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

Bart Zuilhof
University of Amsterdam
Amsterdam, Netherlands
bart_zuilhof@hotmail.com

Rinse van Hees
Info Support
Veenendaal, Netherlands
rinse.vanhees@infosupport.com

Clemens Grelck
University of Amsterdam
Amsterdam, Netherlands
c.grelck@uva.nl

Abstract

With the evolution of object-oriented languages such as C#, new code constructs that originate from the functional programming paradigm are introduced. We hypothesize that a relationship exists between the usage of these constructs and the error-proneness. Measures defined for this study will focus on functional programming constructs where object-oriented features are used, this often affects the purity of the code. Built on these measures we try to define a metric that relates the usage of the measured constructs to error-proneness. To validate the metric that would confirm our hypothesis, we implement the methodology presented by Briand et al. [BEEM95] for empirical validation of code metrics. The results of this research granted new insights into the evolution of software systems and the evolution of programming languages regarding the usage of constructs from the functional programming paradigm in object-oriented languages.

1 Introduction

The growing popularity of multi-paradigm language Scala [Car] and the introduction of functional programming (FP) features in significant object-oriented (OO) languages such as Java [Ora] and C# [Mic], code evaluation for multi-paradigm languages have become more significant. Landkroon has shown [Lan17] that metrics from the OO paradigm and the FP paradigm can be mapped to the multi-paradigm language Scala. However, the integration of OO and FP features introduces artifacts that are neither covered by OO-inspired code metrics nor by FP-inspired code metrics. These constructs, such as usage of mutable class variables in lambda functions whose execution might be deferred, are not possible in the FP languages but are used in a functional manner. To evaluate these patterns, the metrics that are proposed for the OO paradigm [BBM96, HKV07, McC76] and FP paradigm [RT05, VdB95] might be unsuitable to give a valuable indication of quality regarding the usage of these combined constructs.

Measures that indicate complexity might have an intuitive relationship with error-proneness. But this does not have any concrete meaning and usefulness since you can not substantiate a predication by just intuition. Therefore evidence must be provided that a measure is useful, this can be done by proving there is a relationship to an external attribute such as error-proneness [BEEM95].

We follow the use of the term 'measure' as used by Briand et al. [BEEM95]. Where the term 'measure' refers to an assessment on the size of an attribute of the code.

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

2 Background

Since version 3.0 a set of features were added to C#, that are inspired by FP. Functions are now first-class constructs, which enables higher-order functions.

In: Anne Etien (eds.): Proceedings of the 12th Seminar on Advanced Techniques Tools for Software Evolution, Bolzano, Italy, July 8-10 2019, published at http://ceur-ws.orgProceedings of the SATToSE.

Lambda functions provide a concise way of describing an anonymous function. Pattern matching allows concise and rich syntax for doing switch-case statements. The concept of lazy evaluation, which comes along with the LINQ[1] query syntax, which was previously only possible by using the `Lazy<T>`-keyword [Mic]. LINQ introduced syntax for list operations such as `map`, `filter` and `sort`, which are basically higher order functions as known from functional languages. This syntax enables list mutations with concise syntax in C# as shown in Listing 1.

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

Listing 1: C# With LINQ

3 Problem Analysis

In the following code snippets, two basic implementations are presented which have the functionality to get a list with vehicles that that start with 'Red'. For the first implementation (Listing 2) an imperative approach is chosen. The snippet has a Source Lines of Code (SLOC) count of 11 and a Cyclomatic Complexity of 3 since there are two branching points. This is how the general complexity of the snippet translates back into the values returned by the metrics.

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

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

Listing 2: C# Without LINQ

The second implementation (Listing 4) LINQ library is used, which encourages the use of lambda-expressions. The snippet has a SLOC count of 5 and a cyclomatic complexity of 1 since there are no branching points. Even though the functionality and the logical complexity are the same with both snippets, the cyclomatic complexity and SLOC differ drastically.

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

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

Listing 3: C# With LINQ

```
int Foo(int x, int y)
{
    if (x < 0)
        x = 0;
    if (y > 5)
        y = 2;
    return y - x;
}
```

Listing 4: C# With Temp

4 Candidate Measures

4.1 Number Of Lambda Functions Used In A Class (LC)

Lambda functions in the context of OO languages are a concise way to write anonymous functions inline. Compared to a regular method, the parameter type, return type can all be omitted. This might introduce constructs which are harder to understand. An example for this scenario is given in Listing 5.
To calculate the value for this measure, we traverse the AST (abstract syntax tree). For each *SyntaxNode* which has the type *LambdaExpression*, we raise the counter for this measure.

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

Listing 5: An Example Of A Lambda Expression

4.2 Source Lines of Lambda (SLOL)

Where simple lambda expressions might not be extra information to reason about the execution, once the lambda expression becomes most complex this might not be the case. In listing 6 an example is given with a multiline lambda expression. As curly braces are taken included in the 'source lines of code'-measure [HKV07],

[1] Language Integrated Query, an uniform query syntax in C# to retrieve data from different sources [Wag17]

we also include these curly braces when calculating the span of the lambda expression. Therefore, the snippet in Listing 6 has an 'Source Lines of Lambda'-count of $1 + 1 + 4 = 6$.

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

Listing 6: An Example Of A Multiline Lambda Expression

4.3 Lambda Score (LSc)

The density of the usage of lambda functions in a class can give an indication of how functional a class is. Our hypothesis for this measure is, that a relationship exists between how functional a class is and the error-proneness of the class. We calculate this lambda density with Equation 1.

$$LSc = \frac{SLOL}{SLOC} \tag{1}$$

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

4.4 Number Of Lambda Functions Using Mutable Field Variables In A Class (LMFV)

Sometimes it might be hard to predict when a lambda function is executed, therefore, it might be hard to reason about what value for the mutable field will be used. An example for this scenario is given in Listing 7.
To calculate the value for this measure, we traverse the AST. For each variable inside a lambda expression, we check if the variable is non-constant and field scoped by using the semantic data model (SDM) of the class. If this test passes, we increase the counter for this measure.

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

Listing 7: An Example Of A Lambda Expression With A Reference To A Mutable Field Variable

4.5 Number Of Lambda Functions Using Mutable Local Variables In A Class (LMLV)

Sometimes it might be hard to predict when a lambda function is executed, therefore, it might be hard to reason about what value for the mutable local variable will be used. An example for this scenario is given in Listing 8.
To calculate the value for this measure, we traverse the AST. For each variable inside a lambda expression, we check if the variable is non-constant and local scoped by using the semantic model of the class. If this test passes, we increase the counter for this measure.

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

Listing 8: An Example Of A Lambda Expression With A Reference To A Mutable Local Variable

4.6 Number Of Lambda Functions With Side Effects Used In A Class (LSE)

We think that the combination of side effects in lambda functions with e.g. parallelization or lazy evaluation is dangerous because it can be hard to reason about when these side effects occur. An example for this scenario is given in Listing 9.
To calculate the value for this measure, we traverse for each class its AST. For each variable inside a lambda expression, we check if local or field variables are being mutated.

```
static int _y = 2;

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

Listing 9: An Example Of A Lambda Expression With A Side Effect To A Mutable Field Variable

4.7 Number Of Non-Terminated Collection Queries In A Class (UTQ)

By not terminating a collection query, it will be hard to reason when the query will be executed. Since these collection queries may contain functions that contain side effects and use outside scope variables, the execution at different run-times can yield different and unexpected results. An example for this scenario is given in Listing 10.
To calculate the value for this measure we traverse the AST and count how many `IEnumarable<T>` are initiated.

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

Listing 10: An Example Of A LINQ-Query That Is Not Evaluated/Terminated

5 Methodology

To empirically validate a proposed metric according to Briand et al [BEEM95] describes three assumptions that should be satisfied namely:

1. **The internal attribute A_1 is related to the external attribute A_2** . The hypothesized relationship between attribute A_1 and A_2 can be tested if assumption 2 and assumption 3 are assumed, by find a relationship between X_1 and X_2.
2. **Measure X_1 measures the internal attribute A_1**. Measure X_1 will measure defined attributes of the code such as mutable external variables used in lambda functions. This measure X_1 will be assumed to measure A_1, A_1 will the internal attribute such as purity of the lambda usages.
3. **Measure X_2 measures the external attribute A_2**. Measure X_2 will measure the error-proneness A_2 of a given class. The measure X_2 used will be if the class contains a bug yes or no.

In Section 7 we elaborate on our approach to satisfying the above assumptions.

6 Dataset

For this study we analyzed the following projects:

- **CLI** The .NET Core command-line interface (CLI) is a new cross-platform toolchain for developing .NET applications. The CLI is a foundation upon which higher-level tools, such as Integrated Development Environments (IDEs), editors, and build orchestrators, can rest [Fou15b]. *Analyzed version: bf26e7976*
- **ML** Machine Learning for .NET is a cross-platform open-source machine learning framework which makes machine learning accessible to .NET developers. ML.NET allows .NET developers to develop their own models and infuse custom machine learning into their applications, using .NET, even without prior expertise in developing or tuning machine learning models [Fou18]. *Analyzed version: b8d1b501*
- **AKK** Akka.NET is a community-driven port of the popular Java/Scala framework Akka to .NET. Akka is a toolkit for building highly concurrent, distributed, and resilient message-driven applications. Akka is the implementation of the Actor Model. [Akk14]. *Analyzed version: bc5cc65a3*
- **ASP** ASP.NET Core is an open-source and cross-platform framework for building modern cloud based internet connected applications, such as web apps, IoT apps and mobile backends. ASP.NET Core apps can run on .NET Core or on the full .NET Framework [Fou15a]. *Analyzed version: 5af8e170bc*
- **IS4** IdentityServer is a free, open source OpenID Connect and OAuth 2.0 framework for ASP.NET Core [Fou15c]. *Analyzed version: da143532*
- **JF** Jellyfin is a Free Software Media System that puts you in control of managing and streaming your media [Jel13]. *Analyzed version: d7aaa1489*
- **ORA** OpenRA is an Open Source real-time strategy game engine for early Westwood games such as Command & Conquer: Red Alert written in C# using SDL and OpenGL [For09]. *Analyzed version: 27cfa9b1f*
- **DNS** dnSpy is a debugger and .NET assembly editor. You can use it to edit and debug assemblies even if you don't have any source code available [0xd16]. *Analyzed version: 3728fad9d*
- **ILS** ILSpy is the open-source .NET assembly browser and decompiler. [ics09] *Analyzed version: 72c7e4e8*
- **HUM** Humanizer meets all your .NET needs

for manipulating and displaying strings, enums, dates, times, timespans, numbers and quantities. [Kha12] *Analyzed version: b3abca2*

- **EF** EF Core is an object-relational mapper (O/RM) that enables .NET developers to work with a database using .NET objects. It eliminates the need for most of the data-access code that developers usually need to write [Fou14]. *Analyzed version: 5df258248*

7 Evaluation Setup

7.1 Relating Functional Constructs to Error-Proneness

Investigating the relationship between code metrics to error-proneness is commonly done by creating a prediction model for error-proneness based on code metrics [BBM96, BEEM95, GFS05, BMW02].

The logistic regression classification technique [HJLS13] is often used to create such a predication model [GFS05, Lan17, BBM96, LV97].

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

7.1.1 Univariate

With a univariate logistic regression model, we can evaluate, in isolation the predication model for error-proneness based on the measured constructs. Using the Equation 2 we construct a prediction model.

$$P(faulty = 1) = \frac{e^{\beta_0 + \beta_l X_l}}{1 + e^{\beta_0 + \beta_l X_l}} \tag{2}$$

Where $\beta_l X_l$ is the coefficient multiplied with the value of the added measure.

7.1.2 Baseline

To show that our measures are useful regarding error-proneness prediction, their inclusion must yield better results than metrics that are being used in the industry. This set of metrics will define the baseline for this study. We use the union of a set of general code metrics with a set of object-oriented metrics. For general code metrics we take Source Lines of Code (SLOC) [NDRTB07], Cyclomatic Complexity (CC) [McC76] and Comment Density [Son19].

The 00 metric suite used for this study was defined by Chidamber [CK94]. For our baseline, we implemented the metrics which showed any significance in the study. The metrics suite contains the following metrics *Weighted Methods per Class (WMC), Dept of Inheritance Tree (DIT), Response For a Class (RFC), Number of Children of a Class (NOC), Coupling between Object Classes (CBO), Lack of Cohesion of Methods (LCOM).*

7.1.3 Multivariate

Besides looking if our univariate logistic regression model gives an indication of good performance, we can use multivariate logistic regression to test if we can improve the model with the in-place OO metrics. The baseline for the evaluation of the model will be a multivariate logistic regression model based on our baseline set of metrics. To see if we can achieve an increased performance compared to our baseline model, we substitute baseline dependent variable with candidate measures as l.

$$P(faulty = 1) = \frac{e^{\beta_0 + \beta_1 X_i + \beta_2 X_2 + ... + \beta_n X_n + \beta_l X_l}}{1 + e^{\beta_0 + \beta_1 X_i + \beta_2 X_2 + ... + \beta_n X_n + \beta_l X_l}} \tag{3}$$

7.1.4 Model Validation

We choose to validate our model using cross-validation, which is commonly used for the validation of prediction models [Sto74, K+95]. We use the Hold-out method for cross-validation. By default, holdout cross-validation separates the data set into a train set and a smaller test set. To compensate for the randomness of the division, we run the model fitting with multiple different selections of the training set and average the results and assess the standard deviation. Based on a classification report created for the hold-out set, we assess the performance of the model. We use the F_1-score, which calculates the harmonic mean of the precision and recall (F_1-score) to assess our model performance with Equation 4.

$$F_1 = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{4}$$

Since our data set is unbalanced, as seen in Table 1 one could choose to calculate the micro-average between the F_1-scores for the 'faulty-classes'-class and the 'non-faulty-classes'-class, where the support for each class is weighted. However, since we want good prediction performance in both classes we use the macro-average instead which calculates the harmonic mean between the two F_1-scores [SL09].

7.2 Measuring Functional Constructs

By using the compiler platform SDK 'Roslyn' which is created by the '.NET Foundation' we can derive AST's and SDM's from the classes of a given project. By traversing the AST for each class using 'Roslyn's implementation of the visitor pattern, we visit each

syntax node in depth-first order. Where needed we can request additional data from the SDM during the traversal, such as, what is the type and the level of scope for a given variable. Using this method we can calculate the values for all of our candidate measures.

7.3 Measuring Error-Proneness

To make an estimation on how error-prone a class is, we make the assumption that if a class during the lifetime of a project was updated by a bugfix, the class is error-prone. Unfortunately, the GitHub API does not provide an easy way to identify bug-fixing commits. From a GitHub repository, we can request all the issues that were created regarding a bug. With this information, we identify all commits that close an issue by searching for issue closing keywords as described in [Git19]. All commits that mention an issue that was identified as a bug related issue, are marked as bug-fix commits. We then extract the affected lines from the metadata of the commit. Then derive with which classes the affected lines intersect in the parent version of the bug-fix commit by parsing the AST for the updated file. We use the parent version of the bug-fix commit since this is the version where the bug existed. Each of these intersected classes will be marked as error-prone.

8 Results

In Table 1 descriptive stats are shown for the output of our static analysis. The test projects have been excluded for this our analysis since they are likely to be modified in a bug-fixing commit to detect the bug if it would occur again.

Table 1: Descritive Stats

Project	Classes	Bug-fixes	Faulty classes
CLI [Fou15b]	328	54	24
ML [Fou18]	1404	27	18
AKK [Akk14]	1621	171	199
ASP [Fou15a]	3212	99	118
IS4 [Fou15c]	331	40	49
JF [Jel13]	1420	81	88
ORA [For09]	1990	227	149
DNS [0xd16]	5345	?*	159
ILS [ics09]	1011	95	70
HUM [Kha12]	124	23	10
EF [Fou14]	1432	975	437

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

Notable, the relationship of bug-fixing commits on faulty classes can be either positive or negative. This can be explained by that a set of the commits are only updating configuration or non-csharp code files, therefore, $|BugFixes| > |FaultyClasses|$ is possible. The alternative scenario, one commit is able to modify multiple classes, therefore, $|BugFixes| < |FaultyClasses|$ is possible. The variance in the ratio of $\frac{Classes}{FaultyClasses}$ between project is also notable, this can be partly attributed during the lifetime of a project. However, some projects contradict this hypothesis. Where ILSpy is one of the older projects that were analyzed, the ratio is not higher than e.g. the AKKA.NET project. Even though, the ILSpy project is more than twice as old.

To evaluate the measures in isolation, we fit and evaluate a prediction model using univariate regression. In figure 1 we can see the macro-average F1-score for each project in combination with each candidate measure.

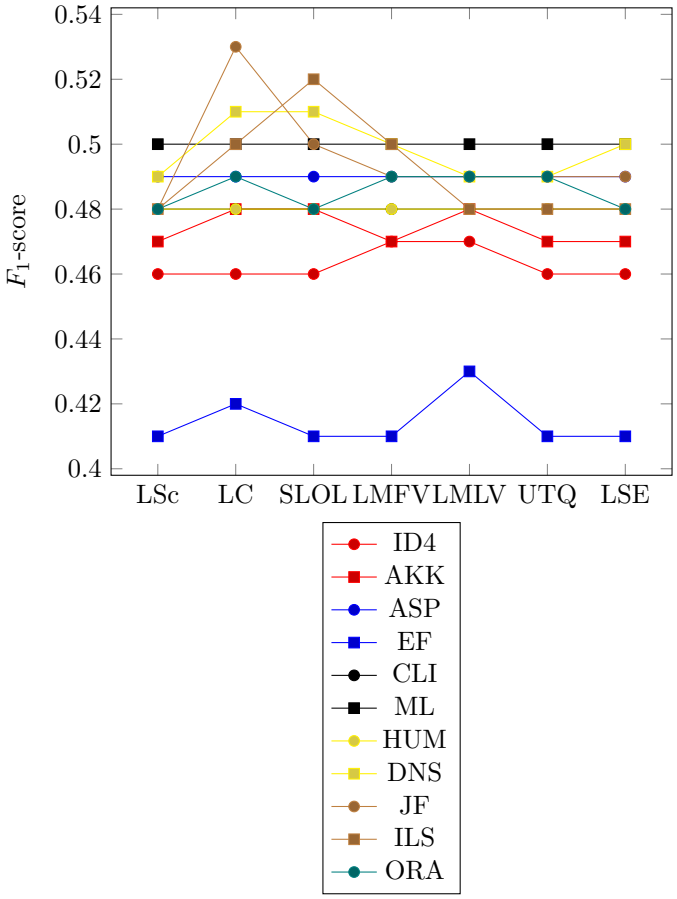

Figure 1: F_1-score Prediction Model: Univariate Regression

In Figure 1 we see m_1 as described in Section 4.1

perform well on the projects: ILSpy and JellyFin. The univariate regression models created for the Entity Framework project, perform relatively bad compared to the other projects. Notable, the SLOC measure performs the best as an isolated measure in the Entity Framework project. Looking at the raw input for the project, we see that only $\frac{1}{5}$ of the classes use lambda expressions, compared to other projects e.g. AKKA.NET $\frac{1}{3}$. The fewer usage of lambda expressions could influence the usability of our measures. The Humanizer project seems to score an almost stable 0.48. When looking into the raw output data from our static analysis we see only $\frac{1}{9}$ classes in this project uses lambda expressions. Therefore, this project might not be functional enough for our measures to yield any value.

To evaluate the value of our candidate measures compared to our baseline, we create a multivariate regression model based on K-Best features for each of the projects. In Table 8 is shown how many out of 11 K-best models included the corresponding measure as a feature.

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

Table 2: Inclusion Candidate Measures K-Best Model

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

To do a comparison between our baseline model and the selection of the K-best features model, the projects are plotted in Figure 2.

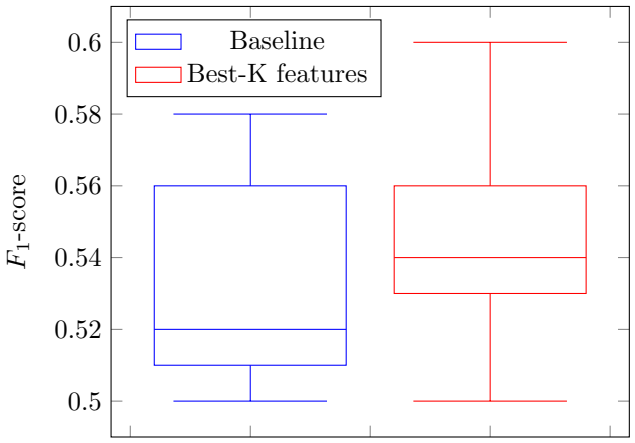

Figure 2: F_1-score Prediction Model: Multivariate Regression

Looking at Figure 2 one can see that the worst-performing project did not gain improvement in performance. However, the first quartile had a performance increase of 0.02. The best performing project also has a 0.02 increase in performance.

9 Threats to validity

- **Generalizability** At this point in the research, only 11 open-source projects were analyzed. For the results of the research to be useful, substantiated claims need to be made about the generalizability of the results. Since we only processed 11 projects, we are unable to make any claims regarding that aspect. By analyzing a bigger corpus of projects, one could make an easier distinction on what different types of projects our proposed measures yield value.
- **Bug fixes vs Bugs** To make an estimation on how error-prone a class is, the assumption was made that bug-fixes made in a class measure the error-proneness of the class. However, there is no way to guarantee that a class that was never updated by a bug-fix is bug-free. Bugs might have not been identified yet, or maybe bugs that were never fixed by a bug-fix commit were accidentally fixed during a refactoring.

10 Related work

Uesbeck [USH+16] did a control experiment where the impact of lambdas in C++ on productivity, compiler errors, percentage of time to fix the compiler errors. The results show that the use of lambdas, as opposed to iterators, on the number of tasks

completed was significant. *"The percentage of time spent on fixing compilation errors was 56.37% for the lambda group while it was 44.2% for the control group with 3.5 % of the variance being explained by the group difference."*. Where the groups consisted of developers with different amount of programming experience.
The increased time of fixing compiler error where lambda functions were used, which seems likely to be the result of lambda expressions being harder to reason about. Which strengthens our hypothesis.
Finifter [FMSW08] shows how verifiable purity can be used to verify high-level security properties. By combining determinism with object-capabilities a new class of languages is described that allows purity in largely imperative programs.

11 Conclusion and Discussion

We investigated the evolution of the OO language C# and what features inspired by the FP paradigm are added. The development and introduction of the FP inspired features seem to be going rapid and there is no indication of the development slowing down. This new declarative syntax enables more concise code constructs. Therefore, enables the software engineer to write more functionality with less code. However, these constructs are introduced without the constraints that would be present in FP languages. Therefore, impure usages of concepts that were designed pure, exist in the OO language C#. To cover the new type of complexity introduced by these FP inspired constructs and impure usages of these constructs, we defined measures. The defined measures cover the following FP inspired constructs: lambda expression usages and impure usages, in which the expression its evaluation is affected by or affects the outside state. Furthermore, we defined a measure to report the unterminated collection queries.

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

So we did find a correlation between our measures and error-proneness. But the presence of this correlation is too uncertain to make claims about causality.

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

References

[0xd16] 0xd4d. dnspy. https://github.com/0xd4d/dnSpy, 2016.

[Akk14] AkkaDotNet. Akka.net. https://github.com/akkadotnet/akka.net, 2014.

[BBM96] Victor R Basili, Lionel C. Briand, and Walcélio L Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, 1996.

[BEEM95] Lionel Briand, Khaled El Emam, and Sandro Morasca. Theoretical and empirical validation of software product measures. *International Software Engineering Research Network, Technical Report ISERN-95-03*, 1995.

[BMW02] Lionel C Briand, Walcelio L. Melo, and Jurgen Wust. Assessing the applicability of fault-proneness models across object-oriented software projects. *IEEE transactions on Software Engineering*, 28(7):706–720, 2002.

[Car] Pierre Carbonnelle. PYPL. http://pypl.github.io/PYPL.html. Accessed: 2019-01-11.

[CK94] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[FMSW08] Matthew Finifter, Adrian Mettler, Naveen Sastry, and David Wagner. Verifiable functional purity in java. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 161–174. ACM, 2008.

[For09] Chris Forbes. OpenRA. https://github.com/OpenRA/OpenRA, 2009.

[Fou14] .NET Foundation. Entityframeworkcore. https://github.com/aspnet/EntityFrameworkCore, 2014.

[Fou15a] .NET Foundation. Aspnetcore. https://github.com/aspnet/AspNetCore, 2015.

[Fou15b] .NET Foundation. Cli. https://github.com/dotnet/cli, 2015.

[Fou15c] .NET Foundation. Identityserver4. https://github.com/IdentityServer/IdentityServer4, 2015.

[Fou18] .NET Foundation. Machine learning. https://github.com/dotnet/machinelearning, 2018.

[GFS05] Tibor Gyimothy, Rudolf Ferenc, and Istvan Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software engineering*, 31(10):897–910, 2005.

[Git19] GitHub. Closing issues using keywords. https://help.github.com/en/articles/closing-issues-using-keywords, 2019.

[HJLS13] David W Hosmer Jr, Stanley Lemeshow, and Rodney X Sturdivant. *Applied logistic regression*, volume 398. John Wiley & Sons, 2013.

[HKV07] Ilja Heitlager, Tobias Kuipers, and Joost Visser. A practical model for measuring maintainability. In *Proceedings of the 6th International Conference on Quality of Information and Communications Technology*, pages 30–39. IEEE, 2007.

[ics09] icsharpcode. OpenRA. https://github.com/icsharpcode/ILSpy, 2009.

[Jel13] Jellyfin. Jellyfin. https://github.com/jellyfin/jellyfin, 2013.

[K+95] Ron Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, volume 14, pages 1137–1145. Montreal, Canada, 1995.

[Kha12] Mehdi Khalili. dnspy. https://github.com/Humanizr/Humanizer, 2012.

[Lan17] Erik Landkroon. Code quality evaluation for the multi-paradigm programming language Scala, 2017. University of Amsterdam.

[LV97] Filippo Lanubile and Giuseppe Visaggio. Evaluating predictive quality models derived from software measures: lessons learned. *Journal of Systems and Software*, 38(3):225–234, 1997.

[McC76] Thomas J McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, (4):308–320, 1976.

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

[NDRTB07] Vu Nguyen, Sophia Deeds-Rubin, Thomas Tan, and Barry Boehm. A SLOC counting standard. In *Cocomo ii forum*, volume 2007, pages 1–16. Citeseer, 2007.

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

[RT05] Chris Ryder and Simon J Thompson. Software metrics: measuring Haskell. In *Proceedings of the Sixth Symposium on Trends in Functional Programming*, pages 31–46, 2005.

[SL09] Marina Sokolova and Guy Lapalme. A systematic analysis of performance measures for classification tasks. *Information Processing & Management*, 45(4):427–437, 2009.

[Son19] SonarQube. Metric definitions. https://docs.sonarqube.org/latest/user-guide/metric-definitions/, 2019.

[Sto74] Mervyn Stone. Cross-validatory choice and assessment of statistical predictions. *Journal of the Royal Statistical Society: Series B (Methodological)*, 36(2):111–133, 1974.

[USH+16] Phillip Merlin Uesbeck, Andreas Stefik, Stefan Hanenberg, Jan Pedersen, and Patrick Daleiden. An empirical study

on the impact of C++ lambdas and programmer experience. In *Proceedings of the 38th International Conference on Software Engineering*, pages 760–771. ACM, 2016.

[VdB95] Klaas Van den Berg. Software measurement and functional programming. *University of Twente*, 1995.

[Wag17] Bill Wagner. Language Integrated Query (LINQ). https://github.com/dotnet/cli, 2017.