Software Maintenance Experiments with the A+ Programming Language and the Primitive Obsession Bad Smell

Ph.D. Dissertation

by
Péter Gál

Supervisor:
Dr. Ákos Kiss

Doctoral School of Informatics
Department of Software Engineering
Faculty of Science and Informatics
University of Szeged

Szeged
2022

"I know nothing about surpassing others. I only know how to outdo myself."

— Bushido Code

Foreword

In one moment, you are playing with puzzle games and LEGO bricks.

Then you blink, and time moves forward.

Now you are coding your first program, solving it like a puzzle, building it up like a LEGO construct. What goes where? How will it work? The questions to be resolved are popping up one by one as more and more pieces are fitted together.

Blink again.

The code became more complex, it now has more components, and more puzzles appeared that need to be solved, more "bricks" to be placed. An endless route on which puzzles are solved, bricks are arranged, and new ones are created constantly.

Over the course of these puzzles, I was fortunate to have support from many people. Mainly from my Mother, my Grandmother, and of course, my family, to whom I am and will always be grateful.

Péter Gál, 2022

Contents

List of Figures

List of Tables

Listings

1 Introduction

Software maintenance is an extensive and diverse topic that focuses not only on fixing defects found in applications, but also on software re-engineering, source code analysis, calculation/evaluation of source code metrics, and detection of various code bad smells. Out of these diverse topics, the author focused on two areas, and these two parts are the main thesis points.

Part I explores the A+ language, its unique features and presents a clean-room implementation of the A+ language on top of the .NET framework. Using the original interpreter and the new .NET version, the runtime and source code metrics were compared to see how the two systems perform. Albeit the .NET version is slower in some cases when executing A+ scripts, the option to use .NET components in A+ could levitate these problems simply by directly using .NET methods from A+ scripts. Additionally, it is essential to note that the original A+ implementation had more than twenty years to optimize its runtime. On the other hand, the maintainability aspects of the two projects are quite different. In this case, the .NET version yields better results. In order to extend the capabilities and make the integration with .NET classes straightforward, the A+ language was extended with object-oriented operations.

Part II presents a new metric and its variants to capture an aspect of the Primitive Obsession code smell. Specifically, to measure the over usage of primitively typed function arguments. This new metric group is the Primitive

Enthusiasm (PE) metric. Each of the metrics is formulated and presented in its respective sections. The main idea of the metric was not to give a globally accepted threshold to mark a method – or class – "primitive" but to compare the methods/classes in a given system to each other to see where the irregularities are. These metrics are measured and investigated on a set of systems to see if they find any methods/classes. This evaluation gave the result that there are indeed classes/methods marked by these metrics. With the creation of these metrics, the correlation between these and other code metrics was also investigated to see how strong the connections are. The possibility of using these new metrics for bug prediction is evaluated on multiple systems in two different ways, by doing a cross-project measurement and – a more realistic scenario – a project version-based measurement.

Part I

Experiments with the A+ Programming Language

2 Introduction

A+ is an array programming language [53] inspired by APL. It was created more than 30 years ago to suit the needs of real-life financial computations. However, even nowadays, many critical applications are used in computationally-intensive business environments. Unfortunately, the original interpreter-based execution environment of A+ is implemented in C and is officially supported on Unix-like operating systems.

By implementing a .NET-based version, the lifetime of existing A+ applications can be extended. Additionally, this will allow A+ developers to access .NET components and .NET developers to use A+ code. Furthermore, A+ applications can be hosted on Windows systems, which was previously impossible with the original implementation. After implementing the new runtime, it is worthwhile to compare the two implementations in terms of runtime and source code metrics to see how they compare to each other.

As the new implementation allows interoperability between .NET classes and A+ scripts, the next logical step is to extend this interoperability between the two worlds. For this, the introduction of object-oriented notation and mechanism is a good choice. This would make for an A+ developer effortless to access various classes or methods from the .NET world.

Structure of this part: Chapter 3 briefly discusses the background of A+, the Dynamic Language Runtime, and various related language implementations

and improvements. In Chapter 4 the new A+.NET clean room implementation is presented, which is compared to the original implementation in Chapter 5. Chapter 6 presents the object-oriented extension for A+. These chapters are covered in papers [24], [25], and [23] respectively. Finally, results and final thoughts are summarized in Chapter 7.

3 Background

3.1 The A+ Programming Language

A+ derives from one of the first array programming languages, APL [12]. This legacy of A+ is one of the most notable differences compared to more recent programming languages. While the operations in modern widespread programming languages usually work with scalar values, the data objects in A+ are arrays. This approach allows the transparent generalization of operations even to higher dimensional arrays.

There are more than 60 built-in functions, which are usually treated as operators in other languages. Among these functions are simple arithmetic functions and some other complex ones, like inner product or matrix inverse calculations. All built-in and user-defined functions are treated as first class-citizens, which means that the programmer can assign functions to variables and is also able to pass functions as arguments to other functions. In addition, almost all built-in functions have special non-ASCII symbols associated. This syntax makes the code compact and also allows mathematical-like notations in an A+ code. However, this could pose a challenge for the untrained eye when trying to read and understand the source code.

The language defines a very narrow set of types, it has integers, floats, characters, symbols – denoted by a starting backtick character –, functions,

and a type named box – which is a special type wrapping another type. In A+, even a single number or character is treated as an array. Arrays can be placed in an array if they have the same type. This means putting a character array inside an integer array is disallowed. The programmer would need to convert the character array into an integer array first to be able to put it into the target array. Currently, the original A+ implementation only allows the creation of arrays with a maximum of 9 dimensions, as specified by the language reference [53].

The variables and functions are categorized in a namespace-like concept called contexts. By default, there is a root context that has no name, and there is always an "active" context. Accessing functions and variables can be done in two ways: using a qualified name and an unqualified name. The qualified name has a format `<context name>.<variable name>`, that is the context name, and the variable name is separated by a single dot character. Context names can contain multiple dots, but variable names cannot. The term "unqualified name" is used when the context is not specified for a variable. For example, if a variable is named `example` and it is in the root context, then the qualified name would be `.example`. If the context was the `demo` then the qualified name would be `demo.example`. Whereas, the unqualified name in both cases is `example`. When an unqualified name is used during A+ script execution, the runtime will automatically extend it to a qualified name based on the currently active context. Using the `$cx <context name>` command the developer can change the active context.

Although being a language of mathematical origin, A+ has an unusual rule: all functions have equal precedence, and every expression is evaluated from right to left.

Listing 3.1: *The computation of the sum and product of the first 10 natural numbers in A+*

```
1  (+/a) , ×/a ← 1 + ⍳ 10
```

Listing 3.1 depicts an A+ example where the sum and product of the first ten natural numbers are calculated and placed into an array. In order to understand better this code snippet, we should go through it in steps. The first expression is `⍳ 10`. Using the symbol *iota*) this will generate a 10-element array containing values from 0 to 9. The result of this is then supplied to for the `1 +` expression. In this case, the operation performed is the addition, but

on the left side of the addition is a scalar and on the left side is the previously calculated 10-element long array. In A+, adding a scalar to an array is not a problem. The addition will be done for each element in the array. In this case, the values from 0 to 9 in the array will be transformed into a new 10-element array containing values from 1 to 10. Next is the assignment `a :=` operation. That is, the result of the addition will be stored in the `a` variable. After this, the `*/` operation will do a multiplication for each element in the array provided by the right-hand side of the operation. In this case the `1 * 2 * 3* 4 * 5 * 6 * 8 * 9 * 10` calculation is performed and gives the `3628800` scalar result. The `+/a` operation will calculate the `a` array's sum, returning `55` as result. The last operation done is the `,` which is the concatenation. The result of the sum and product is concatenated into a single array, resulting in the two-element array `55 3628800`. An important note is that the parentheses around the sum are required. Without this, the concatenation would be applied before the sum calculation, and the whole code snippet would return a single scalar result. As presented, this illustrates the right-to-left evaluation order of the language.

This example also demonstrates that in a very compact form, quite complex computations can be easily expressed in this language. The A+ Language Reference provides other examples. Most of them are for financial applications, for example: how to compute the present value at various interest rates in a single line [53, page 62].

3.2 Dynamic Language Runtime

The Dynamic Language Runtime is a set of classes and methods designed to support dynamic languages, like IronPython [51]. The library adds dynamic objects to C# and Visual Basic to support dynamic behaviour in these languages and enable their interoperation with dynamic languages. In addition, it makes it possible to use .NET classes for the dynamic languages implemented on top of .NET.

The DLR adds the following set of capabilities to the .NET Common Language Runtime [51]:

- Expression trees. The DLR uses expression trees to represent language semantics. For this purpose, the DLR has extended LINQ expression trees to include control flow, assignment, and other language-modelling nodes.

- Dynamic object interoperability. A set of classes and interfaces that represent dynamic objects and operations that can be used by language implementers and authors of dynamic libraries. These classes and interfaces include IDynamicMetaObjectProvider, DynamicMetaObject, DynamicObject, and ExpandoObject.

- Call site caching. A dynamic call site is a place in the code where operations `a + b` or `a.b()` can be performed on dynamic objects. The DLR caches the characteristics of a and b (usually the types of these objects) and information about the operation. If such an operation has been performed previously, the DLR retrieves all the necessary information from the cache for fast dispatch.

3.3 Related Works

Even before the introduction of DLR, there were attempts to integrate languages into other runtimes. One of such attempts was the JPython project (now Jython) [34]. This project implemented the Python language in/on Java. In this implementation, the Python code was compiled to Java byte-code. The developers of the port reported a slowdown by a factor of 1.7 compared to the original Python implementation done in C.

Similarly, for .NET, a Python port was also started, but before the components provided the DLR, this proved to be a difficult task and the project was abandoned [33]. In another approach, the execution engine was not ported to .NET but instead, a bridge was created between the framework and the existing interpreter [47].

After a while, another attempt was made to port Python to .NET. In this case, the result was a success, the IronPython project [35] was created. The first public version was considerably slower on some benchmarks, in some cases 100-fold, but in other cases, it was faster than the C implementation. This project paved the way for other dynamic languages on top of .NET the Dynamic Language Runtime components are created in this work. This project is still under active development, and previously its performance was regularly compared to the reference implementation. However, these results are no longer accessible

The Java community is also putting effort into supporting the execution of script languages on the Java platform. Several related JSRs [65, 66] reached final status and got incorporated into public releases of the platform.

APL also had a few attempts to bring the language to the .NET framework. One of the first of these is the VisualAPL [8]. This version of APL departed from the conventional syntax and adopted object orientation, C# syntax and semantics into the language. The first release was in 2009 as a consumer-ready product [9], and currently, it seems that it was abandoned later, as there are no new news or information available. Another project was the APL# [42] based on Dyalog APL. With this approach, the developers and designers have taken cautious steps in order not to introduce language-breaking features. Unfortunately, after a while, this project was also abandoned.

As previously discussed, porting of various languages are a reoccurring task taken up by various developers. Similarly, re-engineering legacy systems have long been in the focus of researchers. In most cases, however, authors concentrated on estimating the cost of such re-engineering work [62] or on using automated tools to transform legacy systems from one source to another [63]. Alternatively, developing a methodological and technological modernization framework to help the migration of legacy systems based on high-level design models [58].

Furthermore, languages improve over time, and new syntax or semantics are added to them. One such "new" element was the object-oriented notation introduced into various languages. Extending an existing language with object-oriented capabilities is not new in the world of programming languages. Even for APL – from which A+ derives – there are object-oriented extensions [11] and other experiments to incorporate object-oriented notations into the APL language [28]. Both of these implementations provide object-oriented notations, allowing the developers to create classes from APL code and not just operations for objects. However, in one case, the authors introduced the member access operation to be read from left-to-right thus making the right-to-left reading mode a bit awkward when such operations are performed. In the other case, the authors introduced a special system function (`⎕NEW`) to construct new instances from a given class. Additionally, the possibility of interacting with other objects – that are outside of the APL language – was not mentioned.

Moreover, the APL language is still in active use, and there are different companies providing support for its implementations, the most notable one is provided by Dyalog Ltd. [15]. This version also provides object-oriented notations to make the developers' life easier. In addition, it also gives a syntax to write classes, not just the option to instantiate them and access its members. The variant took the approach of introducing a new system function to instantiate classes and use the dot syntax to select members on

classes/instances [43].

Other array-based programming languages used in mathematics already have object-oriented support, for example, the widely used Maple [44] and Matlab [48, 59] software packages.

One could also argue why a more than 30 years old language's life should be extended. For this, the COBOL language is a prime example. COBOL was created in the 1960s for data processing and was first used by the U.S. Department of Defense on mainframes [17]. Over the years, the language has gained quite a bit of adoption in various industries, but mainly in financial solutions. The need for a standardized COBOL language quickly became apparent, and this resulted in an ISO standard [36]. This standard was updated multiple times over these years [37, 38] and the latest one is from 2014 [39]. This new standard also introduced object-oriented elements into the language. This led to the current era of computing, where even after almost half a century, some systems are still using COBOL, and there is no short-term plan to change them. For example, "The Social Security Administration's Software Modernization and Use of Common Business Oriented Language" report from the USA states that they do not have a strategy to convert the COBOL systems to a more modernized programming language [64]. Furthermore, a report from 2016 shows [60] that 43% of U.S. banking systems are built on COBOL, 95% of ATM swipes use in some form on COBOL code. The report also describes that 220 billion lines of COBOL code are still in use. This example also shows that in some cases, systems could be used for a very long time and the option to keep them alive is crucial. Furthermore, even in these days, there are books to help learn COBOL [13, 69].

COBOL's case illustrates, that it is difficult and costly to replace an already existing system. A+ is same in this regard, there are various critical financial applications that still uses this language.

4

A+.NET Implementation

Implementation of an interpreter or compiler for any programming language usually requires a few key information. One such important component is the syntax of the language and the other one is the internal working of the interpreter or compiler.

4.1 Defining the Grammar of the A+ Language

Even before the implementation of the .NET-based runtime could have started, we ran into two major problems of the original system. First, it turned out that A+ has no formal grammar. There are only two official sources of information available on the syntax of A+: the Language Reference [53], which gives only a textual description of the language, and the source code of the reference implementation, which contains a hand-written lexer and parser, from which the formal rules are non-trivial to reverse engineer. In addition quality of the source code of the original implementation is a bit questionable. The quality aspects of this are presented in Chapter 5.

Thus, we had to formalize the grammar of A+ first. In order to formalize the grammar, the A+ language reference was extensively investigated and

methodically processed. Furthermore, a multitude of simple and complex grammar tests were created to understand the syntactic and semantic behaviour of the language. In the end, a context-free grammar was constructed which can handle the required language elements as described in the reference document. With the formalized grammar using a parser-lexer generator, most of the A+ source code processing components can be generated. In our case, we chose the ANTLR [56] parser generator framework to generate the required C# classes.

The second major problem was that the language reference and the reference implementation conflicted at quite a few points in semantic matters. There are two types of conflicts present in the original A+. The first one is due to implementation laziness. That is, at quite a few points, the implementation only checked or required a single value to determine what to do, despite the language reference stating otherwise. For example, in the original implementation the system command `$mode` determines whether a string equals to "apl", "ascii", or "uni" based on the second character of the string instead of looking for an exact match. This could lead to dangerous habits for A+ developers if they find out that `$mode "apl"` and `$mode "ipa"` commands are treated the same way by the implementation.

The other category of differences was when the implementation accepted additional inputs for a given operation. Effectively extending the semantics of the operation. For example, the implementation of the pick function accepts not only one-dimensional arrays as documented but multi-dimensional ones as well.

In those cases that fell into the first category, we decided not to repeat the same errors but to follow the language reference. However, in the case of the second category, we chose to accept the extensions in the .NET version since existing A+ applications may rely on their existence.

Once we handled the disturbing syntactic and semantic problems of the original system, we were able to start working on the adaptation to .NET.

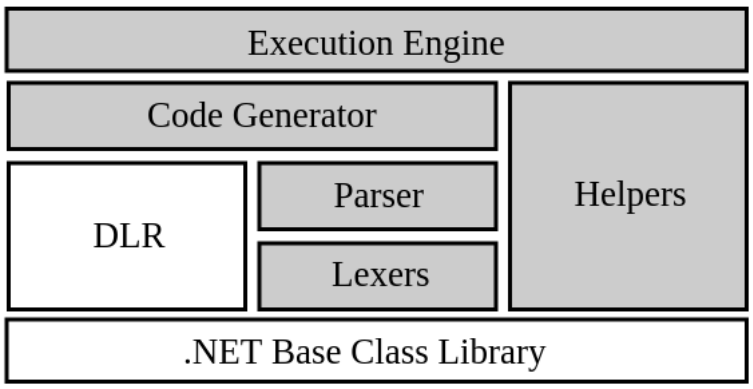
4.2 Architecture Overview

The clean room implementation of an A+ runtime [1] for the .NET environment utilizes the Dynamic Language Runtime (DLR) [51].

In order to provide an insight into how the components of the runtime were designed and implemented, Figure 4.1 depicts main components of the A+ runtime and also describes how the components are layered on top of each other. The white boxes denote components provided by the .NET framework,

including the base class library and the DLR that aids the adaptation of scripting languages to .NET. The shadowed boxes form the system implemented by us. These components build on top of each other as follows.

Figure 4.1: *Components of the A+ .NET runtime*



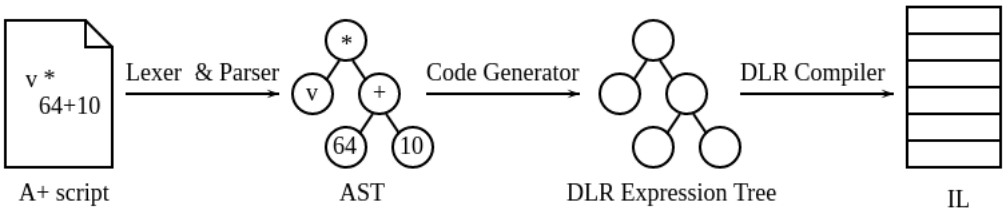
The Lexer and Grammar components can mostly be generated because the A+ grammar was formalized. At first, the .NET implementation only supported the two major lexer modes: APL and ASCII. But over time, the UNI lexer mode was implemented. APL input mode is commonly used by everyday users of the language and requires a special APL font configuration in order for the symbols to be correctly displayed. If this custom font is not available, the ASCII mode is preferred, as this can always be displayed correctly with all commonly used fonts. A small downside of ASCII mode is that it usually requires more characters to specify which operation the developer would like to use. Without the need for a special font, the integration of A+ scripts into C#/.NET makes it easier as the developer only needs the library to get started with A+.NET.

The output of the generated parser is an abstract syntax tree (AST), which is transformed by the Code Generator module into DLR Expression Trees (ET). During transformation, part of the semantics – especially control structures and the structure of statements – are expressed using ETs, while complex functionalities operating on diverse data structures get usually transformed to method calls to various helper functions implemented in C#.

The entry point for the execution of an A+ script is the Execution Engine. This glues together the parser, the Code Generator, and the DLR subsystem by feeding the A+ source code into the lexer-parser, giving the resulting AST

to the Code Generator, passing the generated ET to the compiler of DLR, and finally, calling the compiled executable .NET IL code. In Figure 4.2, we show how the different components of the runtime transform an A+ source code until it becomes executable by the .NET framework.

Figure 4.2: *Compilation steps of an A+ script*



A side effect of the .NET implementation is that the engine can be used not just on Windows-based systems but on systems that support Mono, which is a cross-platform, open source .NET framework [5].

The .NET-based A+ execution engine can be used in two ways. Since DLR provides a command line hosting API, it is very easy to implement a command line tool mimicking the behaviour of the reference interpreter implementation. The biggest advantage of the .NET-based implementation is, however, that it can be embedded into other .NET applications. Moreover, by adding .NET methods into the execution scope of the engine, it is possible to achieve interoperation between A+ scripts and the embedding .NET environment.

4.3 A+ and .NET Integration

One of the advantages of the .NET implementation is that the developer can access the .NET classes and methods from A+ scripts. In order to do this, each value or function that the developer wants to register into the A+.NET runtime must be wrapped into an `AType` and added into the engine's runtime scope. This makes it possible for the runtime to find and use these values. There are five types of values that can be registered: numbers – integer and double –, characters, symbols, and functions. The A+.NET types for these are `AInt`, `AFloat`, `AChar`, `ASymbol`, and `AFunc` respectively. Naming the double type as float might seem strange at first sight, however, floats represent double precision floating point number in A+ terminology. Thus, `AFloat` adheres to the original naming conventions. Listing 4.1 depicts how someone can add the

constant `3.2` into the A+.NET runtime under the name `tst.value`. The first line exemplifies the creation of a new scope in the runtime, which stores, all methods and variables and can be used when executing A+ code from .NET. Character, symbol, and arrays of these types can be added in the same way to the scope. Each A+ .NET type has its own `Create` method to ease the creation of these values.

Listing 4.1: *Example registration of the constant 3.2 into an A+.NET scope in C#*

```
1 var scope = engine.CreateScope();
2 scope.SetVariable("tst.value",
3                   AFloat.Create(3.2));
```

However, functions require a bit of special handling. As each method that the developer wants to add into the scope must adhere to some rules: First, the method must be a static method. Second, the return type must be `AType`, which is the base interface type for all types in the runtime. Third, the first argument must be an `Aplus` type, which contains the runtime environment information and can be accessed by the method. And fourth, any other arguments must be of `AType` type and – most importantly – they must be in reverse order. The reverse order is required because the A+ language evaluates function arguments from right to left while C# does not. Thus, in the A+.NET runtime, we perform a trick and require all methods to have a reverse order of arguments.

So for an A+ function that accepts two parameters, the second argument of the A+ function becomes the first non-environment argument of the registered C# method, and the first argument in A+ will be the last parameter in C#. In Listing 4.2, a conforming method is shown, which meets all the requirements for the A+.NET.

Listing 4.2: *An A+.NET compatible C# method*

```
1 static AType Foo(Aplus env,
2                   AType arg2, AType arg1) {
3     // ...
4 }
```

There are two ways of adding a compatible method into the A+.NET runtime. The first one is to create a scope – which contains the variables – for

the A+.NET engine/runtime and then add the required method as a variable into the scope via the scope's `SetVariable` method. The function must be wrapped into an `AFunc` which contains information about the callable method for the runtime. This approach is shown in Listing 4.3, where method `Foo` from Listing 4.2 is registered into the scope as `tst.foo`.

Listing 4.3: *Registering a C# method into an A+.NET scope*

```
1 var scope = engine.CreateScope();
2 AFunc fun = AFunc.Create("foo", Foo, 2, "Test");
3 scope.SetVariable("tst.foo", fun);
```

The second option is to create a class with the compatible methods and annotate all functions as `AplusContextFunction` and the class as `AplusContext`. The annotation `AplusContext` specifies the context name under which the methods should be registered and is part of the A+.NET runtime. The function annotation specifies the name by which the method should be accessible from A+. Such annotated classes can be loaded via the `$load` system command specifying the name used in the annotation of the class. Listing 4.4 depicts a class with all the annotations. For this specific case, the `$load tst` instruction in A+ will load the method into the context named `tst` with the name of `foo`.

Listing 4.4: *A+.NET conformant C# method with annotations, registered as `tst.foo` in the A+ environment*

```
1 [AplusContext("tst")]
2 class Demo {
3     [AplusContextFunction("foo")]
4     static AType Method(Aplus env,
5                         AType arg2, AType arg1)
6     {
7         // ...
8     }
9 }
```

Under the hood, the load function will traverse the DLL files currently loaded and will search for the `AplusContext` and `AplusContextFunction` annotations. If such classes and functions are found, the runtime will internally

call the same `SetVariable` method on the current scope as mentioned in the previous case.

After we have loaded the required function(s) into the A+.NET runtime – in any of the two ways described above –, it is possible to invoke them just like any other A+ functions. For example, the previously registered method `tst.foo` can be used as shown in Listing 4.5.

Listing 4.5: *Invocation of a registered C# method from A+.NET*

```
1 tst.foo{1;2}
```

As visible from the explanations and examples above, adding a method to the A+.NET runtime requires writing a lot of code. Writing these wrapper methods or classes for each required .NET class is tedious and error-prone, not to mention that there could be a lot of copy-paste code in the end.

Instead of writing these wrappers or providing tools to generate them, a better way would be to have a mechanism that allows runtime access for the required classes, methods, properties, and variables.

5 Comparing A+ Implementations

As the .NET version of A+ is now available, a comparison can be made to see how the two engines perform. First, a run time comparison can be easily made. Just provide the same input for both engines and time the executions. Of course, this is a crude way to measure execution time, but this will contain everything: engine setup, engine overhead, parser/lexer overhead, and finally, the execution of operations defined in the input script.

The second comparison is made by looking at the source code and source code metrics from them. However, there are several difficulties here that need to be handled. Although both implementations aim at doing the same thing, i.e., executing A+ scripts according to the language specification, their architecture and their internal logic are completely different. This is expected as the original interpreter was written in C and the .NET version in C#.

5.1 Run Time Experiments

Although our primary goal for the initial implementation of the .NET-based runtime was to make its observable behaviour as equivalent to the reference implementation as possible and, thus, we did not focus especially on optimizations, we still wanted to get preliminary results on its runtime performance. Thus, we extracted a code fragment from a real-life code base and extended

it with some code performing execution time measurement. Listing 5.1 shows the test A+ script, where lines 1–20 implement URL encoding of strings, and lines 22–28 drive the encoding by feeding a set of URLs to the routines (and repeating this 300 times) and additionally, measure the elapsed time. The input for the encoding has been collected during a browsing session and contains 50 URLs of length ranging from 60 to 1439 characters.

Listing 5.1: *The A+ script used for performance evaluation (written in APL input mode).*

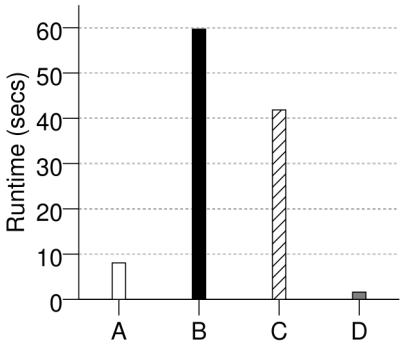
```
1  uri.AN ← { "ABCDEFGHIJKLMNOPQRSTUVWXYZ",
2             "abcdefghijklmnopqrstuvwxyz",
3             "0123456789-_.~" };
4
5  string.join{char; strlist}: {
6    if (0=#strlist) ← "";
7   ← (-#char)↓ ⊃ strlist,¨ <char;
8  }
9
10 uri.encodechar{char; ignore}: {
11   if ((char=' ') ∧ (' ' ∈ ignore)) ← '+';
12   if (char ∈ uri.AN) ← char;
13   if (char ∈ ignore) ← char;
14  ← '%',(16 16 ⊤ `int?char)#"0123456789abcdef";
15 }
16
17 uri.encode{ascii; ignore}: {
18   bts ← uri.encodechar ¨ {ascii; <ignore};
19  ← string.join{''; bts};
20 }
21
22 uri.encodeD{ascii}: ← uri.encode{ascii; ""}
23
24 start ← time{}
25 300 do { uri.encodeD ¨ data; }
26 end ← time{}
27
28 ↓ 'elapsed: ', ⍕ end[2] - start[2]
```

For our experiments, we used a computer equipped with a Dual Core AMD Opteron 275 2.2GHz CPU and 4GB RAM. During the evaluation, the reference implementation of the A+ interpreter (version 4.22) acted as a baseline for comparison, which was executed on a 32-bit Debian Squeeze Linux installation.

Our .NET-based implementation (revision 231) was ran on Windows 7 (32-bit) and .NET framework 4.0 by embedding the execution engine into an application utilizing the command line hosting API of DLR.

In our first experiment, we compared the execution time of the A+ script as measured on the reference implementation and on the .NET-based interpreter. The result of the comparison is shown in the first two columns (A and B) of Figure 5.1. According to the measurements, the reference implementation is about 7 times faster than the .NET port, currently.

Figure 5.1: *Execution times of the A+ test script A) on the Linux reference implementation, B) on the .NET implementation, C) on the .NET implementation with* `string.join` *replaced, and D) on the .NET implementation with* `uri.encode` *replaced.*



At first, the original A+ implementation shows a big lead compared to the A+.NET implementation. An important note here is that the original implementation had more than ten years to do any kind of optimization in their engine whilst the .NET implementation does not have any optimization in it by default. However, we also experimented with the interoperability between A+ and .NET. Since the .NET Base Class Library contains equivalents of `string.join` and `url.encode`. We performed two additional experiments where we removed the A+ implementation of these functions from the test script (lines 5–8 in the first experiment, lines 1–20 in the second) and added their .NET counterparts to the scope of the engine before execution. The result of these experiments is shown in the last two columns (C and D) of Figure 5.1. The replacement of `string.join` by its .NET equivalent resulted in a nearly 30% speedup, even though this version is still slower than the

reference implementation. However, the replacement of `url.encode` yielded a huge performance gain. The execution time in this experiment dropped to 20% of the time measured for the reference implementation, which is equivalent to a 5-fold speedup. This latter result shows one of the possible exploitation of the interoperability, i.e., performance improvement of existing A+ applications by adding .NET implementations for critical code parts.

Speeding up the execution of A+ scripts by calling .NET routines is not the only interesting application of the runtime. Because of the mathematical expressive power of A+, it can be used in .NET code as an embedded domain-specific language. Since A+ is mostly used in financial computations, we illustrate the usefulness of the embedding with the mixed-language implementation of the oft-cited Black-Scholes formula [7] used to calculate the price of European put and call options. Listing 5.2 shows a code fragment where mathematical formulas are written in A+, but control structures are in the host C# language.

Listing 5.2: *A C# code fragment embedding A+ expressions (written in ASCII input mode).*

```
1  aplusEngine.Execute(@"CND{x}: {
2    a := 0.31938153 -0.356563782 1.781477937 -1.821255978
         1.330274429;
3    L := |x;
4    K := %1 + 0.2316419 * L;
5    R := 1 - (%(2 * pi 1) ^ 0.5) * (^-(L ^ 2) % 2) * (a +.* K ^ (1 +
         iota 5));
6    := if(x < 0) 1.0 - R else R }", scope);
7
8  aplusEngine.Execute(
9    "d1 := ((log (S % X)) + T * (r + (v^2) % 2)) % (v * T ^ 0.5)",
         scope);
10 aplusEngine.Execute("d2 := d1 - v * T ^ 0.5", scope);
11
12 String script;
13 if (option == "call") {
14     script = "(S * CND{d1}) - (X * ^(-r * T) * CND{d2})";
15 } else {
16     script = "(X * ^(-r * T) * CND{-d2}) - (S * CND{-d1})";
17 }
18
19 result = aplusEngine.Execute(script, scope);
```

5.2 Source Code Metrics

In order to do a meaningful comparison between the two runtimes, the functionally equivalent parts should be selected. The modules of the .NET implementation, their interrelation, and their behaviour are described in Section 4. However, the reference implementation has no available documentation on internal components other than the source code itself. The only meaningfully deducible model for modules is the directory layout of the source code files.

Other difficulties include the difference in the languages of implementation: the reference interpreter is written in C/C++, while A+.NET in C#. And finally, we admit that the .NET-based version is far from being as functionally rich as the reference implementation. These issues prevent matching lines, functions, classes, or even files directly to each other in the two code bases. Thus, we had to find a way to make our analysis unbiased. We decided to determine the functionally equivalent parts of the two implementations and perform the comparison on those parts.

Determining the functionally equivalent parts required a reasonably large and diverse A+ code base to drive the engines and a way to track which parts of the engines were exercised by the test inputs. Fortunately, during the development of A+.NET, unit tests were written for almost every implemented functionality. At the time of writing the related paper [24], this was 1778 test cases, of which 1719 tests – consisting of about 2300 lines of A+ code – could be used as input to both A+ execution engines.

For the reference implementation, we used the instrumentation support of GCC. Recompiled the interpreter and instructed GCC to instrument every function at its entry point with a call to a routine that determines (with the help of the libunwind library [54]) the name of the called function at execution time and dumps it out into a log file. Then, this instrumented interpreter was used to execute the A+ test scripts and by analysing the log files. With this approach, we were able to determine which functions were called.

For the .NET version, we used the built-in functionality of Visual Studio that can collect the executed methods during a test session in order to gather these code coverage results.

Tables 5.1 and 5.2 show the modularization of both systems and data about the size and the test coverage ratio of each module. The reference interpreter (version 4.22) has quite a large code base of C and C++ files. It consists of 153,990 lines of code and 102,924 statements in 18,981 functions (class methods and global functions included). This means that its code is 8.87-14.38x larger

Table 5.1: *Modules of the A+ reference interpreter, the size of each module (given as NF – number of functions, LOC – lines of code, and NOS – number of statements), and the size and ratio of those code parts that are exercised by the test suite (The granularity of the code coverage information is functions.)*

Module	Size Metrics			Test Coverage					
	NF	LOC	NOS	NF		LOC		NOS	
a	819	7716	12578	576	(70.33%)	5083	(65.88%)	8546	(67.94%)
cxb	26	480	484	1	(3.85%)	18	(3.75%)	15	(3.10%)
cxc	66	1073	877	2	(3.03%)	38	(3.54%)	32	(3.65%)
cxs	1	13	8	0	(0.00%)	0	(0.00%)	0	(0.00%)
cxsys	82	1802	1501	5	(6.10%)	123	(6.83%)	108	(7.20%)
dap	227	4094	2466	18	(7.93%)	344	(8.40%)	208	(8.43%)
esf	201	3690	3494	16	(7.96%)	155	(4.20%)	132	(3.78%)
main	17	425	322	15	(88.24%)	272	(64.00%)	209	(64.91%)
AplusGUI	3328	23761	17041	72	(2.16%)	1768	(7.44%)	1576	(9.25%)
IPC	303	2680	2358	12	(3.96%)	52	(1.94%)	47	(1.99%)
MSGUI	8938	78268	44858	16	(0.18%)	110	(0.14%)	53	(0.12%)
MSIPC	399	2298	1344	28	(7.02%)	261	(11.36%)	156	(11.61%)
MSTypes	4574	27690	15593	100	(2.19%)	431	(1.56%)	179	(1.15%)
TOTAL	18981	153990	102924	861	(4.54%)	8655	(5.62%)	11261	(10.94%)

Table 5.2: *Modules of A+.NET runtime, the size of each module (given as NF – number of functions, LOC – lines of code, and NOS – number of statements), and the size and ratio of those code parts that are exercised by the test suite. (The granularity of the code coverage information is functions. The table does not include data on those parts of the lexers and the parser that are generated.)*

Module	Size Metrics			Test Coverage					
	NF	LOC	NOS	NF		LOC		NOS	
Code Generator	385	5008	1706	217	(56.36%)	4003	(79.93%)	1231	(72.16%)
Execution Engine	87	639	284	55	(63.22%)	355	(54.56%)	176	(61.97%)
Helpers	1213	11396	5020	909	(74.94%)	9171	(80.48%)	4183	(83.33%)
Lexers	5	68	41	3	(60.00%)	48	(70.59%)	29	(70.73%)
Parser	34	255	104	22	(64.71%)	227	(89.02%)	98	(94.23%)
TOTAL	1724	17366	7155	1206	(69.95%)	13804	(79.49%)	5717	(79.90%)

than the code base of A+.NET (revision 232), depending on which code size metrics we compare. It is also visible from the tables that a big portion of the reference interpreter is not covered by the tests. This is not a surprise since the scripts were written as unit tests for the A+.NET system to cover its functionality. However, if we consider that the major part of the functionality of all the modules of A+.NET corresponds to module 'a' of the reference interpreter and only parts of the 'Helpers' module contain code that match other modules of the reference implementation, the coverage results become much closer to each other: the function-level coverage ratio is circa 70% for both the whole A+.NET system and module 'a' of the reference interpreter. Moreover, the size metrics of the covered code do not differ as much as they do in the case of the whole code bases: in the reference implementation, 8,655 lines of code and 11,261 statements were covered by the tests in 861 functions, while the same data for A+.NET is 13,804 lines, 5,717 statements in 1,206 functions. This means that we managed to identify two function sets, one in each system, of comparable size and of equivalent functionality, which can form the basis of our further investigations.

5.3 Maintainability Metrics

Once we identified the functionally equivalent parts of the two A+ execution environments, their comparison became possible. We used the Columbus tool chain [18] to analyze the sources of the two systems and to compute such maintainability-related metrics for the covered functions, which were defined for both implementation languages. As a result, we got two size metrics – (executable) lines of code (LOC) and number of statements (NOS), those that were already presented in Tables 5.1 and 5.2 – and two complexity metrics – McCabe's cyclomatic complexity (McCC) and nesting level (NLE) – for each function.

Two out of the four metrics above are traditional and well-known: LOC is one of the easiest metric to compute (but often still very informative) and McCC [49] is also often used. For the sake of completeness, however, NOS and NLE are explained below. NOS counts the number of control structures (e.g., if, for, while), unconditional control transfer instructions (e.g., break, goto, return), and top-level expressions in a function. This definition makes NOS capture a more syntax-oriented concept of size than LOC (at least for languages where the concepts of line and statement are not related). The NLE metric determines the maximum number of the control structure depth in a

function.

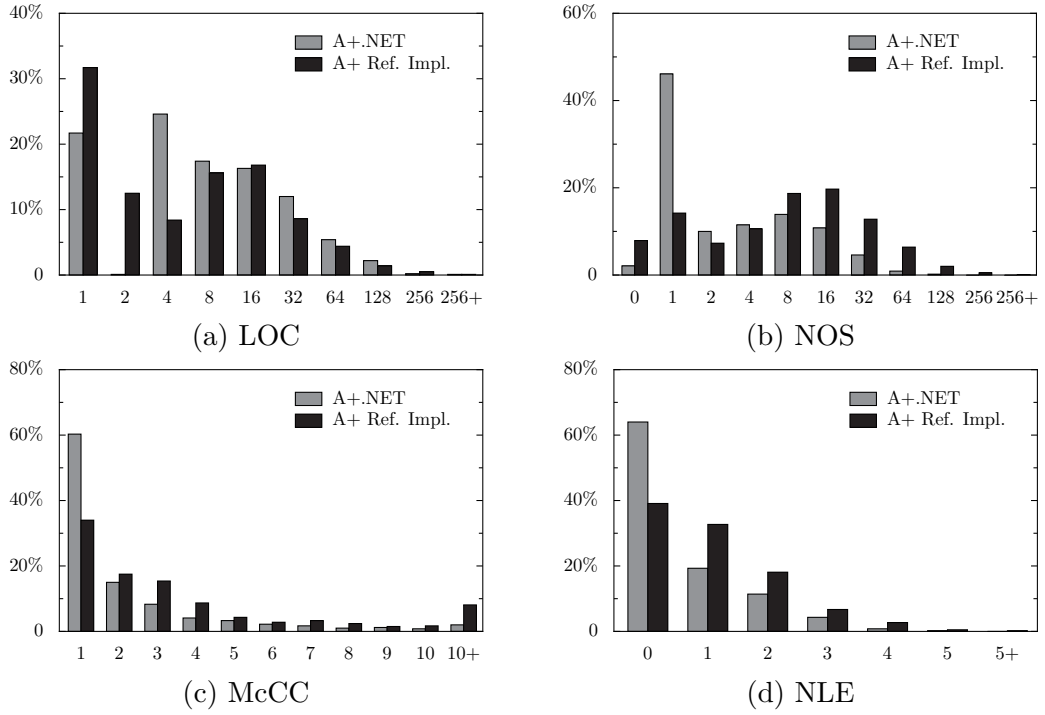
Table 5.3: *Maintainability-related metrics measured for the A+ reference interpreter and A+.NET*

Metric	Reference Interpreter			A+.NET		
	min	avg	max	min	avg	max
LOC	1	10.07	375	1	11.46	275
NOS	0	13.08	372	0	4.73	71
McCC	1	4.45	123	1	2.38	71
NLE	0	1.04	6	0	0.59	5

Table 5.3 presents the aggregated metrics for both systems. The averages of NOS, McCC, and NLE and the maximums of all metrics show that the size and the complexity of the functions in the reference implementation are higher (sometimes significantly larger, see NOS) than in A+.NET, which is usually an accepted mark of lower maintainability. The only outlier is the average of LOC values, where the reference implementation produces a lower (i.e., better) metric. However, the difference in the case of this metric is much smaller (a factor of 1.14 only) than for the others.

In addition to the aggregated results, Figure 5.2 shows the histograms of the computed metrics. In the case of the size metrics, the histograms use exponentially growing intervals with numbers on the horizontal axis denoting the upper bound of the interval. For the complexity metrics, the intervals are of equal size. The vertical axis denotes the percentage of functions falling in a given interval. The histogram of LOC explains why the aggregations of the metric do not give a conclusive result. Whether the relative number of functions falling into a size category (interval) is larger in the reference implementation or in A+.NET is almost alternating. However, the histograms of the other three metrics, especially NOS and McCC, strengthen the hypothesis that the reference interpreter is larger and more complex. For A+.NET, a larger portion of functions falls into the lower ranges of the metrics than for the reference implementation, while in the upper ranges, the reference implementation dominates. We can observe that in A+.NET, circa 60% of the functions have two statements at the maximum, while in the reference

Figure 5.2: *Histograms of the computed maintainability-related metrics.*



interpreter, on the contrary, circa 60% of the functions have more than four statements. For the McCabe complexity, we can see that while in A+.NET only 2% of the functions have a metric value higher than 10, in the reference interpreter 8% of the functions can be found in the same complexity range.

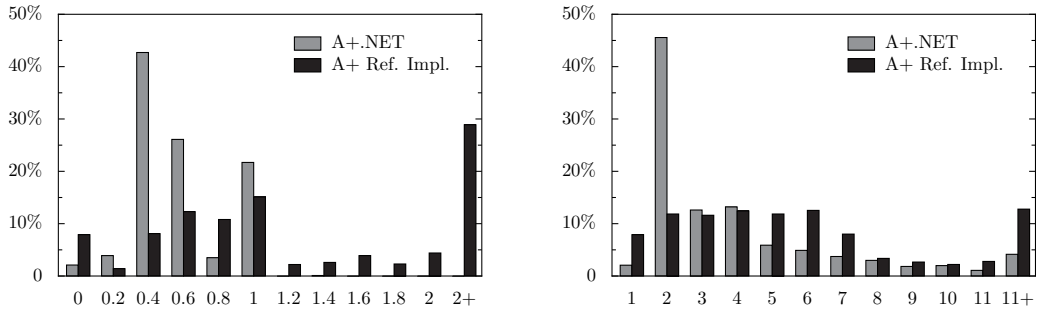
Additionally to the metrics that are directly computed by the Columbus toolchain from the sources, we experimented with derived metrics as well. Our original plan was to compute the Maintainability Index (MI) [55] for both systems, but unfortunately, the version of the toolchain at that time was unable to compute the Halstead Volume metric [32], that is a component of MI. Thus, the analysis of MI is left for future work. We compute two simpler formulae that grasp some key differences between the reference implementation and A+.NET. The first formula is NOS/LOC: this derived metric tells the average number of statements written in a single line. Guidelines normally suggest one statement per line to keep the code readable and comprehensible. The second formula is $\mathrm{McCC} + \ln(1 + \mathrm{NOS})$: combining the complexity and the size of a

function into a single number. This metric is motivated by the Maintainability Index, but the currently unavailable Halstead Volume component is left out, and the scale is inverted: now, higher values represent larger code size and/or higher complexity, thus – presumably – lower maintainability. As this metric combines McCC and NOS, the only way a function can have a value of 1 is when the McCC is one, and the NOS is zero. This means only empty functions could get this small value.

Table 5.4: *Derived metrics computed for the A+ reference interpreter and A+.NET.*

Metric	Reference Interpreter min / avg / max	A+.NET min / avg / max
NOS/LOC	0 / 2.94 / 37	0 / 0.50 / 1.40
McCC + ln(1 + NOS)	1 / 6.42 / 128.36	1 / 3.72 / 75.28

Figure 5.3: *Histograms of the derived metrics.*



(a) NOS/LOC (b) McCC + ln(1 + NOS)

The derived metrics were also computed for each investigated function, and their aggregated data is shown in Table 5.4. For all averages and maximums, A+.NET scores considerably lower than the reference interpreter. Moreover, the statements per line metric of the reference interpreter are stunning: the average number of statements in every executable line of source code is about

3, and the most "crowded" function contains 37 top-level statements in a line on average! Manual investigation revealed that this extreme function consisted of a single line only. However, multi-line functions with a high NOS/LOC ratio are not uncommon either. Actually, circa 30% of the investigated functions of the reference interpreter have more than two statements on a line on average, as depicted in Figure 5.3.

In case of the McCC + ln(1 + NOS) metric, circa 70% of the the A+.NET runtime's functions fall in the 1-4 range. This means most of the functions are small and less complex. Whereas in the case of the reference implementation, this number is around 40%. Based on the averages, maximum values and the histogram data, it can be concluded that the A+.NET implementation performs better in terms of maintainability.

6

A+.NET Language Extension

As described in the previous Section 4.3, exposing methods into A+ scripts requires a bit of boilerplate code. To improve the situation on this, we have reviewed the object-oriented concepts and investigated the requirements for the A+ language to handle external objects conveniently, as the language itself is not an object-oriented one by design.

The first step for this was the investigation of the required operations to handle objects in a language. During the investigation, the base requirements to handle objects in a language were identified. First, the introduction of a way to represent objects in the runtime was required. This essentially means that a new type should be added to the language.

Second, there are four basic operations that should be supported by a language to handle the most basic tasks on objects. These are the following:

- Accessing members (methods, variables, and properties): this operation provides the means to read variables, properties, and to access methods.

- Modifying variables and properties: the operation makes it possible to assign new values to properties and values.

- Type casting: an important operation to resolve ambiguities.

- Using indexer properties: in .NET, indexing objects is done via a special property, and there are compound types where there is no other way of accessing elements, e.g., in ArrayList. This is mainly required for the .NET binding.

Note that the method invocation is not among the operations. This is because most languages already support method calls in some way. So the existing method invocation syntax and semantics can be improved to handle the invocation of methods on instances and classes.

Also, note that with the operations above, the language will still not support classes natively. That is, it is not possible to write classes in the same language for which the operations are implemented. Fortunately, the described operations take the language closer to the real object-oriented languages. So it becomes possible to build further object-oriented functionalities on top of these new operations. Also, note that the concept of these operations is not tied to one specific language, in our case, to A+. Thus, these could be added to other non-object-oriented languages as well.

These four operations are a must for the A+.NET runtime to handle external objects. In the case of A+.NET, the implementation of the required operations can take advantage of the reflection capabilities of the .NET framework and also the code generation possibilities of the Dynamic Language Runtime (DLR) framework. The use of reflection provides ways to search for classes and methods, and the code generation capabilities of DLR enable the building of wrapper method(s) at runtime. With these two techniques combined, it is possible to provide general functions for A+.NET that can perform the required lookups and code generations, essentially avoiding unnecessary and repetitive manual coding. The most important requirement for the language extension is that it must take into account the language's most unique aspects, which is the order of evaluation. The main reason for this is to not break existing code by adding new precedence into the language.

As mentioned previously, a new type must be added to the runtime. In A+.NET, this new type is internally named `AObject`, and its instances can store any .NET object. The subsequent sections will define each previously mentioned operation in more detail.

6.1 Accessing Methods, Variables, and Properties

In A+, functions are first-class citizens, thus it must be ensured that each .NET method can also be used accordingly and not just for simple method invocation. Fortunately, this lenience is a win in our case as this will allow us to handle the methods, variables, and properties in a similar way, thus simplifying the implementation.

The following algorithm describes the inner workings of the accessor operation. This operation has two input arguments, the name of the property, variable, or method to be looked up and the instance or class on which the lookup should be performed, and can be formulated as `SelectMemeber(x, y)`. In order to not interfere with the context concept of the A+ language, both arguments of the operation must be of a symbol type. Steps preformed by the algorithm:

1. Collect all methods, variables, and properties of the instance or class specified by the second parameter (that is `y`).
2. For each method, variable, and property, check if it has the same name as specified by the first parameter (that is `x`).
3. If no match is found, return an error, reporting that there is no member accessible with the given name.
4. If a variable or property was found, return it as an A+ type. In the case of a primitive type, such as string, number, or enum value, they are converted into their compatible A+.NET counterpart which are: array of `AChar`, `AInt`, `AFloat`, and `ASymbol` respectively. In any other case, the value must be returned as an `AObject`.
5. If the lookup found a method, construct a lambda function using the DLR capabilities, which accepts a variable number of arguments, performs the type matching algorithm – which is described in Section 6.5 – and invokes the method returned by type matching. The constructed lambda method is then returned as an A+ function type, represented by the `AFunc` type in A+.NET.

Constructor access can be thought of as a special case of method access. The main difference, in this case, is that the function name parameter is always that of the class.

When the `SelectMember` returns the generated lambda function – either a constructor or another method – the A+ function containing it can be called just like a traditional function in the runtime. Thus, method invocation does not require additional implementation or functionality in the engine. The generated lambda function contains all required information on how to perform the .NET method invocation.

To integrate seamlessly into the syntax of the A+, the existing but hitherto unused $\ominus$ symbol was chosen for the `SelectMember` operation. The format $x \ominus y$ is treated as `SelectMember(x, y)`. Listing 6.1 shows an example use case for the `SelectMember` operation in A+. In line 1, the constructor lookup for the .NET class named `Bar` is demonstrated, which is invoked in line 2. This invocation looks like any other A+ function invocation. Line 3 shows how variables can be accessed with this new operation. Finally, on line 4, the method access and invocations are demonstrated. As this line shows, the call to the $\ominus$ function is enclosed in parentheses to ensure the evaluation order dictated by A+. Also, this way, the language grammar does not require radical changes to give the $\ominus$ function a higher precedence.

Listing 6.1: *Example A+.NET accessor usage*

```
1 constructor ← ⊖ ‘Bar
2 instance ← constuctor{}
3 value ← ‘variable ⊖ instance
4 (‘method ⊖ instance){}
```

6.2 Variable and Property Modification

The modification of variables and properties is similar to the access case, with the addition of a third parameter for the accessor algorithm. This third parameter will serve as the new value for the target variable or property. Additionally, searching for methods are not allowed when performing the member name lookup. This restriction is chosen due to the fact that in the case of .NET, changing an existing method on a class or instance is not permitted.

The following algorithm describes how to modify a variable or property on a given instance or class. It has three input parameters. The first is the name of the variable or property to modify. The second is the class or instance on which the variable/property lookup should be performed. And the third

argument is the new value for the variable/property. The operation is named `SetMember(x, y, z)` and the following steps are performed:

1. Get all variables and properties of the object or instance specified by the second parameter (that is `y`).

2. For each variable and property, check if it has a name which is specified by the first argument (that is `x`).

3. If no match is found, return an error, reporting that there is no such member to modify.

4. As the lookup found a variable or a property, try to cast the new value – specified by the third argument `z` – to the type of the variable/property.

5. If the cast is not possible, return with an error, stating that it is not possible to update the variable with the supplied new value.

6. If the cast was correctly performed, assign the cast value to the selected variable/property.

7. Return with the new value as specified by the third argument. Note that this is the original value of `z`, not after the casting operation.

For the `SetMember` operation, the same $\ominus$ symbol was used, but to invoke the algorithm, it must be used in the $(x \ominus y) \leftarrow z$ format. This can be detected during the parsing of the code, as the accessor function is on the left side of the assignment function. Note that the parentheses are required because of the strict right-to-left evaluation order of the A+ code and the fact that the assignment function has the same precedence as any other function in the language. Listing 6.2 exhibits a use case for the operation. In this example the value of a member named `variable` on a previously accessed class instance named `instance` is updated to have the value `42`.

Listing 6.2: *Example A+.NET instance variable modification*

```
1 (‘variable ⊖ instance) ← 42
```

6.3 Indexers

In .NET, there are numerous objects that provide access to their individual elements. This is usually performed with the help of indexers. To have better integration with .NET types in the new runtime, it is essential to provide an intuitive way to access such elements.

Fortunately, there is also the notation of indexing elements in A+, and therefore, there is no need for introducing new syntactic elements. We can leverage the existing indexing mechanism and only improve its internal mechanisms to add the .NET indexer binding.

As the indexers in .NET are unique properties that can be queried by reflection, the `SelectMember` algorithm can be used to get the value of an item at a given index from an object or instance. Thus, in the indexer case, the `SelectMember` algorithm will search for the indexer properties. In the case of element assignment at a given index, the `SetMember` algorithm will perform the same search as in the element access case.

An example of indexer usage is shown in Listing 6.3. After creating an instance of `ArrayList` in line 1, the code performs an element addition to the list in line 2. (The equivalent code in C# would be: `list.Add(1);`.) Then the usage of the indexers is shown: first, the access of a single element by its index on line 3, continued by the value modification of a given index in line 4.

Listing 6.3: *Example indexer usage for .NET types*

```
1 list ← (⊖ `ArrayList){}
2 (`Add ⊖ list){1}
3 list[0]
4 list[0] ← 2
```

6.4 Type Casting

Type casting is an important operation in the .NET world, as it allows developers to resolve ambiguities. A new ⋄ symbol is introduced into the language, providing the means to perform the .NET type casting functionality in the runtime from the A+ code. This function can also be used for resolving method call ambiguities, which is detailed in Section 6.5.

Listing 6.4 depicts an example of the type casting function, represented by the ⋄ symbol. In the example, the number `1` is initially represented as an A+ number type, but using the new type cast function, it will be changed to a `Boolean` type, which is from the .NET world. The A+.NET symbol `‘Boolean` is used to specify the target type for the cast function. After the cast is performed, the value can be assigned to a boolean variable or property on a given instance if needed.

Listing 6.4: *Example A+.NET type casting for .NET types*

```
1 (‘booleanVariable ⊖ instance) ← ‘Boolean ⋄ 1
```

If the type cast operation cannot be accomplished, an error is reported to the A+.NET runtime. For example, if the developer wants to cast an integer to a string, the error is returned, stating that an invalid cast was attempted.

6.5 Type Matching

As mentioned before in the `SelectMember` operations, .NET methods are looked up by their names. However, just a method name is not always enough to correctly match a method. It is possible that there is more than one method with the same name and the difference is only in the number of arguments or in their types. Thus, to correctly select a method, the types and number of parameters are also required.

The type matching algorithm should be performed after the potential methods based on their names have already been found. First, any method is ignored if it does not have the same number of arguments as the number of arguments supplied for the method invocation. In case there are no methods left to select from, an error is reported during execution, that the number of parameters is incorrect. Second, as the number of arguments is now correct, a type distance vector calculation is performed. The basic building block of the calculation is the type distance notation.

The type distance calculation of non-primitive types (i.e., classes) is based on the inheritance hierarchy of .NET types. If two types are in an inheritance relation, then the type distance of those types is the length of the shortest path between them in the inheritance graph, with the result of `0` if the two types are the same. If the two types are unrelated inheritance-wise, their type distance

is specified as infinite. For example: if there is a class named `Bar` which is a subclass of class `Place` then the type distance between `Bar` and `Place` is one.

For primitive types, inheritance hierarchy is not applicable. However, the C# reference documentation [16, § 11.2] specifies conversion tables, which help to define a pseudo-hierarchy between them and can be used the same way as the real inheritance for non-primitive types. This pseudo-hierarchy is as follows: `Boolean` → `SByte` → `Byte` → `Int16` → `UInt16` → `Int32` → `UInt32` → `Int64` → `UInt64` → `Single` → `Double` → `Decimal` → `AType` → `Object`, the `Char` type connects into the `UInt16` type, the string values are starting with `String` → `Enum` and connecting into the `AType`. Thus, for example, if there is an input argument with the type `Byte` and the method requires an `Int32`, the distance between the two types is `3`. The A+ types are using this pseudo-hierarchy to allow interoperability with the .NET methods. A+ symbols are using the chain starting from the `String` type, integers either from `Int32` or `Double`, and characters use the `Char` type as a starting point.

Based on this type of distance information, it is now possible to define the type distance vector. The type distance vector is a vector of `N` elements where `N` is the number of input parameters for the current method invocation and for each element, the type distance is calculated between the input parameter and the parameter of the potential method. After the distance calculations, the best method is chosen. A method is considered better than the other if each element of its type distance vector is smaller than or equal to the corresponding element in the other method's vector, but at least one element is strictly smaller than its corresponding element. This selection method is based on the better function member selection described in the C# language specification [16, § 12.6.4.3]. Note, however, that there may be incomparable elements in this relation. Only if there is a single method that is better than all other alternatives does the runtime select that method for invocation. Otherwise, the runtime reports an error because of the ambiguity. Such ambiguities can be resolved with the newly introduced type casting operation.

Listing 6.5 depicts a C# method ambiguity. For this use case the `(`Foo ⊖ `class){(⊖ `Bar){}, (⊖ `Car){}}` invocation would not be successful. The calculated type vector distance for the invocation and the first `Foo` method would be – based on the type hierarchy – the vector `[1, 0]`, and for the second `Foo` method it would be `[0, 1]`. Thus, based on the better method selection rules, it is not possible to select one method, and the ambiguity error is reported. However – as stated before – it is possible to resolve the ambiguity if, e.g., the second parameter for the method invocation is cast to `Vehicle`. Then the

Listing 6.5: *Example for C# method ambiguity*

```
1 class Place {}
2 class Bar : Place {}
3 class Vehicle {}
4 class Car : Vehicle {}
5
6 void Foo(Place arg1, Car arg2) {}
7 void Foo(Bar arg1, Vehicle arg2) {}
```

type distance vector for the second case would result in a `[0, 0]` vector, and a perfect match would be found.

7 Conclusions

In this part of the thesis, topics related to the A+ language were presented. In order to be a bit more familiar with the language itself, a small language introduction discusses the various unique aspects of A+.

The first new topic was the implementation of the A+.NET environment. The main challenge of this implementation was twofold: there was no existing formal grammar, and the original implementation was not always in line with the operations described in the A+ Language Reference document. With our work, a formalized grammar was created for A+ and we adopted an approach to follow the original implementation's deviations from the reference document in some cases. With the newly formalized grammar, the implementation of the A+.NET environment leveraged the ANTLR lexer-parser generator to help build ASTs from the input A+ code. For the code execution, the Dynamic Language Runtime component was used. This component gave the Expression Trees with the transformation ASTs to executable code (MSIL) was done. Furthermore, with the usage of DLR and its hosting capabilities, the integration of A+ code into a C# application was demonstrated.

In the second topic, the original A+ and the A+.NET implementation were compared in terms of script execution speed and maintainability metrics. Here it is important to note that even though the A+.NET implementation is slower in runtime, the ability to replace A+ script parts with .NET implementations

can provide huge speedup. Furthermore, the original implementation has almost twenty years of extra time to add various optimizations. In terms of source mode metrics, the two systems' comparable parts were found, and for these, a maintainability comparison was made. Based on this comparison, the presented data indicate that the re-implemented version of the runtime is less complex and thus – presumably – more maintainable.

In the final chapter of this part, the improvement of the A+ language is presented. A set of required operations are presented in order to access object-oriented components in a language. These are the member access, member set, indexing, and type cast operations. For these operations, extra care was taken to make sure that the new syntax and semantics do not collide with existing elements of the language. The right-to-left evaluation order was kept, and the context handling of the language was not changed. Furthermore, a vector based type matching algorithm is also presented that is mainly used for method invocations in order to correctly report any type errors or method ambiguities. In the future, the A+ language could be extended with full object-oriented notation. This would allow a developer to create classes/objects in A+ itself, further extending the language's lifetime and expending the possible usages of this environment.

Part II

Primitive Enthusiasm Metrics

8 Introduction

During software development, the structure of its source code will change and could even degrade. After a while, identifying the base design will be more difficult. In addition, the previously easily understandable code parts could be harder to comprehend. Modifications and refactorings that help to improve various aspects – like complexity or maintainability – of the code base without changing the external functionality should be applied frequently to preserve code quality.

There are already several known indicators that can help to notice if code refactoring should be done. Fowler and Beck [20] list 22 code bad smells that can be used as indicators for such cases. Usually, code smells are not coding errors but highlight design issues that could cause problems in the future. This is why it is not surprising that there are diverse discussions on code smells among the researchers, not just on how to detect them but also on the possible impact on maintenance costs. The original 22 code smells were extended over the years, introducing new smell types and the means to detect them.

Not all code smells are treated equally, some of them get less attention than other code smells. One such smell is the Primitive Obsession, which occurs if the primitive data types are overused. We decided to study this code smell and defined the Primitive Enthusiasm metric and its derivatives. This is done to grasp a part of the Primitive Obsession code smell and show where it could

occur in a software.

Finding various bugs in applications could take from several minutes to hours or even more. As a project development continues, the cost of finding and fixing a bug will increase [10]. Therefore, researchers are also working on creating and improving various automated tools that can help a developer to quickly find a bug or even just provide hints on where the bug could be in the system. In pursuit of automated tools to provide bug predictions there are already several papers present [40, 68]. Some of these tools use various source-level metrics to train and build models which will be used for bug predictions. In relation to this, we investigated how the new Primitive Enthusiasm metrics can be leveraged for bug prediction.

Structure of this part: Chapter 9 briefly describes the Primitive Obsession code smell and bug prediction background. The new set of metrics is presented in Chapter 10. A quick look at small projects with the new metrics is covered in Chapter 11. In Chapter 12 the bug prediction capabilities are presented. These topics are presented in papers [26], [57], and [22]. Results and final thoughts are summarized in Chapter 13.

9 Background

9.1 Definition of Primitive Obsession

Usually, data types can be categorized into two groups: primitive and complex. Primitive types are simple data types provided by a language, e.g., *char*, *integer*, etc. Complex types are constructed from these primitive types or by using other complex types and are usually named *class* or *struct*. Additionally, such complex types can encapsulate operations on their data, providing semantic knowledge on how the data can or should be used. For example, it would be convenient to place a *string* and two *integer*s that represent a task name, a start, and end time into a class instead of using them separately. Doing such data coupling and providing a name for the class will allow the developers to understand the connections between the data. However, neglecting such minor but useful code compositions is also common. Similarly, adding additional method parameters to an already long parameter list seems to be a quick solution to achieve a programming goal. However, in the long run, this will make the relation between data harder to understand and decrease the source code's maintainability. Excessive usage of primitive data types instead of creating small objects is the core of Primitive Obsession. Mäntylä et al. classify Primitive Obsession as a type of *bloater* smell [46], in addition, it is a symptom of the existence of overgrown, chaotic code parts. A simple example of Primitive

Obsession is depicted in Listing 9.1. The `NORMAL` and `URGENT` class constants on Lines 2 and 3, can be considered type codes. Such type codes are usually integers or strings that have a name and are used to simulate types. In this case these are used to indicate task priorities. Type codes are widely used in various projects and can be considered a version of Primitive Obsession. They break the object-orient paradigm and can cause hidden dependencies [71]. There are various ways to remove such type codes from a project, for example, with a *Strategy* or *State* pattern [27].

Listing 9.1: *Sample code containing Primitive Obsessions*

```
1  class Task {
2    static const int NORMAL = 1;
3    static const int URGENT = 2;
4
5    public void work(
6        int from, int to, int breakCount, String worker)
7    {
8      if(worker == null || worker.length() == 0) {
9      /*...*/
10     }
11   }
12 }
```

Starting from Line 5, the `work` method is presented. This method has three integer parameters and a single string argument. As all of these arguments are primitive types, the method can be considered to have the Primitive Obsession code smell. Even more so if these arguments are repetitively used throughout the project's codebase. Please note that in Java strings are classes thus, they are not really primitive types, but logically it makes sense to categorize it as a kind of primitive type. Line 8 is also a good target for refactoring because the checks present in this line should be encapsulated into a class. Especially if these checks are used in multiple places. For more examples that contain Primitive Obsession, Steven A. Lowe's GitHub project [45] is a good start. This project provides a set of steps on how to clean up this kind of code smell.

9.2 Challenges using Primitive Obsession

Previously, the Primitive Obsession was not in the main focus of researchers. A team of researchers conducted a literature overview on code smells [72]. During

the review, they gathered 319 papers from 2000 to 2009. From these collected papers, they examined 39 in more detail. One of the investigation point was to determine which code smells attached the most attention from researchers. The most discussed topic was the Duplicated Code smell, 21 papers from 39 elaborated on this. At the same time, other code bad smells received very little attention. Among the unpopular smells was the Primitive Obsession, which was only present in 5 papers. However, these papers also discussed Fowler's other code bad smells.

Another literature overview [29] by Gupta et al. collected 60 papers from 1999 to 2016 and concluded that four code bad smells did not have any detection method, including Primitive Obsession. An investigation of five tools that could detect code smells reported that none of them could find the Primitive Obsession code smells.

In a thesis, Roperia introduced JSmell for Java applications that could detect Primitive Obsession smells [61]. The detection technique for this smell was based on the number of primitive data types declared in a class. JSmell reports Primitive Obsession when the number of primitively typed data is above the average of a project, and the class was not instantiated. As the tool was not accessible, a comparison was not made between JSmell's approach and the Primitive Enthusiasm metrics solution.

9.3 Bug Prediction and Datasets

Using a bug predictor tool could help developers quickly triage an issue in a given system cost-effectively. In one research [30], the authors used source code metrics in conjunction with machine learning methods to provide fault predictions. They observed that on the Mozilla suite, there are source code metrics that can help to predict fault-prone classes. A different study [50] concluded that defect predictors that were learned from source code metrics were useful and easy to use. In addition, they are widely used. Another conclusion of the paper is that the goal of the learning should be used to select a preferred machine learning algorithm. Other papers elaborate on the usefulness of code smells for maintenance indicators [52, 70]. However, these cannot be considered as an all-in-one solution for defect predictions, but they still give insights into important maintainability aspects for a project, even more if they are combined.

In order to evaluate a method that provides bug prediction capabilities, a dataset is required. Ideally, this dataset should provide additional data,

such as various source code metrics and the correct classification of classes or methods that have bugs in them. In addition, it is favourable to have real-world software in a dataset, as ultimately, a bug prediction tool should be used on real applications, not on small benchmarks. In order to achieve this, a unified Java dataset was used. This bug dataset was introduced by Ferenc et al. [19] This dataset contains a collection of five publicly available bug datasets in a unified format containing class and/or file level code metrics, including information on defects. These are the following:

- PROMISE dataset [41]
- Eclipse bug dataset [73]
- Bug Prediction dataset [14]
- Bugcatchers dataset [31]
- GitHub bug dataset [67]

10 Defining Primitive Enthusiasm

As the previous section highlights, giving Primitive Obsession an exact, quantifiable definition is challenging. It has many aspects, and every program is unique with its own traits and criteria. Developers also abstract and implement the components differently. Therefore, it is challenging to find an absolute threshold for how many times the primitive types can be used before something can be considered Primitive Obsession, and to apply such a threshold to every project is a similarly demanding endeavour.

Since defining Primitive Obsession is hardly possible with a single formula, the deconstruction of the bad smell is a logical approach.

A part of this deconstruction is the new Primitive Enthusiasm metric. The idea is to quantify the primitive typed arguments in a function so that the methods can be compared to other methods in the system. To achieve this, the metric does not employ a globally – as in outside of a selected project's scope – defined value but tries to capture each system's uniqueness by comparing the results to other methods of the same system.

The base of the metrics is the Formula 10.1 which describes how the primitive-typed parameters are collected for a given M_i method.

$$Primitives(M_i) := \langle P_{M_i,j} | 1 \leq j \leq |P_{M_i}| \wedge P_{M_i,j} \in PrimitiveTypes \rangle \quad (10.1)$$

In this formula, the definitions of the parameters are the following:

- ***PrimitiveTypes*** is the set of types that are handled as primitive ones. For Java this contains the following types: `boolean`, `byte`, `short`, `int`, `long`, `char`, `float`, `double`, and `String`.
- $\boldsymbol{N}$ represents the number of methods in the current class.
- $\boldsymbol{M_i}$ denotes the $\boldsymbol{i}$th method of the current class.
- $\boldsymbol{M_c}$ denotes the current method under investigation in the current class.
- $\boldsymbol{P_{M_i}}$ denotes the list of types used for parameters in the $\boldsymbol{M_i}$ method.
- $\boldsymbol{P_{M_i,j}}$ defines the type of the $\boldsymbol{j}$th parameter in the $\boldsymbol{M_i}$ method.

Albeit, `String` is a class in Java and not a primitive type, the way it is usually used is just like a primitive type. This is why treating it as a primitive in the current metric is recommended. Using this *Primitives* function, three metrics were constructed. These metrics are described in the following sections.

10.1 Local Primitive Enthusiasm

Formula 10.2 depicts the calculation of the new Local Primitive Enthusiasm (LPE) metric.

$$LPE(M_c) := \frac{\sum_{i=1}^{N} |Primitives(M_i)|}{\sum_{i=1}^{N} \mid P_{M_i} \mid} < \frac{|Primitives(M_c)|}{\mid P_{M_c} \mid} \tag{10.2}$$

The left-hand side of the inequality calculates the percentage of how many parameters of the current class are of primitive types. While the right-hand side denotes how many parameters of the M_c method are of primitive types. This can be considered as the method's primitiveness ratio. If the right-hand side is greater than the left-hand side, that is, there are more primitive types in the M_c method's parameter list than in the given class on average, then the method is considered to be an LPE true method.

The Formula 10.2 is calculated for each method in a given class and compared to the ratio calculated for the same class. This is why the formula is treated as a Local Primitive Enthusiasm. Originally, this was the initial version of the Primitive Enthusiasm metric.

10.2 Global Primitive Enthusiasm

Global Primitive Enthusiasm calculation is shown in Formula 10.3. In this formula, the G is the list of methods in the analysed system, and G_i is the ith method in this list. The main difference from GPE is that the left-hand side now describes the average number of primitive-typed arguments in the whole system and this is compared to the right-hand side, which is the current method's primitiveness ratio.

$$GPE(M_c) := \frac{\sum_{i=1}^{|G|} |Primitives(G_i)|}{\sum_{i=1}^{|G|} \mid P_{G_i} \mid} < \frac{|Primitives(M_c)|}{\mid P_{M_c} \mid} \tag{10.3}$$

The idea behind GPE was that this compares the current method to the other methods in the whole project, which in turn can highlight methods that are above the system's average.

10.3 Hot Primitive Enthusiasm

By combining the LPE and GPE formulae Formula 10.4 was constructed. The purpose of this combination is to direct the developer's attention to more suspicious code parts.

$$HPE(M_c) := LPE(M_c) \wedge GPE(M_c) \tag{10.4}$$

If a method is HPE true, then it indicates that in terms of primitive method arguments it is an outstanding method both in the scope of the current class and in the whole system.

10.4 Primitive Enthusiasm and Wrapper Classes

The *PrimitiveTypes* set in the original formulate contained only the primitive types found in Java, as this was the first implementation of the metric calculation. However, in Java, there are wrapper classes for the primitives defined by the language standard, namely: `Boolean`, `Byte`, `Short`, `Integer`, `Long`, `Character`, `Float`, `Double`. Although these types are classes, they can be considered as they represent primitive data types. Thus, we constructed the

wrapper class versions of the Primitive Enthusiasm Metrics by including these types in the *PrimitiveTypes* set used for the metric calculation. In order to differentiate these variants in the thesis, a `W+` subscript is added if the wrapper classes are included and a `W-` subscript is added if the wrapper classes are not included. So the Local Primitive Enthusiasm with Wrapper classes is $\mathrm{LPE}_{\mathrm{W+}}$. Similarly the $\mathrm{GPE}_{\mathrm{W+}}$ and $\mathrm{HPE}_{\mathrm{W+}}$ variants are also constructed.

11

Metric Calculation Evaluation

To implement and evaluate the proposed formulas and their variants, the Open Static Analyzer [6] was used, which is an open-source, multi-language, static code analyzer framework developed at the Department of Software Engineering, University of Szeged. The prototype implementation processes Java source code, however, with a minor modification, the metrics could be applied to other object-oriented languages as well.

As there is no benchmark for Primitive Obsession detection, manual validation was performed to see how the Primitive Enthusiasm metric and variants are present in systems in three open-source Java projects. These projects are often used in research papers working with various metrics:

- Joda-Time version 2.9.9 [4] a date and time replacement library,
- Apache Log4j [3] a commonly used logging library,
- Apache Commons Math version 3.6.1 [2], a library of lightweight, self-contained mathematics and statistics components.

11.1 Eliminated Methods

The formulae introduced in Section 10 for Primitive Enthusiasm calculation – by default – do not take into account that some methods have only one parameter or none at all. Also, certain types of methods should be eliminated from the calculation of the metrics. The implementation of PE metrics skipped the investigation of constructors, the methods with an empty parameter list, and methods that could be considered a class member setter method.

Using primitives for constructor arguments should not be considered in PE calculations, as without these methods, it would be challenging to encapsulate primitive types in a class. Similarly, the removal of setter methods is in the same category, as the developer should allow the object's internal properties to be changed in some cases. In Java, setter methods are usually following the `setX` format where `X` is the name of internal value they want to change. This variant of method eliminations was called Skipping Set Methods (SSM).

Upon further work, the restriction was enhanced to skip the processing of all methods with less than two parameters. The idea behind this approach is that if a method only has a single primitive parameter, it usually either sets it as an internal value or does some kind of calculation with it, probably mostly short, simple getter-setters. In addition, a single parameter for a method can't be considered an overuse of primitive types. This variant was named Skip Ones (SO).

A comparison of these two method skip variants is presented in the following section.

11.2 Results

In order to better understand the target projects, a few metrics were calculated. Table 11.1 summarizes these metrics.

The smallest project is log4j, with "only" 1561 methods in 189 classes and the biggest one is the Apache common-maths library with 8808 methods in a bit more than a thousand classes.

11.2.1 Exclusion Strategy

As it was stated in Section 11.1, two strategies were evaluated for eliminating methods: to skip only setter methods (*Skip Setter Methods - SSM*) or skip every method with just one parameter (*Skip Ones - SO*). These approaches

Table 11.1: *Properties of the examined projects: thousand lines of code (KLOC), number of classes (NC), number of methods (NM), longest parameter list (LPL)*

Project	KLOC	NC	NM	LPL
log4j	16	189	1561	9
joda-time	28	249	4265	10
commons-math	100	1033	8808	14

were proposed and implemented by co-author Edit Pengő. For the sake of completeness, Table 11.2 sums up the details of the two strategies for the examined three systems.

Table 11.2: *Comparison of the two elimination strategies: number of not eliminated methods (NNM), average parameter list length of not eliminated methods (AVG)*

	SSM		SO	
Project	NNM	AVG	NNM	AVG
log4j	1371	1.33	429	3.01
joda-time	3787	0.98	893	2.55
commons-math	7229	1.12	1953	2.93

The numbers show that by skipping every method with just one parameter, only a subset of the methods was processed for the calculation. During the manual validation, we observed that the SSM strategy causes noise in the results. It is not surprising as it is hard to interpret Primitive Obsession on methods with only one parameter, as it is pretty hard to argue that a single primitive parameter of a method is too much.

11.2.2 Effect of Wrapper Classes

In order to see all aspects of the Primitive Enthusiasm metrics, it is important to see how the previously introduced variants are behaving in relation to each other. The PE variant, which includes the wrapper types was already

introduced in Section 10.4 as having a `W+` as a subscript, similarly to the `W-` annotation.

To see how the inclusion or exclusion of the wrapping types changes the number of detected methods, the number of methods with at least one (AL1) and at least three (AL3) primitive typed parameters were counted. During this calculation, only the constructor methods were eliminated, the previously introduced skip strategies were not applied The results for this are summarized in Table 11.3, and the difference is much less than expected. This table shows that the extension of the *PrimitiveTypes* set with boxing types has very little consequence on the set of methods that use primitive types in their parameter lists, as the difference is only a few methods or none at all. These results also mean that the wrapper types are less frequently used in these Java projects and probably in other projects. With this we can conclude that there is no demerit including wrapper types in the *PrimitiveTypes* set.

Table 11.3: *The impact of wrapper classes by the number of methods: at least one primitive parameter in the parameter list (AL1), at least three primitive parameters in the parameter list (AL3), exclude wrapper classes from the PrimitiveTypes set (W-), include wrapper classes in the PrimitiveTypes set (W+)*

	AL1		AL3	
Project	W-	W+	W-	W+
log4j	577	577	35	35
joda-time	1580	1583	95	96
commons-math	2758	2759	556	556

11.2.3 Reports on Primitive Enthusiasm Metrics

The Primitive Enthusiasm metrics suggest such methods for refactoring that use an unusually lot of primitive parameters. However, for large projects, the list of reported methods can be long, making the review a demanding task. Therefore, a class-level aggregation of the results is proposed and used during the evaluation. This way, the classes can be ordered by the number of their reported methods.

Table 11.4: *Comparison of Primitive Enthusiasm reports on method level*

	Number of reported methods					
	LPE_{W-}	LPE_{W+}	GPE_{W-}	GPE_{W+}	HPE_{W-}	HPE_{W+}
log4j	165	165	217	217	153	153
joda-time	301	301	429	431	230	231
commons-math	698	698	1192	1192	553	553

Table 11.5: *Comparison of Primitive Enthusiasm reports on class level*

	Number of reported classes					
	LPE_{W-}	LPE_{W+}	GPE_{W-}	GPE_{W+}	HPE_{W-}	HPE_{W+}
log4j	29	29	74	74	29	29
joda-time	92	92	104	105	65	66
commons-math	213	213	390	390	145	145

Table 11.4 shows the number of reported methods of the three studied Java systems, while Table 11.5 depicts the number of classes with at least one reported method. Both of these tables are calculated with the *Skip Ones* (SO) strategy.

These warnings were processed manually to validate the results of the various Primitive Enthusiasm metrics. After randomly sampling the reported methods, they were checked in the source code. Additionally, the class level aggregations were processed with great attention to the classes with the most and least reported methods. As both the evaluation and the judgment of Primitive Obsession itself are subjective, deciding if a warning is true positive or not is hard to tell, therefore, the significance of the warning was investigated.

It is clear from the results of Table 11.4 and Table 11.5 that the Primitive Enthusiasm metrics report quite a few methods and classes. However, the number of reported methods is less than 8% of the total number of methods on average. The other important aspect of these results is that there is almost zero difference between the metrics that include or do not include the wrapper classes.

Considering the results of Section 11.2.2 it is not surprising, nevertheless, it is recommended to include these types in the *PrimitiveTypes* set as their usage might depend on the nature of the project or the habits of the programmer.

GPE_{W-} and GPE_{W+} express how different the composition of the parameter list of a method is from the global average, whilst LPE_{W-} and LPE_{W+} express only a class-level distinction. The combined metrics HPE_{W-} and HPE_{W+} perform well as they are able to further fine-tune the results given by the LPE and GPE variants. In most cases, the reduction in the number of reported methods and classes was more than 25% percent. Based on this information, it is recommended that first, the HPE_{W-} or the HPE_{W+} reports should be investigated by the developer as they report both class-localized and application-global suspicions.

12
Bug Prediction Capabilities

The previous sections presented the Primitive Enthusiasm metrics and a simple evaluation of these metrics. By itself, it can be already helpful, but going forward, it is worth investigating if these new metrics could add benefits in bug prediction.

To investigate the bug prediction capabilities of the PE metrics, an already existing Java-based bug dataset was used [19]. From this dataset, the PROMISE, GitHub, and Bug Prediction datasets were selected and used for prediction calculations. These selected datasets contain class-level metrics, and in total there are 66 systems in them. Among these 66 systems, there are even multiple version for a few projects. For each system, 63 commonly used source code metrics are already calculated, and classes with bugs are also marked. This dataset was extended with the previously mentioned PE metrics and calculated using the same Open Static Analyzer (OSA) [6] tool where the new metric calculations were implemented.

12.1 Calculating the Metrics

The selected bug dataset already contains class-based metrics, and for each of the classes in this dataset, the PE metrics were calculated. However, the PE metrics are inherently method-based. In order to resolve this minor incompati-

bility, the PE metrics were aggregated by class. That is, for a given class the number of LPE, GPE, and HPE true functions are counted, and these values are used as class-level metrics. This aggregation inherently connects the PE metrics with the number of methods in a class. If there are more methods in a class, the class has a higher probability of having more PE true methods. After the aggregation, the class-level PE metric value could be between 0 and the number of methods in that class. In the following sections, the PE metrics will refer to this kind of aggregated value if not mentioned otherwise.

12.1.1 Information on Selected Systems

In order to get an initial insight into the selected systems, an initial metric calculation and aggregation were done. As in the used bug dataset, there are multiple versions present for some projects, the author selected the latest version of each project for further evaluation. Based on this criteria, 33 systems were selected from the original dataset. We first checked how many classes there are and how many bugs were reported for these systems. In addition, the aggregated PE metric values were calculated for each system. These values can be seen in Table 12.1.

The first column, "Class" describes the number of classes in the given system. The second "Bugs" column represents the number of classes with at least a single reported bug, and the third "Bugs/Class" column shows how many of the system classes have a bug in them. This was calculated by dividing the number of classes that have bugs in them with the total number of classes in the system. The LPE, GPE, and HPE columns show how many classes have at least a single LPE, GPE, or HPE method. There are various-sized projects ranging from a small 8 class system to a more than 5500 class system. What is interesting to see is that there are usually more GPE true classes than LPE true ones. For example, in the case of the Elasticsearch system, there are 503 more GPE true classes than LPE ones. In addition, the HPE value is usually lower than the LPE value. This indicates that the combination of the LPE and GPE metrics can help focus the developers' attention.

There are two tiny projects on this list: the Ckjm 1.8 and Forrest 0.8. As there are already a small number of classes present, the PE metric values are also small for these two.

Table 12.1: *Class count, Bug count, and PE metric count information on selected systems*

	Class	Bugs	Bugs/Class %	LPE	GPE	HPE
Equinox	319	126	39.5	78	114	74
Lucene	670	63	9.4	115	175	85
Myly	1405	209	14.9	240	421	206
Eclipse PDE UI	1491	208	14.0	357	510	321
Android U. I. L.	73	20	27.4	7	23	7
ANTLR	479	21	4.4	60	119	56
Broadleaf	1593	292	18.3	173	271	164
Eclipse p. for Ceylon	1610	68	4.2	187	284	176
Elasticsearch	5908	678	11.5	560	1063	480
Hazelcast	3412	377	11.0	204	391	185
JUnit	731	35	4.8	17	29	12
MapDB	331	40	12.1	64	89	62
mcMMO	301	57	18.9	41	35	26
MCT	1887	9	0.5	146	219	106
Neo4j	5899	58	1.0	582	1025	498
Netty	1143	271	23.7	128	191	122
OrientDB	1847	280	15.2	288	402	253
Oryx	533	74	13.9	36	97	32
Titan	1468	96	6.5	135	263	126
Ant 1.7	681	165	24.2	100	190	84
Camel 1.6	795	170	21.4	88	173	87
Ckjm 1.8	8	5	62.5	1	1	1
Forrest 0.8	31	2	6.5	7	14	7
Ivy 2.0	294	37	12.6	81	97	68
JEdit 4.3	439	11	2.5	150	162	119
Log4J 1.2	191	175	91.6	30	71	29
Lucene 2.4	320	193	60.3	65	92	46
Pbeans 2	37	8	21.6	13	18	12
Poi 3.0	414	276	66.7	62	274	50
Synapse 1.2	228	84	36.8	47	83	46
Velocity 1.6	188	66	35.1	51	76	46
Xalan 2.7	848	837	98.7	191	208	147
Xerces 2.0	342	303	88.6	78	102	64

12.2 Correlation Between Metrics

As various metrics are already present in the bug dataset, it is worthwhile to consider how the PE metrics behave compared to other metrics. With this investigation, we can see if there are any metrics that have strong connections with the new PE metrics. For this, a correlation matrix was calculated using Pearson's correlation for each metric, merging all of the selected systems into a single input dataset. This matrix is shown in Table 12.2. A colour map was applied to help show the correlation values between the metrics. The greater the correlation between the metrics, the darker the cell colour.

Based on the correlation matrix, the LPE, GPE, and HPE metrics strongly correlate. The LPE and GPE metrics have a 0.86 correlation value which is caused by the fact that there is only a minor difference between the calculation of these two metrics. In addition, the fact that the HPE has a high correlation with both LPE and GPE is not a surprise as the HPE metric by definition is calculated from these two metrics.

The highest correlation between the PE metrics and traditional metrics can be found for the NLM and LPE metrics that have the value of 0.58. All other correlation values are less than this value. By investigating some of the higher correlation values, we could have an insight into the connection between the metrics.

The PE metrics have greater correlations with metrics that are calculated from the number of methods. These are the relevant metrics:

- NLM: Number of Local Methods.
- TNLM: Total Number of Local Methods.
- NLPM: Number of Local Public Methods.
- TNLPM: Total Number of Local Public Methods.
- RFC: Response set For Class.
- WMC: Weighted Methods per Class.

Their highest correlation values for PE metrics are 0.58, 0.54, 0.52, 0.48, 0.53, and 0.51 in the same order. These correlations are understandable, as the PE metrics have connections to the number of methods since the aggregation was done by class (as was described in Section 12.1). The chance of having PE true methods is higher if there are more methods. In addition, the PE metrics

Table 12.2: *Correlation between PE and other metrics*

	LPE	GPE	HPE		LPE	GPE	HPE
LPE	1.00	0.86	0.96	NM	0.28	0.27	0.26
GPE	0.86	1.00	0.91	NOA	0.03	0.01	0.03
HPE	0.96	0.91	1.00	NOC	0.04	0.04	0.03
AD	0.10	0.09	0.08	NOD	0.04	0.03	0.03
CBO	0.31	0.24	0.27	NOI	0.39	0.33	0.36
CBOI	0.19	0.20	0.19	NOP	0.06	0.04	0.05
CC	0.00	-0.01	0.00	NOS	0.40	0.35	0.38
CCL	0.11	0.09	0.10	NPA	0.02	0.02	0.02
CCO	0.09	0.08	0.09	NPM	0.26	0.27	0.25
CD	0.08	0.08	0.07	NS	0.11	0.12	0.11
CI	0.08	0.07	0.08	PDA	0.37	0.34	0.36
CLC	-0.01	-0.02	-0.01	PUA	0.36	0.41	0.36
CLLC	0.00	-0.01	0.00	RFC	0.53	0.49	0.50
CLOC	0.41	0.35	0.39	TCD	0.07	0.08	0.07
DIT	0.01	0.00	0.01	TCLOC	0.41	0.35	0.39
DLOC	0.38	0.33	0.36	TLLOC	0.49	0.44	0.47
LCOM5	0.26	0.28	0.24	TLOC	0.51	0.45	0.49
LDC	0.15	0.13	0.15	TNA	0.11	0.10	0.11
LLDC	0.16	0.14	0.16	TNG	0.17	0.15	0.17
LLOC	0.49	0.44	0.46	TNLA	0.13	0.12	0.13
LOC	0.51	0.45	0.48	TNLG	0.35	0.33	0.34
NA	0.09	0.07	0.08	TNLM	0.54	0.52	0.51
NG	0.18	0.17	0.18	TNLPA	0.02	0.02	0.02
NII	0.26	0.28	0.26	TNLPM	0.48	0.49	0.46
NL	0.28	0.25	0.26	TNLS	0.20	0.22	0.21
NLA	0.10	0.09	0.10	TNM	0.23	0.21	0.22
NLE	0.30	0.27	0.28	TNOS	0.42	0.36	0.39
NLG	0.36	0.34	0.35	TNPA	0.03	0.02	0.03
NLM	0.58	0.57	0.55	TNPM	0.23	0.22	0.22
NLPA	0.01	0.01	0.01	TNS	0.13	0.12	0.13
NLPM	0.50	0.52	0.49	WMC	0.51	0.45	0.49
NLS	0.20	0.22	0.20	bugs	0.15	0.14	0.14

are calculated from method argument lists. Thus, if there are more methods, there are usually more arguments. Another interesting insight can be seen if the correlation between these common metrics is investigated. Table 12.3 elaborates on the correlation between these commonly used metrics.

Table 12.3: *Correlation between a selected set of method count related metrics*

	NLM	NLPM	RFC	TNLM	TNLPM	WMC
NLM	1	0.92	0.84	0.91	0.86	0.75
NLPM	0.92	1	0.75	0.82	0.89	0.55
RFC	0.84	0.75	1	0.79	0.72	0.68
TNLM	0.91	0.82	0.79	1	0.94	0.69
TNLPM	0.86	0.89	0.72	0.94	1	0.53
WMC	0.75	0.55	0.68	0.69	0.53	1

A strong correlation can be seen in most cases between these method-based metrics, and these values are not a surprise as the metrics are also calculated mainly or partially from the number of methods in a class. Excluding the WMC metric, the correlation values were above 0.70 in every case, and in some instances, it is even above 0.90. The NLM, TNLM, NLPM, and TNLPM correlation results are not surprising, each of these captures the number of methods in a class, but with a little variation. As shown in Table 12.2 and Table 12.3, the PE metrics have less correlation than these method-related metrics have between each other. This shows that the PE metrics can give an extra dimension – aside from the already existing method-related metrics – as they are not tightly coupled even if there is a connection between PE and traditional method-based metrics.

The other set of interesting correlations with PE metrics are the ones related to lines of code:

- LOC: Lines of Code.
- LLOC: Logical Lines of Code.
- CLOC: Comment Lines of Code.
- DLOC: Documentation Lines of Code.
- TLOC: Total Lines of Code.

- TLLOC: Total Logical Lines of Code.
- TCLOC: Total Comment Lines of Code.
- TNOS: Total Number of Statements.
- NOS: Number of Statements.

The connection between these and the PE metrics can be attributed to the fact that if there are more lines of code, then there are usually more methods in an application. With the 0.51 correlation value, the LPE, LOC, and TLOC metrics are the most connected. In every other case, the correlation was less. Just like previously, it is also worth checking the correlations between these common metrics. This matrix is depicted in Table 12.4.

Table 12.4: *Correlation between a selected set of line count related metrics*

	CLOC	DLOC	LLOC	LOC	NOS	TCLOC	TLLOC	TLOC	TNOS
CLOC	1.00	0.95	0.53	0.69	0.46	0.99	0.53	0.68	0.48
DLOC	0.95	1.00	0.39	0.56	0.32	0.94	0.40	0.56	0.34
LLOC	0.53	0.39	1.00	0.97	0.97	0.52	0.97	0.95	0.96
LOC	0.69	0.56	0.97	1.00	0.93	0.68	0.95	0.97	0.92
NOS	0.46	0.32	0.97	0.93	1.00	0.46	0.93	0.90	0.98
TCLOC	0.99	0.94	0.52	0.68	0.46	1.00	0.54	0.69	0.48
TLLOC	0.53	0.40	0.97	0.95	0.93	0.54	1.00	0.98	0.97
TLOC	0.68	0.56	0.95	0.97	0.90	0.69	0.98	1.00	0.93
TNOS	0.48	0.34	0.96	0.92	0.98	0.48	0.97	0.93	1.00

As shown in Table 12.4, most of these metrics have high correlation values with each other. Just like before, these values are understandable since all of them are size-related metrics and calculated from various types of code lines.

Based on these experiments, it can be concluded that although the Primitive Enthusiasm metrics have some kind of connection between already existing metrics, they can still give extra information for a developer.

12.3 Cross-project Bug Prediction

A cross-project validation was performed to evaluate the PE metrics' bug prediction capabilities. The cross-project validation was chosen as a method to see how the newly added metric changes the bug prediction, and classification.

II. 12. Bug Prediction Capabilities

For the prediction, the Weka [21] tool was used with the default Random Forest classificator. The classification label was the bug presence for a given class. That is, there is at least a single bug in the target class. Using selected each project, a model was trained, and every other project was used as evaluation test data. These training and evaluation steps were done for the original bug dataset metrics and for the newly extended dataset containing the Primitive Enthusiasm metric aggregated to a class level. This gave us two sets of classification results. For comparing the two result sets Weka's F-measure value was used. In Weka, the weighted average F-measure is calculated with the following formula:

$$\frac{Fmeasure(a) * InstanceCount(a) + Fmeasure(b) * InstanceCount(b)}{InstanceCount(a) + InstanceCount(b)}$$

Where a and b are the two classes for classification, $Fmeasure(x)$ represents the F-measure value for class x, and $InstanceCount(x)$ gives the number of instances in the x class. Using this, the difference between F-measure values was calculated for each train-test system pair. This difference shows how the classification results changed after adding the new metrics to the dataset. These differences can be examined in Table 12.5.

In this table, the first and second column describes the project on which the training was done. The IDs of the projects are used as column labels to reduce the size of the table a bit. The columns after the first two are the project IDs on which the trained model was evaluated.

To highlight interesting parts, the values in the table were colour-coded. To indicate improvement, cells that have a value of 0.05 or higher are coloured green. In return, values less or equal to -0.05 are coloured red to indicate a decrease in weighted F-measure values. Due to rounding, it is possible that a value that is equal to one of these thresholds is not coloured. Aside from the matrix main diagonal, some other cells do not have any value. In these cases, the classificator was unable to calculate the weighted F-measure. This usually meant that only bugged or non-bugged classes were reported by the classificator resulting in an invalid weighted F-measure value.

The highest decrease in F-measure value was in the case of the Xerces 2.0 system. Based on Table 12.1, most of the classes in the system have a bugged class. Due to this, an over-fitting can be observed by the classificator marking more classes as bugged. A similar over-fitting can be observed in the case of the Xalan 2.7 project. However, the F-measure change cannot be observed in Table 12.5, because in most cases, even using the original dataset

Table 12.5: *Weighted fmeasure changes in case of cross-project validation*

ID	Name	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
1	Equinox		-0.00	-0.02	-0.03	0.06	0.01	0.00	-0.01	-0.01	0.01	0.02	0.03	-0.03	0.01	-0.00	0.04	0.00	0.00	0.01	-0.05	-0.03	0.26	0.25	-0.01	-0.02	0.00	0.04	0.00	-0.11	0.00	0.04	0.01	0.08
2	Lucene	-0.04		-0.00	0.01	-0.04	0.01	0.01	-0.01	0.01	-0.00	-0.00	-0.02	0.03	-0.00	-0.01	-0.00	0.01	-0.02	0.00	0.03	0.03			-0.01	0.01	-0.02	0.01	0.01	0.03	-0.00	0.07	-0.03	-0.03
3	Mylyn	0.01	0.03		0.01	-0.05	-0.01	-0.03	0.03	-0.02	0.02	-0.01	0.04	-0.07	0.01	-0.00	-0.01	-0.01	0.00	-0.01	-0.03	0.01	0.28	0.02	0.04	0.03	0.04	-0.01	0.03	-0.02	0.09	-0.04	0.03	0.03
4	Eclipse PDE UI	-0.02	0.00	-0.00		0.09	-0.02	-0.03	0.01	-0.00	-0.00	-0.01	-0.03	-0.00	-0.00	-0.00	-0.02	-0.01	-0.01	0.01	0.02	0.02	0.46	-0.09	-0.04	0.04	0.02	0.02	-0.06	0.04	-0.07	-0.03	-0.08	0.00
5	Android U. I. L.	0.02	0.00	0.02	-0.04		-0.03	-0.07	-0.04	-0.02	-0.02	-0.12	-0.01	-0.04	-0.02	-0.02	-0.04	-0.06	0.03	-0.01	-0.03	0.10	-0.04	0.12	-0.06	-0.10	0.05	-0.02	0.03	0.08	-0.10	-0.04	0.00	-0.05	-0.04
6	ANTLR	0.03	0.00	-0.00	0.00	0.03		-0.03	0.00	-0.00	-0.01	-0.00	-0.02	-0.01	0.00	0.00	0.02	0.00	-0.00	0.01	-0.03	0.01	-0.03	-0.05	0.00	-0.02	-0.00	0.00	-0.02	-0.02	0.03	0.04	-0.01	-0.10
7	Broadleaf	-0.04	0.00	0.01	-0.00	-0.07	-0.03		-0.02	-0.01	-0.00	-0.01	-0.03	-0.04	0.01	0.00	0.01	0.00	0.02	-0.01	0.03	-0.02	0.28	0.02	-0.02	0.00	0.00	0.10	0.02	-0.00	-0.09	-0.05	0.04	0.03
8	Ceylon	-0.04	-0.00	0.02	-0.01	0.03	0.02	0.01		0.01	-0.01	-0.00	-0.03	0.02	0.00	0.01	0.03	-0.01	0.01	0.01	-0.00	0.00			-0.04	-0.01	-0.01	-0.07	0.05	-0.02	-0.00	0.02	0.01	-0.04
9	Elasticsearch	-0.02	-0.01	-0.02	-0.01	0.03	-0.01	0.02	-0.01		0.02	-0.01	-0.03	-0.01	0.00	-0.01	-0.01	-0.00	0.01	-0.01	-0.01	-0.00	-0.09	-0.07	-0.01	-0.02	-0.03	0.02	-0.02	-0.02	-0.01	-0.05	0.04	-0.12
10	Hazelcast	0.02	0.00	-0.00	0.00	0.06	-0.02	-0.03	0.01	-0.01		0.01	-0.02	-0.03	-0.00	-0.00	0.02	0.03	-0.02	0.01	0.01	0.00		-0.07	0.01	-0.02	0.11	0.01	-0.14	0.00	-0.06	-0.00	0.12	-0.02
11	JUnit	0.00	-0.02	0.00	-0.04	-0.02	0.01	0.05	-0.03	-0.01	-0.02		0.00	0.03	-0.01	-0.01	0.02	0.00	0.05	0.00	0.03	0.04	0.25	0.07	-0.02	-0.09	0.07	0.06	0.13	0.11	0.03	0.02	0.01	0.24
12	MapDB	-0.02	-0.01	0.01	0.02	0.04	-0.00	-0.03	0.01	0.02	-0.01	-0.02		0.05	0.00	-0.01	0.01	-0.01	-0.02	-0.01	-0.02	-0.01		0.04	0.03	0.00	-0.02	-0.03	-0.02	-0.09	-0.07	0.13	-0.12	-0.06
13	memMO	0.05	0.00	-0.01	0.02	0.01	0.04	0.02	-0.01	-0.01	-0.05	0.01	0.00		-0.02	-0.01	-0.03	-0.03	-0.01	0.00	-0.01	0.03		0.07	0.02	0.08	0.05	-0.13	-0.09	-0.09	0.02	-0.13	-0.10
14	MCT	-0.01	0.01	-0.01	0.00		-0.02	0.00	-0.01	-0.00	-0.00	-0.00	0.02	-0.00		-0.00	-0.02	0.00		-0.00	-0.01	-0.00			-0.01	0.00	0.01		-0.01	-0.00	-0.05	-0.03	-0.07
15	Neo4j	0.02	-0.00	0.00	0.01	-0.02	0.00	-0.01	0.00	-0.00	-0.01	0.00	0.01	0.02	-0.00		0.00	0.00		-0.00	-0.01	-0.01			-0.01	-0.01		0.02		-0.01		0.02	0.01
16	Netty	-0.00	-0.02	-0.00	0.03	-0.05	-0.02	0.02	0.01	-0.02	0.02	-0.00	-0.00	-0.01	-0.02	-0.00		-0.04	0.04	0.00	-0.05	0.03		-0.05	0.01	0.00	-0.05	0.05	-0.04	-0.01	0.08	0.01	0.04	0.08
17	OrientDB	0.00	0.00	0.01	0.03	0.00	0.02	-0.03	-0.01	-0.00	-0.02	-0.01	-0.06	0.01	-0.00	-0.01	0.02		0.01	0.00	0.01	0.02		-0.03	-0.02	-0.01	-0.00	-0.01	-0.05	0.03	0.02	-0.04	0.08	-0.10
18	Oryx	-0.06	0.01	0.01	0.01	0.08	0.01	-0.00	-0.02	-0.00	-0.01	0.01	-0.02	-0.05	0.02	0.01	-0.03	-0.01		0.00	0.01	-0.01	0.28	-0.01	0.05	0.01	-0.01	-0.01	0.08	-0.05	0.05	-0.06	-0.06	-0.08
19	Titan	-0.03	0.02	0.00	-0.01	-0.01	-0.01	-0.00	-0.01	-0.00	-0.01	-0.00	-0.02	0.02	-0.01	0.00	-0.00	-0.00	-0.02	0.06	-0.02	-0.00		0.05	-0.01	0.02	0.04	0.05	0.00	-0.02	-0.02	0.07	0.04	0.02
20	Ant 1.7	0.08	-0.00	-0.01	-0.02	0.01	0.05	0.03	0.09	0.07	0.06	0.15	0.07	0.01	0.06	0.11	0.06	0.01	-0.02	0.06				-0.05	-0.01	-0.01	0.00	-0.05	0.00	0.00	-0.05	0.09	0.02	0.00
21	Camel 1.6	0.01	0.02	-0.00	-0.00	-0.01	-0.01	0.00	-0.00	-0.01	0.00	0.00	-0.00	-0.05	-0.02	-0.02	0.02	-0.00	-0.04	0.01	-0.05		0.35	-0.07	-0.03	-0.04	-0.00	0.06	0.07	0.09	0.06	-0.15	0.05	0.14
22	Ckjm 1.8	-0.00	0.09	0.00	0.08	-0.04	0.04	-0.02	0.05	0.04	0.12	0.08	0.05	-0.05	0.11	0.09	0.03	0.07	-0.01	0.02	0.12	-0.04		-0.04	-0.03	-0.01	0.04	-0.03	0.00	-0.09	0.17	-0.01	-0.03	-0.07
23	Forrest 0.8	-0.02	0.06	-0.01	0.04	-0.03	0.05	0.01	0.03	0.01	0.05	0.03	0.01	0.04	0.05	0.04	-0.02	0.01	0.00	0.07	0.05	-0.02			0.01	0.04	-0.09	-0.00	0.02	-0.15	-0.05	0.02	-0.11	0.05
24	Ivy 2.0	-0.02	-0.02	0.01	0.02		0.02	0.01	-0.00	0.01	-0.00	-0.00	0.03	-0.01	-0.00	0.00	-0.01	0.00	-0.01	0.00	0.01	0.01		-0.05		0.05	-0.09	0.01	-0.08	-0.06	-0.04	0.03	-0.00	0.02
25	JEdit 4.3	-0.01	-0.00	-0.00	0.00		0.01	-0.00	-0.00	-0.01	-0.00	0.01	-0.01	-0.01	-0.01	-0.00	-0.01	-0.01		0.00	0.01	-0.01			-0.00		0.02	0.00	0.00	0.03	-0.01	0.00	0.11	-0.02
26	Log4J 1.2	-0.06	0.03	0.01	0.04	0.04	0.19	0.01	-0.01	0.08	-0.04	0.05	0.03	0.08	0.07	0.03	0.04	-0.03	-0.05	0.01	0.01	0.07		-0.11	0.06	0.11		0.00	0.10	0.05	-0.02	-0.00	-0.01	-0.03
27	Lucene 2.4	-0.08	0.00	0.04	-0.00	-0.05	-0.02	-0.04	-0.03	0.01	-0.01	-0.05	-0.04	-0.00	-0.02	-0.02	-0.02	-0.01	0.00	-0.03	-0.00	-0.03	0.00	0.00	-0.08	-0.05	0.06		-0.12	0.02	0.02	-0.01	0.05	0.11
28	Pbeans 2	-0.06	-0.03	0.04	-0.02	0.05	0.04	0.01	0.01	-0.00	0.03	0.01	-0.01	0.01	0.01	-0.02	0.00	0.00	0.04	0.01	-0.01	0.00	0.13	-0.04	0.03	-0.09	0.12	0.06		-0.05	0.05	-0.03	0.12	0.05
29	Poi 3.0	0.08	0.01	-0.03	0.01	0.04	-0.02	0.07	0.07	0.01	0.05	0.08	0.07	0.15	0.03	0.03	0.04	0.01	0.05	0.05	0.01	-0.01	-0.16	-0.12	0.03	0.01	-0.02	-0.01	0.01		0.07	0.00	0.02	0.07
30	Synapse 1.2	0.12	-0.03	0.05	0.01	0.03	-0.06	0.03	0.04	-0.00	0.00	0.02	0.04	0.05	0.02	0.00	0.06	0.04	-0.01	-0.03	-0.05	-0.06	0.00	0.02	-0.00	-0.00	0.03	-0.02	0.12	0.03		0.02	0.00	-0.09
31	Velocity 1.6	-0.03	0.03	0.05	0.00	-0.01	-0.03	0.04	-0.03	0.02	0.02	-0.01	-0.05	-0.03	0.00	0.02	0.01	-0.04	0.04	0.01	0.04	-0.00	0.00	0.01	0.04	-0.03	0.04	-0.02	0.10	-0.13	0.04	0.01	0.01	0.12
32	Xalan 2.7	-0.05	-0.04	0.02	-0.02	0.02	-0.02	0.01	-0.02	-0.00	-0.01	-0.05	0.01	0.02	0.03	0.00	0.03	0.03	-0.02	-0.01	-0.01			-0.06	-0.01	0.01	-0.02	0.01		0.01	0.01	-0.02		0.00
33	Xerces 2.0	-0.11	-0.06	-0.02	-0.10	-0.05	0.04	-0.05	-0.08	-0.06	-0.08	-0.13	-0.08	0.08	-0.11	-0.10	-0.10	-0.09	0.01	-0.16	-0.04	-0.04		-0.06	-0.08	0.03	-0.04	0.02	-0.17	-0.01	0.07	0.04	-0.00	

for classification, the calculated weighted F-measure was already low, and it was unable to predict if the class is bugged or not correctly.

Most of the increase can be seen when the classifier was trained on the JUnit, Ant 1.7, Ckjm 1.8, Log4J 1.2, or Poi 3.0 projects. For these systems, there are multiple instances where the F-measure change is greater or equal to 0.05, and there are only a few instances where a decrease can be observed.

The two small systems, Ckjm 1.8 and Forrest 0.8, improve overall. Additionally, using these systems as test data could make the classificator over-fit in some cases, as shown by the empty cells in rows ID 22 and 23. However, in other cases, there are quite big improvements when using bigger systems for training and these two as test data. For example, in the case of the Equinox training data, the F-measure improvements were 0.26 and 0.25.

Overall, from the 1089 results, there are 123 instances where the weighted F-measure changes are greater than or equal to 0.05. In 107 cases, the F-measure changes are less than -0.05. These cases indicate that adding the PE metrics can be beneficial for bug prediction. Furthermore, the increase of the weighted F-measure is greater than the decrease presented in the change matrix. It is important to note that, as the PE metric values are intended to be used inside a single system – and not comparing system to system –, this kind of usage is less favourable for these new metrics. Adding the PE metrics for further bug prediction classifications can be worthwhile based on these data.

12.4 Bug Prediction Across Versions

Training and testing the bug prediction across projects is a good idea and can be useful. However, during a software development process, usually a single application is developed from which multiple releases are created. As the system evolves, an earlier project version could be used to predict bugs. In this case, an earlier version can be used to train a bug prediction model and used on later versions. Fortunately, the selected bug dataset contains multiple versions for some systems. There are 12 such systems, which are Ant, Camel, Ivy, JEdit, Log4J, Lucene, Pbeans, Poi, Synapse, Velocity, Xalan, and Xerces.

During the comparison of the other projects with multiple versions, the most negative changes can be observed in the case of the Ant system, and all other projects have better values. The Ant system's comparison is depicted in Table 12.6 and like before, the first column describes the systems on which the training was performed and the other columns are the systems where the evaluation was done.

Table 12.6: *Weighted f-measure changes in the case of the Ant project across versions*

	Ant 1.3	Ant 1.4	Ant 1.5	Ant 1.6	Ant 1.7
Ant 1.3		-0.01	0.01	-0.07	-0.01
Ant 1.4	-0.01		0.05	-0.02	-0.02
Ant 1.5	-0.05	-0.03		0.02	0.01
Ant 1.6	-0.08	-0.01	0.03		-0.02
Ant 1.7	0.00	-0.02	-0.02	0.03	

In this case, there are a bit more than 10 cases where the addition of the PE metrics resulted in slightly negative values. The largest impact was Ant 1.6 and Ant 1.3, which resulted in a -0.08 change in the F-measure values. Still, most of the changes are between the -0.05 and 0.05 ranges, indicating that the change is minimal in most cases. We can conclude that the changes are small overall, and in most cases, the F-measure values changed slightly.

The Velocity project gave one of the best results with the addition of the PE metrics. The improvement in the weighted F-measure value can be observed in Table 12.7.

Table 12.7: *Weighted f-measure changes in the case of the Velocity project across versions*

	Velocity 1.4	Velocity 1.5	Velocity 1.6
Velocity 1.4		0.03	0.09
Velocity 1.5	0.13		-0.00
Velocity 1.6	0.03	0.03	

In almost every version combination, the addition of PE metrics improved the F-measure value. If the 1.4 version was used as the training data, the evaluation of the bug prediction improved on newer versions of the project. Interestingly, using a newer version as a base and making predictions for an older system also gave positive results.

In the case of cross-project evaluation, the Xerces 2.0 system's weighted F-measure changes were usually negative, as previously shown in Table 12.5. In this instance, the model trained with Xerces 2.0 could not improve the

prediction for other projects. In this case, it is worth investigating what would happen if the various Xerces versions were compared.

Table 12.8: *Weighted f-measure changes in case of the Xerces project across versions*

	Xerces 1.2	Xerces 1.3	Xerces 2.0
Xerces 1.2		0.02	-0.00
Xerces 1.3	0.02		0.06
Xerces 2.0	-0.01	-0.01	

This comparison can be observed in Table 12.8. This table describes that using an older Xerces version for training dataset with the added PE metrics can improve the bug prediction capabilities for future Xerces versions. The most notable change is when Xerces 1.3 was used as a training dataset, and the 2.0 version was used to evaluate the predictions.

Based on these experiments, the addition of PE metrics to perform bug prediction across multiple versions is a viable option.

13 Conclusions

This part presented a set of new metrics to define one aspect of Primitive Obsession code smell. Mainly the over usage of primitively typed parameters. First, we discussed the new Primitive Enthusiasm metric and its variants. Investigated results of these formulae on a selected set of systems. In order to do this, the metric calculation was implemented in a Java static code analyser. The results show that Primitive Enthusiasm metrics can detect such methods, classes that could indicate a truly primitive obsessed element in a system.

Second, the new metric's connection with other metrics was presented, and the bug prediction capabilities were discussed. For this, a cross-project validation was done to see how the F-measure values are changed by adding the metrics to the already existing metrics dataset. Although there were cases where the trained models performed less adequately, overall there were more improvements in the classification of bugged classes. A different approach was also presented where a project version-based bug prediction evaluation was done. In this case, the addition of the new PE metrics was also able to add improvements in terms of weighted F-measure.

Based on these data, we recommend the adoption of the new Primitive Enthusiasm metrics in real-world use-cases and other source code metric research as it tries to quantify the previously less investigated Primitive Obsession code smell.

II. 13. Conclusions

Part III

Appendices

A

Summary

Software maintenance is a big and diverse topic that focuses not only on fixing defects found in applications, but also on software re-engineering, source code analysis, calculation/evaluation of source code metrics, and detection of various code bad smells. Out of these diverse topics the author worked on two areas and the thesis is divided accordingly into two parts. The experiments with the A+ programming language and the investigation of the Primitive Obsession bad smell. Both of these topics are tightly related software maintenance and software quality aspects.

I. Experiments with the A+ Programming Language

In this thesis point, one of the goals was to extend the lifetime of existing A+ applications by providing a new runtime that can be made faster and could be maintained with less effort. In addition, the new object-oriented operations also help the binding between .NET and A+ worlds making adoptions smoother.

In Chapters 4, 5, and 6 the contributions for the first thesis point are discussed. This thesis point can be separated into the following two main results.

III. A. Summary

1. A+.NET Implementation and Comparison of Runtimes

A+ is an array programming language [53] inspired by APL. It was created more than 30 years ago to suite the needs of real-life financial computations and even even nowadays, many critical applications are used in computationally-intensive business environments. Unfortunately, the original interpreter-based execution environment of A+ is implemented in C and is officially supported on Unix-like operating systems only.

By creating a clean-room implementation on top of .NET, the A+.NET can extend the lifetime of existing A+ applications and also makes it possible to run them not just on Unix-like systems. To test and prototype the interpreter the A+ language's grammar was defined in ANTLR grammar format. Using this new – previously non-existent – formalized grammar and the Dynamic Language Runtime with the .NET framework the base of the interpreter was built. Leveraging the capabilities of DLR the interoperability between A+ and .NET was achieved. The methods to expose .NET elements into the A+.NET runtime presented in the thesis could be useful for A+ and .NET developers alike.

After the new A+.NET implementation was created, it was compared to the original A+ runtime. Comparison was done in terms of script execution runtime and source code/maintainability metrics. The runtime comparison was done using a code fragment that was extracted from a real-life code base. This code fragment performs URL encoding on an input string. By using this script, the measurements show that the original interpreter is significantly faster than the clean-room implementation. This is not surprising as the original runtime had more time to add various optimizations. But, the ease of exposing method into the A+.NET runtime makes it possible to replace parts of the URL encoding A+ code. By replacing the A+ part where strings are joined with the .NET equivalent method string join method, there was 30% speedup. Going further, the replacement of the whole URL encoding function with the .NET counterpart the execution time dropped to 20% of the time measured for the reference implementation, which is equivalent to a 5-fold speedup. This speedup make sense as there is no additional A+.NET overhead when the .NET equivalent URL encoding is executed.

In terms of source code metrics, we determined functionally equivalent parts between the two runtimes by using a set of A+ test scripts to calculate source code function-level coverage. The two function sets, one in each system, is of comparable size and of equivalent functionality and was used for further inves-

tigation into the maintainability of the two systems. We used the Columbus toolchain to analyse both interpreter's sources and as a result we got two size metrics – LOC and NOS – and two complexity metrics – McCC and NLE – for each function. The averages of NOS, McCC, and NLE, and the maximums of all metrics show that the size and the complexity of the functions in the reference implementation are higher than in A+.NET. We also experimented and investigated derived metrics. The calculation of statements per line metric (NOS/LOC) revealed that in the reference implementation the average number of statements in every executable line of source code is about 3. Moreover, the most "crowded" function contains 37 top-level statements in a line on average. This instance turned out to be a single line function. Overall, 30% of the investigated functions of the reference interpreter have more than two statements on a line on average. For the A+.NET variant this is 0%. The combination of McCC and NOS metrics re-confirmed that circa 70% of the compared A+.NET implementation functions are small and less complex methods. Overall, the A+.NET version displayed better results in terms of maintainability.

2. A+.NET Language Extension

The second main result of this thesis point is the new A+ language extension. A set of required operations is presented in order to allow access to various object-oriented components in a language. These are the member access, member set, indexing, and type cast operations. For these operations extra care was taken to make sure that the new syntax and semantics do not collide with existing elements of the language. The right-to-left evaluation order was kept and the context handling of the language was not changed. In order to help resolve ambiguous method calls a vector based type matching algorithm was also discussed. Using these new operations A+ developers can quickly access .NET classes in their code without writing wrapper methods. Furthermore, these new operations are not strictly tied to the A+ language and could be adapted for use in other languages.

The Author's Contributions

The author worked on designing and developing the A+.NET clean-room implementation. He designed the formal grammar for A+ that was previously non-existent. The comparison and evaluation of the two A+ runtimes were carried out by the author both in terms of runtime and in terms of source code/maintainability metrics. The author constructed and formalized four new

operations in order to allow object-oriented components to be used in the A+ language. To resolve method call ambiguities, the author formalized a type vector based approach.

The publications related to this thesis point are the following:

[24] **Péter Gál** and Ákos Kiss. Implementation of an A+ Interpreter for .NET. In *Proceedings of the 7th International Conference on Software Paradigm Trends (ICSOFT 2012)*, pages 297-302, Rome, Italy, July 2012. SciTePress.

[25] **Péter Gál** and Ákos Kiss. A Comparison of Maintainability Metrics of Two A+ Interpreters. Proceedings of the 8th International Joint Conference on Software Technologies (ICSOFT 2013), pages 292-297, Reykjavík, Iceland, July 2013. SciTePress.

[23] **Péter Gál**, Csaba Bátori, and Ákos Kiss. Extending A+ with Object-Oriented Elements: A Case Study for A+.NET. In *21st International Conference on Computational Science and Its Applications (ICCSA 2021)*, Proceedings, Part IX, volume 12957 of Lecture Notes in Computer Science (LNCS), pages 141-153, Cagliari, Italy, September 2021. Springer. Best paper award.

II. Primitive Enthusiasm Metrics

In this thesis point our goal was to quantify the "too many primitives" definition seen in the description of the Primitive Obsession bad smell. This was achieved by the creating of the Primitive Enthusiasm metrics and was shown that they can be used to improve bug predictions.

In Chapters 10, 11, and 12 the contributions for the second thesis point are discussed. This thesis point can be separated into the following two main results.

3. Definition and Evaluation of Primitive Enthusiasm Metrics

A previously less investigated Primitive Obsession code smell is decomposed. The original idea that "too many primitives are used" is a bit vague. In order to resolve this, a concrete calculable metric was created to capture part of the Primitive Obsession bad smell, and it is called Primitive Enthusiasm (PE). This metric captures the number of primitively typed parameters for a method in a given class and compares it the averages of the same class. Based on this metric

two other variants were created called Global Primitive Enthusiasm (GPE), and Hot Primitive Enthusiasm (HPE). The original metric was renamed to Local Primitive Enthusiasm (LPE). The GPE variant changes the metric's formulae in way that it is now compares the current function under investigation to the whole system. HPE then incorporates both LPE and GPE results into a single result. These metric calculations were implemented into the Open Static Analyzer [6] framework to evaluate the PE metrics on selected systems. During the evaluation two additional aspects were investigated. First, the option to skip methods was added. For this there were two variants: to skip only setter methods (SSM) or skip every method with just one parameter (SO). Based on the results the SO variant proved to be better in terms of reducing noise in the reported results. The other option added was related to Java wrapper classes that could be treated as primitives. The experiment revealed that by treating the wrapper classes as primitives the results did not change drastically. Based on this it is a good idea to treat the wrapper classes as primitives.

4. The Bug Prediction Capabilities of the Primitive Enthusiasm Metrics

After this, the bug prediction capabilities of the new metrics were investigated. For this, an already existing bug dataset was used that contained pre-calculated metrics. In order to add the PE metrics to this dataset, they had to be aggergated to class-level. An interesting experiment was to see correlation between the already existing metrics and the new PE metrics. Not surprisingly there were positive correlations between PE and older metrics that are using the number of parameters as their base. The bug prediction capabilities were tested by adding the new metrics to the data set. The original and extended datasets were trained and evaluated to see the F-measure changes between them. Furthermore, this training and evaluation was done in two ways. First, a cross-project based evaluation was performed on 33 selected systems. In 123 of the 1089 cases, the change in the weighted F-measure is greater or equal than 0.05. And in 107 cases the changes were less than -0.05. In the second experiment the training and evaluation was done across project versions. This concluded that adding the PE metrics to perform bug prediction across multiple versions is a viable option.

The Author's Contributions

The author designed the original Primitive Enthusiasm metric. Implemented

the calculation of this metric into a static analyzer for Java systems. The author participated in the selection of the analyzed systems and the evaluation of the original Primitive Enthusiasm metric. He designed the experiment to see how Java wrapper classes affect the metric results. Based on the Primitive Enthusiasm metric, the LPE, GPE, and HPE metrics were formalized by the author. With the usage of an existing bug dataset, correlations between the new metrics and other existing ones were investigated by the author. For the bug prediction capabilities, the target systems were selected by the author. He executed and evaluated the cross-project based bug prediction experiment with the addition of PE metrics. The version-based bug prediction investigation was also done by the author.

The publications related to this thesis point are the following:

[26] **Péter Gál** and Edit Pengő. Primitive Enthusiasm: A Road to Primitive Obsession. In *The 11th Conference of PhD Students in Computer Science (CSCS 2018)*, Volume of short papers, pages 134-137, Szeged, Magyarország, June 2018.

[57] Edit Pengő and **Péter Gál**. Grasping Primitive Enthusiasm - Approaching Primitive Obsession in Steps. In *Proceedings of the 13th International Conference on Software Technologies (ICSOFT 2018)*, pages 389-396, Porto, Portugal, July 2018. SciTePress.

[22] **Péter Gál**. Bug Prediction Capability of Primitive Enthusiasm Metrics. In *21st International Conference on Computational Science and Its Applications (ICCSA 2021)*, Proceedings, Part VII, volume 12955 of Lecture Notes in Computer Science (LNCS), pages 246-262, Cagliari, Italy, September 2021. Springer.

B
Összefoglalás

A szoftverkarbantartás egy nagy és szerteágazó téma, amely nem csak a az alkalmazásokban talált hibák javítására koncentrál, hanem szoftver újratervezéssel, forráskód metrikákkal, forráskód elemzéssel, és gyanús kódok detektálásával. A disszertáció kettő fő tézis csoportba kategorizálja az elért eredményeket. Az első tézis az A+ nyelvvel végzett kísérleteket taglalja, míg a második tézis az ú.n. Primitive Obsession[1] gyanús kóddal foglalkozik.

I. Kísérletek az A+ programozási nyelvvel

Az első tézispontban az egyik cél a meglévő A+ alkalmazások élettartamának a meghosszabbítása volt. Ez sikerült is, mivel az új .NET megvalósítás ámbár lassabb jelenleg, de több platformon használható, forráskód metrikái jobban, és könnyebben kiegészíthető .NET környezetből érkező eljárásokkal. Valamint, az új objektumorientált műveletek segítségével az A+ és .NET világok között egyszerűbb az átjárás.

A 4, 5, és 6 fejezetekben az első tézisponthoz tartozó eredmények kerülnek bemutatásra. Ez a tézispont két fő eredményre osztható.

[1]Nincs elfogadott magyar fordítása, szó szerinti jelentése “primitív megszállottság”. A primitív típusok indokolatlanul túlzásba vitt használatát takarja.

III. B. Összefoglalás

1. A+.NET fejlesztése és a végrehajtó motorok összehasonlítása

Az A+ egy tömb alapú programozási nyelv [53], amelyet az APL ihletett. Több mint 30 éve hozták létre a valós pénzügyi számítási igények kielégítésére, és még napjainkban is számos kritikus A+ alkalmazást használnak üzleti környezetben. Sajnos az A+ eredeti, interpretert C nyelven implementálták, és hivatalosan csak Unix-szerű operációs rendszereken támogatott.

Az A+.NET egy .NET alapon létrehozott A+ interpreter, mely lehetőséget ad arra, hogy a meglévő A+ alkalmazások használati ideje hosszabbodjon, mivel nem csak Unix-szerű rendszereken képes működni. Azonban, az interpreterhez első lépésben az A+ nyelvtanát formalizáltuk, és az ANTLR [56] lexer-parser generátorral használtuk fel. Az A+-nak eddig nem létezett formális nyelvtana. A generált lexer-parser és a Dynamic Language Runtime használatával .NET alapon készült el az értelmező alapja. A DLR képességeinek kihasználásával megvalósult az átjárhatóság az A+ és a .NET világ között. A disszertációban bemutatott eljárásokkal .NET fejlesztők képesek különféle eljárásokat, osztályokat az A+ fejlesztők részre bocsájtani. Hasonlóan, a .NET fejlesztők is képesek A+-ban írt eljárásokat, adatokat elérni-

Miután elkészült az új A+.NET implementáció, összehasonlítottuk azt az eredeti A+ interpreterrel. Az összehasonlítás során az A+ szkript végrehajtási sebességét és az interpreterek forráskód/fenntarthatósági metrikáit vizsgáltuk. A végrehajtási idő összehasonlítás egy valós kódbázisból származó kódrészlet segítségével történt. Ez a kódrészlet URL-kódolást végez egy bemeneti karakterláncon. Ennek használatával a mérések azt mutatták, hogy az eredeti interpreter jelentősen gyorsabb, mint a .NET implementáció. Ez nem meglepő, mivel az eredeti futtatókörnyezetnek több mint 20 éve volt az optimalizálásra. De .NET esetében azzal, hogy egyszerűen lehet .NET eljárásokat az A+ szkripteknek használhatóvá tenni, lehetőséget adott arra, hogy az URL-kódolást megvalósító eljárás egy vagy teljes részét lecseréljük. Azzal, hogy lecseréltük a karakterek összefűzéséért felelős kódrészletet egy megfelelő .NET eljárással, 30%-os gyorsulást értünk el a normál A+.NET végrehajtási sebességhez képest. Továbbhaladva a kísérlettel a teljes URL-kódolását felelős eljárást egy .NET alternatívával helyettesítve, a végrehajtási idő a referencia implementáció idejének a 20%-ra csökkent A+.NET esetén. Ez ötszörös gyorsulásnak felel meg.

Annak érdekében, hogy a két értelmezőt megfelelően össze tudjuk hasonlítani forráskód metrikákkal, elsőként mindkét rendszerben meghatároztuk a funkcionálisan azonos részeket. Ezt oly módon tettük, hogy A+ teszt

szkripteket használva meghatároztuk a lefedett eljárásokat. Az így meghatározott részhalmazok összehasonlítható méretűek a két rendszerben és ugyanazokat a funkcionalitásokat látják el összességében. A forráskód metrikák előállításához a Columbus [18] eszközt használtuk fel, mely mindkét forráskódra kiszámolt különféle forráskód metrikákat. Kettő méret alap metrika – LOC és NOS – és kettő komplexitást reprezentáló metrikát – McCC és NLE – használtuk fel a további vizsgálatok során. A NOS, McCC és NLE átlagai és maximum értékei azt mutatták, hogy a referencia implementációban a függvények nagyobbak és bonyolultabbak, mint az A+.NET esetében. Kísérletünk folytatásával, kettő származtatott metrikát is megvizsgáltunk. Az átlagos soronkénti utasítás (NOS/LOC) metrika értékek azt mutatták, hogy az eredeti implementációban átlagosan 3 utasítás található. Ezen kívül egy kirívó esetben az egyik "legzsúfoltabb" eljárásban 37 utasítás található egy sorban. Ez az eset ráadásul egy egy soros eljárás. Összességében, a referencia megvalósításban a vizsgált eljárások 30%-ban több mint két utasítást találtunk egy sorban. Ezzel szemben a .NET változatban nem volt egy ilyen eljárás se. A McCC és az NOS metrikák kombinációja megerősítette, hogy az összehasonlított eljárások a A+.NET esetben kb. 70%-a kicsi és kevésbé összetett. Összességében az A+.NET változat jobb eredményeket mutatott a szoftver karbantarthatóság tekintetében.

2. A+.NET nyelv kiterjesztése

A tézis második fő eredménye egy A+ nyelvi kiterjesztés. Bemutatásra került egy sor művelet mely segítségével lehetőség adódik arra hogy különböző objektumorientált elemekhez férjen hozzá dinamikusan egy A+ fejlesztő. Ezek a műveletek az adattag hozzáférés, adattag módosítás, indexelés és típus változtató eljárások. A eljárások kidolgozása során különösen figyeltünk arra, hogy az újonnan bevezetett szintaxis és szemantika ne ütközzön a meglévő nyelvi elemekkel. Megtartottuk a jobbról-balra történő kiértékelési szabályt és az A+ nyelv kontextus kezelése sem változott. Annak érdekében, hogy megoldjuk a nem egyértelmű eljárásívási eseteket egy vektor alapú típusillesztési algoritmust dolgoztunk ki. Ezek új műveltek használatával az A+ fejlesztők könnyebben hozzáférhetnek a különféle .NET osztályokhoz, azok adattagjaihoz anélkül, hogy csomagoló eljárásokat kelljen írniuk. Érdemes megemlíteni, hogy az új nyelvi szabályok nem A+ nyelv függőek. Ezek alkalmazhatóak lennének más nyelv esetében is ahol hasonlóan hiányoznak az objektumorientált komponensek. Fontos azonban megemlíteni, hogy ettől még az A+ nem lett objektumorientált, mivel osztály megadására még mindig nincs lehetőség

A+-ban.

A szerző hozzájárulásai

A szerző megtervezte és fejlesztette A+.NET interpretert. Megtervezte az A+ formális nyelvtanát ANTLR nyelvtan formátumban. Az eredeti és új futtató környezetet összehasonlította A+ szkript végrehajtási sebesség és forráskód/karbantarthatósági metrikák tekintetében. A szerző megtervezte és formalizálta a külső objektumok kezelését lehetővé tevő eljárásokat. Ehhez a szerző megalkotta a vektor alapú az A+ nyelvben való felhasználását. Ehhez a szerző megalkotta a vektor alapú típusillesztési algoritmust.

Az első tézisponthoz a következő publikációk kapcsolódnak:

[24] **Péter Gál** and Ákos Kiss. Implementation of an A+ Interpreter for .NET. In *Proceedings of the 7th International Conference on Software Paradigm Trends (ICSOFT 2012)*, pages 297-302, Rome, Italy, July 2012. SciTePress.

[25] **Péter Gál** and Ákos Kiss. A Comparison of Maintainability Metrics of Two A+ Interpreters. Proceedings of the 8th International Joint Conference on Software Technologies (ICSOFT 2013), pages 292-297, Reykjavík, Iceland, July 2013. SciTePress.

[23] **Péter Gál**, Csaba Bátori, and Ákos Kiss. Extending A+ with Object-Oriented Elements: A Case Study for A+.NET. In *21st International Conference on Computational Science and Its Applications (ICCSA 2021)*, Proceedings, Part IX, volume 12957 of Lecture Notes in Computer Science (LNCS), pages 141-153, Cagliari, Italy, September 2021. Springer. Best paper award.

II. Primitive Enthusiasm[2] metrikák

Ebben a tézispontban a fő célunk a Primitive Obsession leírásában található "túl sok primitív típus használata" számszerűsítése volt. Ehhez megalkottuk a Primitive Enthusiasm metrikákat és megvizsgáltuk a hiba előrejelzési képességét.

A 10, 11, és 12 fejezetekben a második tézisponthoz való hozzájárulásokat tárgyaljuk. Ez a tézispont is két fő eredményre osztható.

[2]A Primitive Obsession nevéhez hasonlóan a Primitive Enthusiasm metrikának sincs hivatalos magyarítása. Neve szó szerinti fordításban "primitív(ekért) lelkesedés".

3. A Primitive Enthusiasm Metrikák Definiálása és Kiértékelése

Ebben a részeben egy korábban kevésbé vizsgált gyanús kóddal foglalkoztunk, a Primitive Obsession-el. Az eredeti gondolat, miszerint "túl sok primitív adattípus van használva" nem jól behatárolható. Ezért egy konkrétan kiszámítható metrikát hoztunk létre ennek a gyanús kódnak az egyik aspektusának a mérésére. Ezt Local Primitive Enthusiasm (LPE)-nak neveztük el. A metrika egy adott eljárás primitív típusú paramétereinek az arányát hasonlítja össze az aktuálisan vizsgált osztályban található primitív paramétereknek az arányával. Az ötlet az, hogy egy globális határérték megadása helyett az osztály saját maga adja meg a limitet mikor gyanús egy eljárás. Ennek a metrikának számítását beleintegráltuk az Open Static Analyzer [6] keretrendszerbe és három Java alapú rendszeren kiértékeltük. A kiértékelés során megvizsgáltuk, hogy mi történik ha kihagyunk eljárásokat amikben csak egy argumentum van és akkor ha a "setter" eljárásokat hagyunk ki. Ebben az esetben a jobb eredményt az hozta, ha minden olyan eljárást kihagyunk a kiértékelésből amiben csak egy argumentum van. Ezt a stratégiát ajánlott a későbbiekben is használni. A másik stratégia a Java csomagoló osztályokkal kapcsolatos. Ebben kísérletben megvizsgáltuk, hogy ha a csomagoló osztályokat primitív típusoknak tekintjük vagy éppen ellentétben nem tekintjük annak, miként változnak a megtalált eljárások mennyiségei. A kísérlet eredménye az, hogy célszerű a csomagoló osztályokat is primitív típusoknak tekinteni mivel a kiértékelt rendszereken alig fordultak elő és nem igazán változtak az eredmények. Az LPE metrika alapján kettő másik változatot is készítettünk, A Global Primitive Enthusiasm (GPE) és a Hot Primitive Enthusasm (HPE)-t. A GPE változat esetén az aktuális függvény a rendszerben található összes eljáráshoz van arányosítva, ezzel egy más határértéket adva mint az LPE esetén. A HPE pedig az LPE-t és GPE-t kombinálja egyetlen eredménybe.

4. A Primitive Enthusiasm metrikák hiba előrejelzési képességei

A tézispont második fő eredményeként megvizsgáltuk a Primitive Enthusiasm metrikák hiba előrejelzési képességeit. Ehhez egy már meglévő hiba adatbázist [19] használtunk fel, mely már tartalmazott meglévő metrikákat. Így ezt csak a PE metrikákkal kellett kiegészíteni. Azonban, ehhez a PE metrikákat osztály szinten aggregálni kellett. Egy másik érdekes kísérletben megnéztük, hogy mekkora korreláció van a meglévő metrikák és a PE metrikák között. Bizonyos eljárás számosságot figyelembe vevő metrika esetén kapcsolat van a PE metrikákkal. De összességében a PE metrikák egy alternatívát nyújtanak

az eddigi metrikákhoz képest. A hiba-előrejelzési változást az eredeti és a PE metrikákkal kibővített adathalmazon végeztük el. A tanítás és kiértékelés kétféleképpen történt. Az első esetben, 33 kijelölt projektet egyenként minden más projekttel egy-egy tanítás-kiértékelés sorozatnak vetettük alá. A kapott súlyozott F-mértékek változását figyelve állapítottuk meg, hogy javul-e a hiba előrejelzési képesség. Az 1089 esetből 123 esetben a súlyozott F-mérték változása nagyobb vagy egyenlő 0,05-nél, és 107 esetben a változás kisebb volt -0,05-nél. A másik esetben ugyanazon projekt különböző verzióin történt a tanítás és kiértékelés, ezzel modellezve azt, hogy egy szoftver fejlesztés során folyamatosan használják a régebbi rendszeren szerzett metrika adatokat a hiba előrejelzéshez. Mindkét kísérlet alapján arra a következtetésre jutottunk, hogy a PE metrikák hozzáadása hibaadatbázisokhoz segíthet a hibák előrejelzésében. A második tézispont kettő fő eredménye a PE metrikák megalkotása, azoknak a kiértékelése és a hiba-előrejelzési képességének a vizsgálata.

A szerző hozzájárulásai

A szerző tervezte meg az eredeti Primitive Enthusiasm metrikát. Implementálta ennek a metrikának a kiszámítását egy Java statikus elemezőben. A szerző részt vett a rendszerek kiválasztásában és az eredeti Primitive Enthusiasm metrika kiértékelésében ezen a rendszereken. Megtervezte a kísérletet annak megállapítására, hogy a Java wrapper osztályok hogyan befolyásolják a metrika eredményeit. A Primitive Enthusiasm metrika alapján a szerző formalizálta az LPE, GPE és HPE metrikákat. Egy meglévő hiba adathalmaz felhasználásával korrelációkat mutatott ki az új metrikák és más meglévő metrikák között. A hiba előrejelzési képességek vizsgálatához kiválasztotta a vizsgálandó rendszereket. Kidolgozta, végrehajtotta és kiértékelte a kétféle módszert a hiba előrejelzéshez, PE metrikák hozzáadásával.

Az második tézisponthoz a következő publikációk kapcsolódnak:

[26] **Péter Gál** and Edit Pengő. Primitive Enthusiasm: A Road to Primitive Obsession. In *The 11th Conference of PhD Students in Computer Science (CSCS 2018)*, Volume of short papers, pages 134-137, Szeged, Magyarország, June 2018.

[57] Edit Pengő and **Péter Gál**. Grasping Primitive Enthusiasm - Approaching Primitive Obsession in Steps. In *Proceedings of the 13th International Conference on Software Technologies (ICSOFT 2018)*, pages 389-396, Porto, Portugal, July 2018. SciTePress.

[22] **Péter Gál**. Bug Prediction Capability of Primitive Enthusiasm Metrics. In *21st International Conference on Computational Science and Its Applications (ICCSA 2021)*, Proceedings, Part VII, volume 12955 of Lecture Notes in Computer Science (LNCS), pages 246-262, Cagliari, Italy, September 2021. Springer.

Acknowledgments

Firstly, I would like to thank Dr. Ákos Kiss, my supervisor, for his professional help and unique opinions during my PhD studies. Secondly, to co-author Csaba Bátori, who also shared the bizarre endeavour known as the A+ language, and to my other co-author and PhD study partner, Edit Pengő. Finally, I would also like to express my gratitude for the continuous support of my mother, my grandmother and my whole family

- Part ot this thesis was supported by grant NKFIH-1279-2/2020 of the Ministry for Innovation and Technology, Hungary.
- This thesis was partially supported by the EU-funded Hungarian national grant GINOP-2.3.2-15-2016-00037 titled "Internet of Living Things" and by the project "Integrated program for training new generation of scientists in the fields of computer science", no EFOP-3.6.3-VEKOP-16-2017-0002. Part of this work was supported by the European Union and co-funded by the European Social Fund.
- This thesis was partially supported by the EU-supported Hungarian national grant GINOP-2.3.2-15-2016-00037 and by grant NKFIH-1279-2/2020 of the Ministry for Innovation and Technology, Hungary.

Bibliography

[1] A+.NET implementation. https://github.com/elecro/aplusdotnet/ [Last accessed: 1 May 2022].

[2] Apache Commons Math. https://github.com/apache/commons-math. [Last accessed: 3 May 2022].

[3] Apache Log4j. https://github.com/apache/log4j. [Last accessed: 3 May 2022].

[4] Joda-Time. https://github.com/JodaOrg/joda-time. [Last accessed: 3 May 2022].

[5] *Mono*. https://www.mono-project.com/ [Last accessed 22 April 2022].

[6] Department of Software Engineering, University of Szeged. Open Static Analyser. https://github.com/sed-inf-u-szeged/OpenStaticAnalyzer. [Last accessed: 3 May 2022].

[7] Fischer Black and Myron Scholes. The pricing of options and corporate liabilities. *Journal of Political Economy*, 81(3):637–654, May–June 1973.

[8] Joe Blaze. VisualAPL documentation. http://forum.apl2000.com/viewtopic.php?f=4&t=626&p=2383&hilit=documentation#p2383 [Last accessed: 23 April 2022].

[9] Joe Blaze. Prior and current versions of VisualAPL. APL2000 Developer Network Forum, http://forum.apl2000.com/viewtopic.php?t=447 [Last accessed: 22 April 2022], April 2009.

[10] Barry Boehm and Victor R. Basili. Software defect reduction top 10 list. *Computer*, 34(1):135–137, January 2001.

[11] Robert G. Brown. Object oriented apl: An introduction and overview. In *Proceedings of the International Conference on APL-Berlin-2000 Conference*, APL '00, pages 47–54, New York, NY, USA, 2000. ACM.

[12] Leigh Clayton, Mark D. Eklof, and Eugene McDonnell. *ISO/IEC 13751:2000(E): Programming Language APL, Extended*. Internation Standards Organization, June 2000.

[13] Michael Coughlan. *Beginning COBOL for Programmers*. Apress, USA, 1st edition, 2014.

[14] M. D'Ambros, M. Lanza, and R. Robbes. An extensive comparison of bug prediction approaches. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 31–41, 2010.

[15] Dyalog Ltd. Dyalog APL. `http://www.dyalog.com/` [Last accessed: 26 March 2022].

[16] ECMA International. *ECMA-334 - C# Language Specification*. 5th edition, December 2017. `https://www.ecma-international.org/wp-content/uploads/ECMA-334_5th_edition_december_2017.pdf` [Last accessed 14 May 2022].

[17] N.L. Ensmenger. *The Computer Boys Take Over: Computers, Programmers, and the Politics of Technical Expertise*. History of Computing. MIT Press, 2012.

[18] Rudolf Ferenc, Árpád Beszédes, Mikko Tarkiainen, and Tibor Gyimóthy. Columbus – reverse engineering tool and schema for C++. In *Proceedings of the 18th International Conference on Software Maintenance (ICSM 2002)*, pages 172–181, Montréal, Canada, 2002. IEEE.

[19] Rudolf Ferenc, Zoltán Tóth, Gergely Ladányi, István Siket, and Tibor Gyimóthy. A public unified bug dataset for java and its assessment regarding metrics and bug prediction. *Software Quality Journal*, Jun 2020.

[20] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

[21] Eibe Frank, Mark A. Hall, and Ian H. Witten. *Online Appendix for "Data Mining: Practical Machine Learning Tools and Techniques"*. Morgan Kaufmann, Fourth Edition, 2016.

[22] Péter Gál. Bug prediction capability of primitive enthusiasm metrics. In *Computational Science and Its Applications – ICCSA 2021: 21st International Conference, Cagliari, Italy, September 13–16, 2021, Proceedings, Part VII*, pages 246–262, Berlin, Heidelberg, 2021. Springer-Verlag.

[23] Péter Gál, Csaba Bátori, and Ákos Kiss. Extending A+ with object-oriented elements: A case study for A+.NET. In *Computational Science and Its Applications – ICCSA 2021: 21st International Conference, Cagliari, Italy, September 13–16, 2021, Proceedings, Part IX*, page 141–153. Springer, 2021.

[24] Péter Gál and Ákos Kiss. Implementation of an A+ interpreter for .NET. In *Proceedings of the 7th International Conference on Software Paradigm Trends (ICSOFT 2012)*, pages 297–302, Rome, Italy, July 24–27, 2012. SciTePress.

[25] Péter Gál and Ákos Kiss. A comparison of maintainability metrics of two A+ interpreters. In *Proceedings of the 8th International Joint Conference on Software Technologies - ICSOFT-EA, (ICSOFT 2013)*, pages 292–297. INSTICC, SciTePress, 2013.

[26] Péter Gál and Edit Pengő. Primitive enthusiasm: A road to primitive obsession. In *The 11h Conference of PhD Students in Computer Science*, pages 134–137. University of Szeged, 2018.

[27] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., USA, 1995.

[28] Jean Jacques Girardot and Sega Sako. An object oriented extension to apl. In *Proceedings of the International Conference on APL: APL in Transition*, APL '87, pages 128–137, New York, NY, USA, 1987. ACM.

[29] Aakanshi Gupta, Bharti Suri, and Sanjay Misra. A systematic literature review: Code bad smells in java source code. In *Computational Science and Its Applications – ICCSA 2017*, pages 665–682, Cham, 2017. Springer.

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

[31] Tracy Hall, Min Zhang, David Bowes, and Yi Sun. Some code smells have a significant but small effect on faults. *ACM Transactions on Software Engineering and Methodology*, 23(4), September 2014.

[32] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., USA, 1977.

[33] Mark Hammond. *Python for .NET: Lessons learned*. ActiveState Tool Corporation, November 2000. `https://web.archive.org/web/20060925092941/http://starship.python.net/crew/mhammond/dotnet/PythonForDotNetPaper.doc` [Last accessed: 29 April 2022].

[34] Jim Hugunin. Python and Java – the best of both worlds. In *Proceedings of the 6th International Python Conference*, pages 11–20, San Jose, CA, USA, October 1997.

[35] Jim Hugunin. IronPython: A fast Python implementation for .NET and Mono. In *PyCON 2004*, Washington, DC, USA, March 2004.

[36] ISO 1989:1978, Programming languages - COBOL. Standard, February 1978.

[37] ISO 1989:1985, Programming languages - COBOL. Standard, December 1985.

[38] ISO/IEC 1989:2002, Information technology - Programming languages - COBOL. Standard, December 2002.

[39] ISO/IEC 1989:2014, Information technology - Programming languages, their environments and system software interfaces - Programming language COBOL. Standard, June 2014.

[40] R. Jayanthi and M. Florence. Software defect prediction techniques using metrics based on neural network classifier. *Cluster Computing*, 22:77–88, 2019.

[41] Marian Jureczko and Lech Madeyski. Towards identifying software project clusters with regard to defect prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, PROMISE '10. ACM, 2010.

[42] Morten Kromberg, Jonathan Manktelow, and John Scholes. APL# - an APL for Microsoft.Net. In *Conference USB stick of APL2010*, Berlin, Germany, September 2010.

[43] Morten J. Kromberg. Arrays of objects. In *Proceedings of the 2007 Symposium on Dynamic Languages*, DLS '07, page 20–28, New York, NY, USA, 2007. Association for Computing Machinery.

[44] P. DeMarco L. Bernardin, P. Chin et al. *Maple Programming Guide*. Maplesoft, 2011.

[45] Steven A. Lowe. kata-2-tinytypes. `https://github.com/stevenalowe/kata-2-tinytypes`. [Last accessed: 3 May 2022].

[46] Mika V. Mäntylä, Jari Vanhanen, and Casper Lassenius. A taxonomy and an initial empirical study of bad smells in code. In *Proceedings of the International Conference on Software Maintenance. ICSM*, pages 381–384. IEEE, 2003.

[47] Fabio Mascarenhas and Roberto Ierusalimschy. LuaInterface: Scripting the .NET CLR with Lua. *Journal of Universal Computer Science*, 10(7):892–909, July 2004.

[48] MathWorks. *Matlab Object-Oriented Programming*, 2021. `https://www.mathworks.com/help/pdf_doc/matlab/matlab_oop.pdf` [Last accessed: 15 June 2021].

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

[50] Tim Menzies, Zach Milton, Burak Turhan, Bojan Cukic, Yue Jiang, and Ayse Bener. Defect prediction from static code features: Current results, limitations, new approaches. *Automated Software Engineering*, 17:375–407, 2010.

[51] Microsoft. *Dynamic Language Runtime Overview*. `https://docs.microsoft.com/en-us/dotnet/framework/reflection-and-codedom/dynamic-language-runtime-overview` [Last accessed: 28 April 2022].

[52] Leon Moonen and Aiko Yamashita. Do code smells reflect important maintainability aspects? In *Proceedings of the 2012 IEEE International Conference on Software Maintenance. ICSM*, pages 306–315. IEEE, 2012.

[53] Morgan Stanley. *A+ Language Reference*, 1995–2008. https://github.com/PlanetAPL/a-plus/blob/master/docs/language_reference.pdf [Last accessed: 15 August 2022].

[54] David Mosberger. *The libunwind project.* http://www.nongnu.org/libunwind/index.html [Last accessed: 15 April 2022].

[55] Paul Oman and Jack Hagemeister. Metrics for assessing a software system's maintainability. In *Proceedings of the 1992 IEEE Conference on Software Maintenance*, pages 337–344, Orlando, FL, USA, 1992. IEEE.

[56] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages.* Pragmatic Programmers. Pragmatic Bookshelf, first edition, May 2007.

[57] Edit Pengő. and Péter Gál. Grasping primitive enthusiasm - approaching primitive obsession in steps. In *Proceedings of the 13th International Conference on Software Technologies. ICSOFT*, pages 389–396. INSTICC, SciTePress, 2018.

[58] Ricardo Pérez-Castillo, Ignacio García-Rodríguez de Guzmán, and Mario Piattini. Mimos, system model-driven migration project. In *2013 17th European Conference on Software Maintenance and Reengineering*, pages 445–448, 2013.

[59] Andy H. Register. *A Guide to MATLAB Object-Oriented Programming.* Scitech Pub Inc, 2007.

[60] Reuters. Cobol blues. Technical report. https://fingfx.thomsonreuters.com/gfx/rngs/USA-BANKS-COBOL/010040KH18J/index.html [Last accessed: 2 May 2022].

[61] Naveen Roperia. Jsmell: A bad smell detection tool for java systems. Master's thesis, Maharishi Dayanand University, 2009.

[62] Harry M Sneed. Estimating the costs of a reengineering project. In *Proceedings of the 12th Working Conference on Reverse Engineering (WCRE 2005)*, pages 111–119, Pittsburgh, PA, USA, 2005. IEEE.

[63] Harry M Sneed. Migrating from COBOL to Java. In *Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM 2010)*, pages 1–7, Timisoara, Romania, 2010. IEEE.

[64] Social Security Administration Office of the Inspector General. The Social Security Administration's Software Modernization and Use of Common Business Oriented Language, May 2012. `https://oig-files.ssa.gov/audits/summary/Summary%2011132.pdf` [Last accessed: 2 May 2022].

[65] Sun Microsystems. *JSR-223: Scripting for the Java Platform*, December 2006.

[66] Sun Microsystems. *JSR-292: Supporting Dynamically Typed Languages on the Java Platform*, July 2011.

[67] Zoltán Tóth, Péter Gyimesi, and Rudolf Ferenc. A public bug database of github projects and its application in bug prediction. In *Computational Science and Its Applications – ICCSA 2016*, volume 9789, pages 625–638. Springer, 2016.

[68] R. S. Wahono. A systematic literature review of software defect prediction: Research trends, datasets, methods and frameworks. *Journal of Software Engineering*, 1:1–16, 2015.

[69] Robert Wingate. *COBOL Basic Training Using VSAM, IMS, DB2 and CICS*. 1st edition, 2020.

[70] Aiko Yamashita and Leon Moonen. To what extent can maintenance problems be predicted by code smell detection? - An empirical study. *Information and Software Technology*, 55(12):2223–2242, 2013.

[71] Zhifeng Yu and V. Rajlich. Hidden dependencies in program comprehension and change propagation. In *Proceedings 9th International Workshop on Program Comprehension. IWPC 2001*, pages 293–299, 2001.

[72] Min Zhang, Tracy Hall, and Nathan Baddoo. Code bad smells: A review of current knowledge. *Journal of Software Maintenance and Evolution*, 23(3):179–202, April 2011.

[73] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, PROMISE '07, page 9. IEEE, 2007.