "Applying Object Oriented Metrics to C#(C Sharp) Programs". In: *International Journal of Computer Technology and Applications* 02 (May 2011).

[5] Shyam R. Chidamber and Chris F. Kemerer. "A Metrics Suite for Object Oriented Design". In: *IEEE Transactions on Software Engineering* 20 (6 1994), pp. 476–493. ISSN: 00985589. DOI: 10.1109/32.295895.

[6] dentityServer. *dentityServer4*. https://github.com/IdentityServer/IdentityServer4. 2022.

[7] dotnet. *Roslyn*. https://github.com/dotnet/roslyn. 2022.

[8] Matthew Finifter et al. "Verifiable functional purity in Java". In: *Proceedings of the ACM Conference on Computer and Communications Security* (2008), pp. 161–173. ISSN: 15437221. DOI: 10.1145/1455770.1455793.

[9] F. Fioravanti and P. Nesi. "A study on fault-proneness detection of object-oriented systems". In: *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR* (2001). 200+ metric used to define a model. Result: 1 model that uses 42 metric and a smaller model with 12 metric that has smaller precision, pp. 121–130. DOI: 10.1109/CSMR.2001.914976.

[10] D. Glasberg et al. "Validating Object-Oriented Design Metrics on a Commercial Java Application". In: (Jan. 2000). DOI: 10.4224/8914217.

[11] Qiong Gu, Li Zhu, and Zhihua Cai. "Evaluation Measures of the Classification Performance of Imbalanced Data Sets". In: *Communications in Computer and Information Science* 51 (2009), pp. 461–471. ISSN: 18650929. DOI: 10.1007/978-3-642-04962-0_53. URL: https://link.springer.com/chapter/10.1007/978-3-642-04962-0_53.

[12] Tibor Gyimóthy, Rudolf Ferenc, and István Siket. "Empirical validation of object-oriented metrics on open source software for fault prediction". In: *IEEE Transactions on Software Engineering* 31 (10 Oct. 2005). Focus on C++, pp. 897–910. ISSN: 00985589. DOI: 10.1109/TSE.2005.112.

[13] Maurice H Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., 1977. ISBN: 0444002057.

[14] Humanizr. *Humanizr*. https://github.com/Humanizr/Humanizer. 2022.

[15] Icsharpcode. *ILSpy*. https://github.com/icsharpcode/ILSpy. 2022.

[16] *ISO/IEC 25010*. Standard. Geneva, CH: International Organization for Standardization.

[17] Jellyfin. *Jellyfin*. https://github.com/jellyfin/jellyfin. 2022.

[18] Jetbrains. *resharper-unity*. https://github.com/JetBrains/resharper-unity. 2022.

[19] Sven Konings. “Source Code Metrics for Combined Functional and Object-Oriented Programming in Scala”. University of Twente, 2020.

[20] Erik Landkroon. “Code Quality Evaluation for the Multi-Paradigm Programming Language Scala”. University of Amsterdam, 2017.

[21] libgit2. *libgit2sharp*. https://github.com/libgit2/libgit2sharp. 2022.

[22] Thomas J. Mccabe. “A Complexity Measure”. In: *IEEE Transactions on Software Engineering* SE-2 (4 1976). Cyclomatic complexity, pp. 308–320. ISSN: 00985589. DOI: 10.1109/TSE.1976.233837.

[23] Microsoft. *Code contracts (.NET Framework)*. 2021. URL: https://docs.microsoft.com/en-us/dotnet/framework/debug-trace-profile/code-contracts (visited on 09/15/2021).

[24] Microsoft. *PureAttribute Class*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.contracts.pureattribute?view=net-6.0.

[25] Microsoft. *Reactive*. https://github.com/dotnet/reactive. 2022.

[26] microsoft. *ML.NET*. https://github.com/dotnet/machinelearning. 2022.

[27] morelinq. *MoreLINQ*. https://github.com/morelinq/MoreLINQ. 2022.

[28] Jens Nicolay et al. “Detecting function purity in JavaScript”. In: *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation, SCAM 2015 - Proceedings* (Nov. 2015), pp. 101–110. DOI: 10.1109/SCAM.2015.7335406.

[29] Jens Nicolay et al. “Detecting function purity in JavaScript”. In: *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation, SCAM 2015 - Proceedings* (Nov. 2015), pp. 101–110. DOI: 10.1109/SCAM.2015.7335406.

[30] OpenRA. *OpenRA*. https://github.com/OpenRA/OpenRA. 2022.

[31] Melkerg Osterber. “Measuring Functional Purity in C#”. Uppsala Universitet, 2021.

[32] David J. Pearce. “JPure: A modular purity system for Java”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 6601 LNCS (2011), pp. 104–123. ISSN: 03029743. DOI: 10.1007/978-3-642-19861-8_7.

[33] Chris Ryder. “Software Measurement for Functional Programming”. Canterbury, UK, Aug. 2004. URL: https://kar.kent.ac.uk/14117/.

[34] Chris Ryder and Simon Thompson. “Software Metrics: Measuring Haskell”. In: *Trends in Functional Programming*. Ed. by Marko van Eekelen and Kevin Hammond. Sept. 2005, pp. 182–196. URL: http://www.cs.kent.ac.uk/pubs/2005/2249.

[35] Alexandru Salcianu and Martin C. Rinard. “Purity and Side Effect Analysis for Java Programs”. In: (2005).

[36] scikit-learn. *scikit-learn*. https://github.com/scikit-learn/scikit-learn. 2022.

[37] Shadowsocks. *Shadowsocks-windows*. https://github.com/shadowsocks/shadowsocks-windows. 2022.

[38] Robert Tarjan. “Depth-First Search and Linear Graph Algorithms”. In: *SIAM Journal on Computing* 1.2 (1972), pp. 146–160. DOI: 10.1137/0201010. eprint: https://doi.org/10.1137/0201010. URL: https://doi.org/10.1137/0201010.

[39] Bart Zuilhof. “Code Quality Metrics for the Functional Side of the Object-Oriented Language C#”. University of Amsterdam, 2019.

Appendix

A Checklist to determine purity

The purity checklist to determine purity as defined by Österberg [31]. Here p means purity level and m denotes a method.

1. If any object field or property of the currently analysed method m's object is read from or modified, mark m as locally impure.
2. If m calls a locally impure method belonging to m's object (i.e.this) m is marked locally impure.
3. If the method m reads or modifies a static field of any class or object, m is marked impure. This is because reading a static field is a non-deterministic action, and modifying a static field is a side effect since it mutates the field for all instantiations of that object's class.
4. If the method m calls a locally impure method of an object assigned to a static field of any class or object, m is marked impure.
5. If the method m calls an input parameter's method m_p and m_p is overridden by any locally impure and/or parametrically impure method, m_p is temporarily marked with the impurities of all the overriding methods in the context of the analysis of the current method m. If m_p is overridden by any impure method, m is permanently marked as impure.
6. If the analysed method m modifies an input parameter of reference type, mark m as parametrically impure. This could be done in a couple of ways:
   (a) By calling an object type parameter's method that has been marked as locally impure.
   (b) By passing an object type parameter as an argument to a method that has been marked as parametrically impure.
   (c) By directly mutating a parameter object's field or property, or the cell of a parameter array. If m does at least one of the above, mark it as parametrically impure.
7. If a method returns this or passes it as an argument to a function it marked locally impure since it is dependent on the state of its object, making it non-deterministic.
8. Any method that raises an event is marked impure.
9. Any method that raises an exception is marked impure.
10. Any method that is marked impure by hand is impure.

B Full details of project used

ML for .NET

- **Commit hash used:** 97a920ac5cd326412a77a3aa13b83e53385e4bde
- **Issues scraped on:** 15 April 2022
- **Url:** https://github.com/dotnet/machinelearning

Identity server

- **Commit hash used:** 5bcc2abbcf6c47200c4eb3b77e756f1bc0e04358
- **Issues scraped on:** 21 April 2022
- **Url:** https://github.com/IdentityServer/IdentityServer4

AKKA.net

- **Commit hash used:** 54d8e5cfc3ab2ded4965f5a9f270ad1b43bc5491
- **Issues scraped on:** 15 April 2022
- **Url:** https://github.com/akkadotnet/akka.net

Jellyfin

- **Commit hash used:** 1fdef1248a77036387de1e4b5b93ac3a100ae49f
- **Issues scraped on:** 15 April 2022
- **Url:** https://github.com/jellyfin/jellyfin

OpenRA

- **Commit hash used:** c6dc0b58be9d3918f6b41d900d1f07815fe87cf3
- **Issues scraped on:** 16 April 2022
- **Url:** https://github.com/OpenRA/OpenRA

ILSpy

- **Commit hash used:** ca9c288dc537723eba5647100588004b80b7c548
- **Issues scraped on:** 11 May 2022
- **Url:** https://github.com/icsharpcode/ILSpy

Humanizer

- **Commit hash used:** 606e958cb83afc9be5b36716ac40d4daa9fa73a7
- **Issues scraped on:** 12 April 2022
- **Url:** https://github.com/Humanizr/Humanizer

MoreLinq

- **Commit hash used:** fb9da772ffc7bd2fd01aa364de2def84df6feca2
- **Issues scraped on:** April 2022

- **Url:** https://github.com/morelinq/MoreLINQ

Reactive

- **Commit hash used:** 85f1eb7c53e27cccdbeee3e0b044916168843fcc
- **Issues scraped on:** 12 May 2022
- **Url:** https://github.com/dotnet/reactive

Shadowsocks

- **Commit hash used:** 8fafef7c5751809751d8eef361e3c39caef75261
- **Issues scraped on:** 12 May 2022
- **Url:** https://github.com/shadowsocks/shadowsocks-windows

Reshaper

- **Commit hash used:** 51d5309de26ffa488ab0a896466fbd1c3fa76269
- **Issues scraped on:** 12 May 2022
- **Url:** https://github.com/JetBrains/resharper-unity

Roslyn

- **Commit hash used:** e1629052738683dac234a4d75bb1d1c442c46d7d
- **Issues scraped on:** 13 April 2022
- **Url:** https://github.com/dotnet/roslyn