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

Summary of the 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

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 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. These topics are tightly related to software maintenance and quality aspects.

The first part focuses on the A+ language, creating a clean-room implementation for the language on top of .NET. This implementation also provides insight into a few minor challenges faced when software is re-implemented to be compatible with the original. The runtime and source code metric comparisons of the original interpreter and the .NET variant provide insight into both systems. In order to ease the connection between the .NET and A+ worlds, a new language extension was developed that allows accessing various object-oriented elements for A+.

In the second part, a newly created set of metrics are presented to handle a part of the Primitive Obsession bad smell. The new metric is the Primitive Enthusiasm and its variants. The idea was to highlight methods with comparatively more primitively typed arguments than other methods in the same class. A derivation of this metric compares the method to all other methods in the system and a variant that encapsulates both class local and system level information.

Between these two main thesis points, the author states four main results as listed below:

1. A+.NET Implementation and Comparison of Runtimes
2. A+.NET Language Extension
3. Definition and Evaluation of Primitive Enthusiasm Metrics
4. The Bug Prediction Capabilities of the Primitive Enthusiasm Metrics

In the rest of the booklet, we summarize the results for each thesis point.

I. Experiments with the A+ Programming Language

The first thesis point discusses the A+.NET implementation and comparison of runtimes with the addition of the object-oriented language extension. These are separated into two main results that are presented in the following two sections.

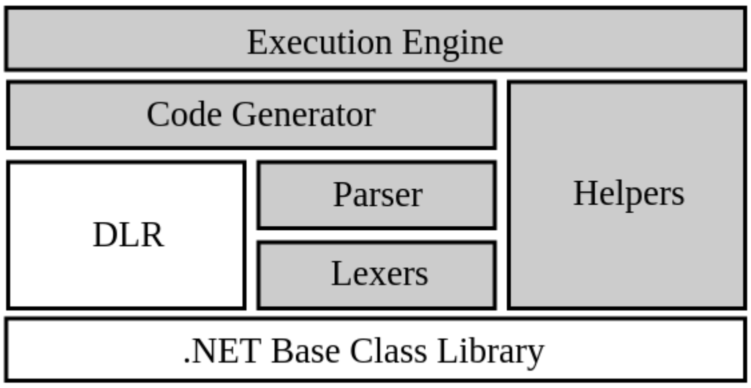
1. A+.NET Implementation and Comparison of Runtimes

There are only two official sources of information available on the syntax of A+: the Language Reference [12], 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. However, to ease the development of the .NET runtime, the A+ grammar had to be formalized.

In order to do this, the grammar of 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 [13] parser generator framework to generate the required C# classes. Interestingly, there were differences between the textual description for a given element in the language reference and its allowed usage in the interpreter. In most cases, we tried to follow the original interpreter's behaviour to not break any existing A+ codes that were built on such edge cases.

The high-level system design of the A+.NET runtime is depicted in Figure 1, highlighting the main components 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 that was implemented by us.

Figure 1: *Components of the A+ .NET runtime*



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.

The .NET-based A+ execution engine can be used in two ways. Since DLR provides a command line hosting API, it is easy to implement a Read-Eval-Loop interface that mimics the behaviour of the reference interpreter implementation. However, the biggest advantage of the .NET-based implementation is that it is just as simple to embed the runtime into other .NET applications. Moreover, it is also possible to expose .NET methods and values into the A+.NET runtime.

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 to the engine's runtime scope. There are five types of values that can be registered: integers, doubles, 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. Functions require a bit of special handling because each method that the developer wants to add to the scope must adhere to some rules :

1. The method must be a static method.

2. The return type must be `AType`, which is the base interface type for all types in the runtime.

3. The first argument must be an `Aplus` type, which contains the runtime environment information and can be accessed by the method.

4. Any other arguments must be of `AType` type, and – most importantly – they must be in reverse order.

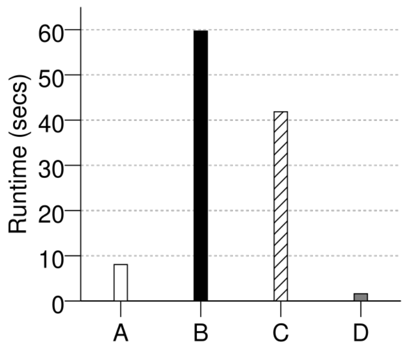
The reverse order is required because the A+ language evaluates function arguments from right to left while C# does not. Thus, in A+.NET runtime, we perform a trick and require all methods to have a reverse order of arguments. So for an A+ function that accepts two parameters, the second argument of 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#.

After the `AType` was created, it can be added to the runtime scope via the set variable method call programmatically. In the case of functions, two annotations were developed to make it possible to automatically load functions into the runtime. These annotations are the `AplusContext` and `AplusContextFunction`. 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+. These classes and functions are looked up when the `$load` system command is used with a context name. Under the hood, the load function will traverse the DLL files currently loaded and will search for the `AplusContext` that has the given context name and add each method annotated with `AplusContextFunction` to the current scope.

When the required functions or values are added to the A+.NET runtime's scope it is possible to use them just like any other variable/function in A+.

As the .NET version of A+ is now available, a comparison can be made to see how the two engines perform. This was done in multiple ways. We compared the execution speed and source code metrics focusing on maintainability. Although our primary goal for the initial implementation of the .NET-based runtime was to make its observable behaviour equivalent to the reference implementation as possible and, thus, we did not focus especially on optimisations, we still wanted to get preliminary results on its runtime performance. In order to do this, we extracted a code fragment from a real-life code base and extended it with some code performing execution time measurement. The extracted A+ fragment was a simple URL encoding algorithm. Four experiments were done that are depicted in Figure 2.

Figure 2: *Execution times of the 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.*



Column A is the execution time of the A+ script with the reference implementation. Column B is the run time of the A+.NET execution. According to the measurements, the reference implementation is about seven times faster than the .NET port at the moment.

In the case of column C, the string join method in the script was replaced by the equivalent .NET variant. This led to nearly 30% speedup in execution time. Finally, in column D, the whole URL encoding code part was replaced by its .NET counterpart. 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.

In terms of source code metrics, we determined functionally equivalent parts between the two runtimes by using a common set of A+ test scripts to calculate source code function level coverage. The two function sets, one in each system, are of comparable size and of equivalent functionality and was used for further investigation into the maintainability of the two systems. We used the Columbus toolchain [3] to analyse the sources, and as a result, we got two size metrics – LOC [1] and NOS [2] – and two complexity metrics – McCC [3] and NLE [4] – for each function.

[1]LOC: executed lines of code

[2]NOS: number of statements

[3]McCC: McCabe's cyclomatic complexity

[4]NLE: nesting level

Table 1: *Maintainability-related metrics measured for the A+ reference interpreter and A+.NET*

Metric	Reference Interpreter min / avg / max			A+.NET min / avg / max		
	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

As depicted in Table 1, the averages of NOS, McCC [10], and NLE, and the maximums of all metrics show that the size and the complexity of the functions in the reference implementation are larger than in A+.NET.

We also experimented with 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.

With the creation of the A+.NET runtime, the lifetime of existing A+ applications can be extended.

2. A+.NET Language Extension

Exposing methods for A+ scripts requires a bit of boilerplate code, as in order to expose a .NET method into the runtime, writing wrapper functions were required. 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. Based on this, the introduction of a way to represent objects in the runtime is required. In the case of A+, this essentially means that a new type should be added to the language. This was named `AObject` internally.

Furthermore, we identified 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 `SelectMemeber` operation provides the means to read variables and properties and to access methods. In most cases, a name lookup on the input class or instance can find the required member. In the case of A+, there is already a notation to do a similar lookup, but it is done on a context. In order to integrate seamlessly into the syntax of the A+, the existing but hitherto unused $\ominus$ symbol was chosen.

- Modifying variables and properties: the `SetMember` operation makes it possible to assign new values to properties and values. Similarly to the previous case, a name lookup is performed to find the member whose values should be changed. For this operation, however, there is no need to introduce a new symbol. The same $\ominus$ symbol can be used, and during the parsing, it is possible to detect if this symbol is on the left side of an assignment or not. If it is on the left side, then it is a `SetMember` operation otherwise it is a `SelectMember` operation.

- 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. Fortunately, A+ already has a syntax for indexing so we can leverage the already existing indexing operation, and if a `AObject` is found then we can perform the indexing operation on the target object.

- Type casting: A new $\diamond$ symbol is introduced into the language, providing the means to perform the .NET type casting functionality in the runtime from the A+ code.

Using these and the power of the .NET runtime, it is possible to provide general functions for A+.NET that can perform the required lookups and runtime code generations to avoid 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 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 for invocation. 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. In such cases, a type matching should be performed to see which is the most suitable method for a given method call.

For the A+.NET, a type distance vector based type matching algorithm was developed. 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 runtime that the number of parameters is incorrect. Second, as the number of arguments is now correct, we can calculate the type distances. 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 [2, § 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

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 or equal to the corresponding element in the other method's vector, but at least one element is strictly smaller than its corresponding element.

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:

[7] **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.

[8] **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.

[6] **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 or at least a part of it. This was achieved by the creation of the Primitive Enthusiasm metrics and was shown that it can be used to improve bug predictions. The two main results of this thesis point is presented in the following sections.

3. Definition and Evaluation of Primitive Enthusiasm Metrics

Primitive Obsession is a type of code smell that has lacked the attention of the research community. Since defining Primitive Obsession is challenging with a single formula, the author chose to deconstruct the bad smell into a smaller part.

A part of this deconstruction is the new Primitive Enthusiasm (PE) metric. The idea is to quantify the primitive typed arguments in a function in such a way that the methods can be compared to other methods in the same system. This 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 in the same system. For the base of the PE metric, the Formula 1 was created by us, which describes how the primitive-typed parameters are collected for a given M_i method.

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

For 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.

Using this *Primitives* function, we created three metrics. These are the Local Primitive Enthusiasm (LPE), Global Primitive Enthusiasm (GPE), and Hot Primitive Enthusiasm (HPE) metrics and are presented in the Formulae 2, 3, and 4 respectively.

$$LPE(M_c) := \frac{\sum_{i=1}^{N} |Primitives(M_i)|}{\sum_{i=1}^{N} | P_{M_i} |} < \frac{|Primitives(M_c)|}{| P_{M_c} |} \tag{2}$$

$$GPE(M_c) := \frac{\sum_{i=1}^{|G|} |Primitives(G_i)|}{\sum_{i=1}^{|G|} \mid P_{G_i} \mid} < \frac{|Primitives(M_c)|}{\mid P_{M_c} \mid} \tag{3}$$

$$HPE(M_c) := LPE(M_c) \land GPE(M_c) \tag{4}$$

In case of LPE and GPE the right hand side of the inequality is calculated for the currently investigated method, whilst the left hand side gives the baseline. For LPE this is the percentage of how many parameters of the current class are of primitive types. In similar fashion, in the GPE formulae the left-hand side now describes the average number of primitive-typed arguments in the whole system. The combination of these metrics is the HPE metric. By definition these metrics are function level metrics but can be aggregated to be a class level metric.

The metric calculation was implemented in the Open Static Analyzer [1] for JAVA. For evaluation three often used projects were selected: Joda-Time version 2.9.9 [5], Apache Log4j [6], and Apache Commons Math version 3.6.1 [7]. We experimented with the inclusion, or exclusion of Java wrapper classes to see how the reported methods and classes change. The results shown that there were only a few method difference by including or excluding these wrapper classes. With the addition of a method skipping strategy to exclude all methods that only have a single parameter, the number of suspicious methods were reduced. With this experiment the best result was achieved when the HPE metric with wrapper classes were taken into account.

4. The Bug Prediction Capabilities of the Primitive Enthusiasm Metrics

In order to evaluate the bug prediction capabilities of the PE metrics and already existing bug dataset [4] was used that contained pre-calculated metrics. This dataset was extended with the PE metrics. However, the PE metrics are inherently method-based metrics. In order to resolve this minor incompatibility, the PE metrics were aggregated by class.

An interesting experiment was to see what kind of correlation is there between the already existing metrics and the new PE metrics. Not surprisingly, there were positive correlations between PE and older metrics that are based on the number of parameters or the number of methods as their base and the highest correlation was found between the NLM and LPE metrics with a value of 0.58. The correlation between a few metrics that are calculated from the number of methods was also investigated. These were the NLM[8], TNLM[9], NLPM[10],

[5]https://github.com/JodaOrg/joda-time

[6]https://github.com/apache/log4j

[7]https://github.com/apache/commons-math

[8]NLM: Number of Local Methods

[9]TNLM: Total Number of Local Methods

[10]NLPM: Number of Local Public Methods

TNLPM[11], RFC[12], and WMC[13] metrics. Between these metrics – excluding the WMC – the correlation values were above 0.70 in every case, and in some instances, it is even above 0.90. The other set of interesting correlations with PE metrics are the ones related to lines of code: LOC[14], LLOC[15], CLOC[16], DLOC[17], TLOC[18], TLLOC[19], TCLOC[20], TNOS[21], NOS[22]. 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. Correlation between a selected set of line count related metrics is similarly strong as in the number of methods based metrics case.

The bug prediction capabilities were tested by using the original and extended datasets from which were 33 selected systems. We trained and evaluated them to see the weighted F-measure changes between them. Furthermore, this training and evaluation were done in two ways. First, a cross-project based evaluation was performed where each project was trained and compared to another project. Out of the 1089 cases, the weighted F-measure changes that are greater than or equal to 0.05 were observed in 123 cases, and in 107 cases, the changes are less than -0.05. Overall, there were more improvements than reductions.

In the second experiment, the training and evaluation were done across project versions. The worst results were provided by the Ant project and its various versions. In this case, there are a bit more than 10 cases where the addition of the PE metrics resulted in slightly negative values. However, overall the F-measure changes are small. The Velocity project gave one of the best results with the addition of the PE metrics. In almost every version combination, the addition of the new metrics improved the F-measure values. Based on these results, we can conclude 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

[11]TNLPM: Total Number of Local Public Methods
[12]RFC: Response set For Class
[13]WMC: Weighted Methods per Class
[14]LOC: Lines of Code
[15]LLOC: Logical Lines of Code
[16]CLOC: Comment Lines of Code
[17]DLOC: Documentation Lines of Code
[18]TLOC: Total Lines of Code
[19]TLLOC: Total Logical Lines of Code
[20]TCLOC: Total Comment Lines of Code
[21]TNOS: Total Number of Statements
[22]NOS: Number of Statements

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:

[9] **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.

[14] 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.

[5] **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.

Summary

In this thesis, two main points were discussed. Both of these topics are related to software maintenance. The first is in connection with the A+ language, and the second is with the Primitive Obsession bad smell.

In the first thesis point, the research focused on the A+.NET clean-room implementation of the A+ language. During this work, we created a formalized grammar for A+ and implemented the core components of the runtime. Using the reference implementation and the .NET variant, the run time and source code metric evaluations were done. The reference implementation is faster by default, but its maintainability aspects are questionable. Whereas the .NET variant is better in terms of source code metrics but slower at the moment. Still, the ability of the new version is that other .NET developers can easily extend it to expose various components into the runtime. Furthermore, the ability to run A+ scripts not only on Unix-like platforms can extend the lifetime of critical A+ applications. In order to further ease the usage of .NET classes, the A+ language was extended with new operations which bring object-oriented notations into the language. These notations enable access to class members, change property values, and resolve method ambiguities without changing existing language rules. However, even with this, the A+ language itself is still not an object-oriented language. The two main results of the first thesis point are the new A+.NET interpreter and the object-oriented language extension.

The second thesis point deals with the Primitive Obsession bad smell. The original idea that “too many primitives are used” is a bit vague. In order to resolve this, we created a concrete calculable metric to capture part of this bad smell, and it is called Primitive Enthusiasm (PE), which we later renamed to Local Primitive Enthusiasm (LPE). This metric captures the number of primitively typed parameters for a method in a given class and compares it to the averages of the same class. We implemented the metric calculation in the Open Static Analyzer [1] framework for the Java language and was evaluated on three selected systems. During the evaluation, the effect of including or excluding Java wrapper classes was investigated, resulting in a negligible difference. Based on this metric, we created two other variants: Global Primitive Enthusiasm (GPE) and Hot Primitive Enthusiasm (HPE). The GPE variant changes the metric’s formulae in a way that it now compares the current function under investigation to the whole system. HPE then incorporates both LPE and GPE results into a single result. We spent further work to see how the bug prediction capabilities changes when the new PE metrics are added. We used an already existing bug dataset [4] that contained pre-calculated metrics for this. We extended this dataset with the PE metrics. However, the PE metrics are inherently method-level metrics, but this was resolved by aggregating the PE metrics by class. An interesting experiment was to see the correlation between the already existing metrics and the new PE metrics. There were a few connections between the metrics, but overall it can be concluded that PE metrics can provide benefits. We tested the bug prediction capabilities by using the original and extended datasets. We done this training and evaluation in two ways. A cross-project based version, where we trained on each project and evaluated it with all others, and a project-version, where we used the different versions of the same system as training data and evaluated them on the other versions. Based on the results provided by these experiments, we can conclude that adding the PE metrics to aid in bug prediction is a viable option. In this second thesis point the main results were the definition and evaluation Primitive Enthusiasm metrics and

the investigation of the bug prediction capabilities of the same metrics.

Finally, Table 2 summarizes the relation between the thesis points and the corresponding publications.

Table 2: *Correspondence between the main thesis points and the corresponding publications*

	[7]	[8]	[6]	[9]	[14]	[5]
I.	●	●	●			
II.				●	●	●

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.

References

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

[2] 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].

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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].

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

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

[14] 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.

Összefoglaló

A disszertáció kettő tézispontra bontható. Az első rész az A+ nyelvvel végzett kísérleteket taglalja, míg a második rész a Primitive Obsession gyanús kóddal foglalkozik.

Az első tézispontban a kutatás egy A+ nyelv interpreter implementálását tárgyalja amely a .NET keretrendszerre épül. Ezen munka során formalizáltuk az A+ nyelvtan, mivel ezidáig az A+-nak nem volt formálisan leírt nyelvtana. A nyelv dinamikusságának a támogatásáért a Dynamic Language Runtime [11] volt használva. Az eredeti interpretert és a .NET változatot felhasználva futás idejű és forráskód metrika alapú kiértékelés végeztünk. Az eredmények alapján a referencia implementáció ugyan gyorsabb, de karbantarthatósági szempontból rosszabbul teljesít a .NET verzióhoz képest. Ezzel szemben, a .NET lassabb, de jobb forráskód metrika adatokkal rendelkezik. A .NET interpreter fontos tulajdonsága, hogy egyszerű bővíteni akár meglévő C# eljárásokkal, melyeket zökkenőmentesen lehet A+ programokban használni. Ezzel akár a lassú kódrészleteket ki lehet cserélni .NET megfelelőjükre. Továbbá egy másik fontos aspektus, hogy a .NET változat kitolhatja a kritikus A+ alkalmazások életidejét, hiszen nem csak Unix-szerű rendszereken működik. A .NET-ben található különféle osztályok használatának megkönnyítése érdekében az A+ nyelvet kiterjesztettük új elemekkel. Ezek az új eljárások, annotációk lehetővé teszik, hogy objektumorientált jellegű jelölésrendszerrel osztályok adattagjait érhessük el, extra csomagoló eljárások készítése nélkül. Fontos szempont volt a kiterjesztés során, hogy a meglévő nyelvi szabályokat ne sértsük meg. Azonban, ezzel maga az A+ nyelv még nem lett objektumorientált. Az új A+.NET értelmező és az objektumorientált nyelvi kiterjesztés az első tézispont két fő eredménye.

A második tézispont a Primitive Obsession gyanús kóddal foglalkozik. Az eredeti gondolat, miszerint "túl sok primitív adattípus van használva" nem jól behatárolható. Ezért egy konkrétan kiszámítható metrikát hoztunk létre ennek a gyanús kódnak a mérésére. Ezt Primitive Enthusiasm (PE)-nak neveztük el, amit később átneveztünk Local Primitive Enthusiasm (LPE)-ra. A metrika egy adott eljárás primitív típusú paramétereinek az arányát hasonlítja össze az aktuálisan vizsgált osztályban található primitív paramétereknek az arányával. A metrika számítását beleintegráltuk az Open Static Analyzer [1] keretrendszerbe és három Java alapú rendszeren kiértékeltük. A kiértékelés során megvizsgáltuk, hogy a Java csomagoló osztályok bevonása vagy kizárása milyen hatással van a detektált eljárások mennyiségére. Az eredmények alapján az eltérés elhanyagolható. Az eredeti 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. Ezután megvizsgáltuk, hogy az új metrikákat hozzáadva egy meglévő hiba adatbázishoz [4] hogyan változnak a hiba-előrejelzési lehetőségek. A meglévő adatbázis többféle osztály szintű metrikát tartalmaz, azonban a a PE metrikák a definíciójukból eredve eljárás szintű metrikák. Viszont a PE értékek összegzésével osztály szintű metrikát kaphatunk. 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, egy a projektet minden más projekttel való tanítás-kiértékelés sorozatnak vetettük alá. Míg a másik esetben ugyanazon projekt különböző verzióin történt a tanítás és kiértékelés. 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.

Témavezetői Nyilatkozat

Alulírott Dr. Kiss Ákos, mint **Gál Péter** doktorjelölt témavezetője kijelentem, hogy a jelölt "**Software Maintenance Experiments with the A+ Programming Language and the Primitive Obsession Bad Smell**" című értekezésében felhasznált eredmények a jelölt saját hozzájárulását tükrözik.

Szeged, 2022. augusztus 23.

[illegible]

Dr. Kiss Ákos
Témavezető

Társszerzői Nyilatkozat

Kijelentem, hogy ismerem **Gál Péter** PhD fokozatra pályázó "**Software Maintenance Experiments with the A+ Programming Language and the Primitive Obsession Bad Smell**" című disszertációját. A disszertációban szereplő és a

1. **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.

2. **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.

3. **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.

cikkekben publikált eredményekre vonatkozóan kijelentem, hogy az alábbi eredményekben a pályázó hozzájárulása volt meghatározó:

- Az A+ nyelvhez korábban nem létező környezetfüggetlen nyelvtan formalizálása. [1] (4. fejezet)
- Az A+.NET futtató környezet tervezése és fejlesztése. [1] (4. fejezet)
- Az eredeti A+ interpreter és az A+.NET összehasonlítása sebesség és szoftver karbantarthatósági metrikák alapján. [2] (5. fejezet)
- Az A+ nyelvhez a külső objektumok kezelését lehetővé tevő nyelvi elemek tervezése és formalizálása. [3] (6. fejezet)
- Az objektum orientált kiterjesztéshez használt típusegyezést kereső vektor alapú megoldás kidolgozása. [3] (6. fejezet)

Szeged, 2022. augusztus 23.

[illegible]

Dr. Kiss Ákos
Társszerző és Témavezető

Társszerzői Nyilatkozat

Kijelentem, hogy ismerem **Gál Péter** PhD fokozatra pályázó "**Software Maintenance Experiments with the A+ Programming Language and the Primitive Obsession Bad Smell**" című disszertációját. A disszertációban szereplő és a

1. **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.

cikkben publikált eredményekre vonatkozóan kijelentem, hogy az alábbi eredményekben a pályázó hozzájárulása volt meghatározó:

- Az A+ nyelvhez a külső objektumok kezelését lehetővé tevő nyelvi elemek tervezése és formalizálása. [1] (6. fejezet)
- Az objektum orientált kiterjesztéshez használt típus egyezést kereső vektor alapú megoldás kidolgozása. [1] (6. fejezet)

A következő eredményekben az én hozzájárulásom volt meghatározó:

- Az A+.NET-hez a formalizált nyelvi kiterjesztés implementálása és tesztelése. [1] (6. fejezet)
- A vektor alapú típus egyezést vizsgáló algoritmus implementálása. [1] (6. fejezet)

Szeged, 2022. május 4.

[illegible]

Bátori Csaba
Társszerző

Társszerzői Nyilatkozat

Kijelentem, hogy ismerem **Gál Péter** PhD fokozatra pályázó "**Software Maintenance Experiments with the A+ Programming Language and the Primitive Obsession Bad Smell**" című disszertációját. A disszertációban szereplő és a

1. 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.
2. **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.

cikkekben publikált eredményekre vonatkozóan kijelentem, hogy a következő eredményekhez való hozzájárulásunk oszthatatlan:

- A statikus elemzéshez használt Java rendszerek kiválasztása. [1, 2] (11. fejezet)
- Az elemzési eredmények kiértékelése, statisztikák készítése. [1, 2] (11. fejezet)

A következő eredményekben a pályázó hozzájárulása volt meghatározó:

- A Primitive Enthusiasm objektum orientált forráskód metrika kidolgozása. [1, 2] (10. fejezet)
- A Primitive Enthusiasm metrikát számító statikus elemző komponens implementálása. [1, 2] (11. fejezet)
- A Java csomagoló osztályokkal való kísérlet kidolgozása. [2] (11. fejezet)
- Az LPE, GPE, HPE metrikák elméleti kidolgozása és formalizálása. [2] (10. fejezet)

A következő eredményekben az én hozzájárulásom volt meghatározó:

- A code smell-ekkel, statikus elemzéssel kapcsolatos irodalom feldolgozása. [1, 2]
- Az MPC, SFPU, SFP-SCU metrikák elméleti kidolgozása. [1]
- Az elemzés során használt eliminációs módszerek kidolgozása. [2]
- A továbbfejlesztett metrikákat számító statikus elemző komponens implementálása. [2]

Szeged, 2022. május 4.

[illegible]

Pengő Edit
Társszerző