Grasping Primitive Enthusiasm
Approaching Primitive Obsession in Steps

Edit Pengő and Péter Gál
Department of Software Engineering, University of Szeged, Dugonics ter 13, 6720, Szeged, Hungary

Keywords: Code Smells, Primitive Obsession, Primitive Enthusiasm, Static Analysis, Refactoring.

Abstract: Primitive Obsession is a type of a code smell that has lacked the attention of the research community. Although, as a code smell it can be a useful indicator of underlying design problems in the source code, there was only one previously presented automated detection method. In this paper, the Primitive Obsession is discussed and multiple variants for Primitive Enthusiasm is defined. Primitive Enthusiasm is a metric designed to highlight possible Primitive Obsession infected code parts. Additionally other supplemental metrics are presented to grasp more aspects of Primitive Obsession as well. The current implementation of the described metrics is for Java and the evaluation was done on three open-source Java systems.

1 INTRODUCTION

During the development the initial structure of the source code will degrade. It will be more difficult to identify the original design and understand the code, therefore later changes and bug fixes will have higher costs. Refactorings, i.e, modifications that improve different attributes (e.g., readability, complexity, or maintainability) without changing the external a functionallity (Fowler, 1999), should be applied regularly to preserve the quality of the code.

Several indicators are already known that suggest the need for refactoring in the code. The 22 code bad smells listed by Fowler and Beck (Fowler, 1999) can act as such indicators. Code smells are not actual coding erros, although they are the symptoms of deeper design problems that might cause trouble in the future. Thus, it is not surprising that there is a lot of discussion within the research community about code smells, their significance and impact on maintenance costs. Over the years, many new smell types were introduced besides the original 22 and various detection techniques were recommended for them to help developers identify and refactor problems in the code.

Some code smells received much less attention than the others. Primitive Obsession, that can be vaguely interpreted as the overuse of primitive data types, is one of them. Besides our previous work (Gál and Pengő, 2018), we have found only one work (Roperia, 2009) that suggests an automated detection approach, whilst most of the research papers consider this smell on the periphery. Therefore, we have decided to study Primitive Obsession and work out a novel detection technique for Java programs. Since its definition can be broadly interpreted multiple detection criteria were investigated and several metrics were defined based on them. The introduced metrics were implemented as parts of a Java static source code analyser framework. The authors evaluated them on three open-source Java systems, then manually validated and compared the results.

The rest of the paper is organised as follows. In Section 2, the related work is discussed. Section 3 introduces Primitive Obsession through an example and examines the difficulties in its definition. The proposed detection methods are presented in Section 4, whilst the results can be found in Section 5. Finally, the paper is concluded in Section 6.

2 RELATED WORK

Primitive Obsession previously lacked the interest of the research community. Zhang and his team analysed the state of the art knowledge about code smells in a systematic literature review (Zhang et al., 2011). They collected 319 papers from 2000 to 2009, and examined 39 in detail. In their first research question, they investigated which code smells attracted the most research attention. The results showed that Duplicated Code smell was discussed the most – in 21 papers out of 39 – whilst many code bad smells re-

Pengő, E. and Gál, P.
Grasping Primitive Enthusiasm - Approaching Primitive Obsession in Steps.
DOI: 10.5220/0006918803890396
In *Proceedings of the 13th International Conference on Software Technologies (ICSOFT 2018)*, pages 389-396
ISBN: 978-989-758-320-9

ceived very little attention. Primitive Obsession was among the unpopular smells having only 5 corresponding papers, although these papers examined all 22 of Fowler's bad smells. This indicates that Primitive Obsession was not studied individually in the investigated period. A more recent systematic literature review (Gupta et al., 2017) came to a similar conclusion. Gupta et al. examined 60 research papers between 1999 and 2016, and found that four of Fowler's bad smells – including Primitive Obsession – had no detection method in any of the papers. A study on five known tools which could detect code smells as well (Fontana et al., 2011) reported that none of them was capable of finding Primitive Obsession.

Only one occurrence was found about how to detect Primitive Obsession in the literature (Roperia, 2009). In his thesis, Roperia introduced an automated bad smell detection tool, named JSmell for Java programs. He proposed a detection technique for Primitive Obsession based on the number of primitive data types declared in a class. If the number of primitive data types was above than the average for the whole project and the class was not instantiated JSmell reports Primitive Obsession. Unfortunately, we were unable to get hold of JSmell, therefore no comparison or closer study was possible.

Mäntylä et al. carried out an empirical study on the subjective smell evaluations of developers (Mäntylä et al., 2004). They found that human factors (knowledge, work-experience, role in the project) have an impact on the identification of a code part as bad smell. Moreover, they compared the results of the subjective evaluations and metric based detection techniques for three smells, and found that the metrics and smell evaluations did not always correlate. One of the smells they used in the comparison was the Long Parameter List that had the largest correlation value. They also observed that the Long Parameter List smell mainly consisted of primitives, which could indicate a connection with the Primitive Obsession smell.

Many papers discuss how good indicators of maintenance problems are the code smells (Yamashita and Moonen, 2013; Moonen and Yamashita, 2012). Although they are not silver bullets for defect prediction code smells, they can still provide valuable insights on some important maintainability factors, especially when more of them are combined.

3 BACKGROUND

In most programming languages there are two categories of data types: primitive and complex. Primitive types are the most basic data types provided by a language, e.g. *boolean*, *integer*, *char* etc. Complex types like *classes* and *structs* are the composites of other existing data types. Complex types have the advantage that they usually also include some semantic knowledge about the data they represent. Instead of using scattered functions all over the source code they let the programmer encapsulate the operations on the data, which is an important principle of object-oriented programming. For example, it is a convenient and more readable solution to place three integers that represent a 3D point into a class instead of using them separately. However, in the pressure of deadlines programmers – especially novice ones – tend to neglect small, but necessary refactorings. Adding one or two new method parameters to an already long parameter list might seem a fast and effortless solution, although in the long run changes like this will decrease the maintainability of the source code.

3.1 Definition of Primitive Obsession

Primitive Obsession is the excessive use of primitive data types. The programmer does not create small objects for small tasks, instead s/he is obsessed with the use of primitive data types. The taxonomy of Mäntylä et al. classifies Primitive Obsession as a type of *bloater* smell (Mäntylä et al., 2003), yet it is not really a bloat, but a symptom for the existence of overgrown, chaotic code parts.

Figure 1 shows several examples of Primitive Obsession. The class constants `ENGINEER` and `SALESMAN` at Lines 3 and 4 can be considered as type codes. Type codes are a set of integer or string variables that usually have an understandable name, and they are employed to simulate types, e.g. different types of employees. Although they are widely used in projects, they are a kind of Primitive Obsession as they violate the object oriented paradigm and can cause *hidden dependencies* (Yu and Rajlich, 2001). Type codes can be removed by forming a class or, for example, with a *State* or *Strategy* pattern (Gamma et al., 1995).

At Line 6, the parameter list of the `work` function is described. Three of its parameters are integers, whilst the fourth is a string. Parameter lists like this are Primitive Obsession especially if they appear several times in the code. (Strictly speaking, strings are not primitive types, but logically it is practical to include them in the definition.) They can be refactored by introducing a parameter object. Lines 8 and 9 also confirm the need for refactoring, because value checks like this should usually be encapsulated. More examples are available in Steven A. Lowe's GitHub

```
1  class Employee
2  {
3    static const int ENGINEER = 1;
4    static const int SALESMAN = 2;
5
6    public void work(int from, int to
         , int numberOfBreaks, String
         task)
7    {
8      if(task == null
9          || task.length() == 0) {
10     /* ... */
11     }
12   }
13 }
```

Figure 1: Sample code containing three Primitive Obsessions.

project[1] that shows step-by-step how to clean up a heavily Primitive Obsession infected code.

3.2 Challenges in Definition

As the previous section highlights it, it is challenging to give Primitive Obsession an exact, quantifiable definition. It has not only many various aspects, but every program is unique with its own traits and criteria. Developers also abstract and implement the components differently. Therefore, it is impossible to find an absolute threshold for how many times the primitive types can be used and to apply it to every project.

Since defining Primitive Obsession is hardly possible with a single formula, the deconstruction of the bad smell is a logical approach. One such deconstruction is the Primitive Enthusiasm metric presented in the next subsection.

3.3 Primitive Enthusiasm

Primitive Enthusiasm is a method level metric that captures and reports one important aspect of the Primitive Obsession bad smell (Gál and Pengő, 2018). It is based on Formulas 1 and 2. The definitions of the parameters are the following:

- *PrimitiveTypes* is the set of types that are handled as primitive ones.
- N represents the number of methods in the current class.
- M_i denotes the ith method of the current class.
- M_c denotes the current method under investigation in the current class.
- P_{M_i} denotes the list of types used for parameters in the the M_i method.
- $P_{M_i,j}$ defines the type of the jth parameter in the M_i method.

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

Formula 1 describes how the primitive-typed parameters are collected for a given M_i method. That is, select all the parameters from the M_i method that can be found in the previously introduced *PrimitiveTypes* set and return them in a list.

The calculation of Primitive Enthusiasm is depicted in Formula 2.

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

The left-hand side of the inequality in Formula 2 denotes the percentage of how many of the parameters of the current class are primitive types. Sum the number of primitive types in the parameter list for each method in the currently processed class and divide this value with the total number of parameters in the class methods. The right-hand side of the inequality in Formula 2 calculates the percentage of the primitive types used in the currently investigated method. The number of primitive types in the current method is divided by the total number of parameters. Both sides of the inequality will calculate a number between 0.0 and 1.0 as the number of all parameters is always greater than or equal to the number of parameters that are primitive types. It is important to note, that only methods with at least one parameter can be used as the subject for the Formula. Methods without parameters do not use any primitive types as input, thus they are not the target of Primitive Obsession code smell.

The formula is calculated method-by-method in a given class context, therefore it is considered as a local value. Thus, in the upcoming parts of the paper this formula will be referred as Local Primitive Enthusiasm (LPE). The value of LPE for a given M_c method ($LPE(M_c)$) is either true or false depending on the satisfiability of the inequality in Formula 2.

4 PROPOSED METRICS

As LPE captures only a small part of the Primitive Obsession bad smell, it is a good idea to refine the formula through various changes. In this section, different variants of LPE for methods are constructed and also new metrics that capture other aspects of Primitive Obsession are presented. These formula variants and their combinations can be used to pinpoint smelly code parts in the program.

[1] https://github.com/stevenalowe/kata-2-tinytypes

4.1 Local Primitive Enthusiasm Variant

The original implementation of LPE was for Java, therefore the *PrimitiveTypes* set contained the following types: `boolean`, `byte`, `short`, `int`, `long`, `char`, `float`, `double`, and `String`. In Java however 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 primitives so the construction of the Local Primitive Enthusiasm with Wrapper classes (LPE_{W+}) is proposed by extending the *PrimitiveTypes* set of LPE with the previously listed boxing types.

4.2 Global Primitive Enthusiasm

Global Primitive Enthusiasm (GPE) is shown in Formula 3, where G is a list which contains all the methods in the analysed application. The right-hand side of the inequality is the same as in Formula 2, the difference can be seen on the left-hand side. Unlike in LPE, the global primitiveness ratio is calculated based on every method in the project and the examined method is compared to it.

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

The idea behind GPE is that it can be profitable to compare classes with each other to see which ones are outstanding in regard to the primitive type usage in the parameter lists of their methods.

Similarly to LPE_{W+}, a new variant can be constructed from the GPE by including the boxing types of Java in the *PrimitiveTypes* set. This new variant will be referred to in the upcoming sections as Global Primitive Enthusiasm with Wrapper classes (GPE_{W+}).

4.3 Hot Primitive Enthusiasm

Both LPE and GPE report a set of methods that might be affected by Primitive Obsession. To focus the attention of the developer on the more suspicious code parts, the combination of LPE and GPE is proposed. The combined formula, Hot Primitive Enthusiasm (HPE) is presented in Formula 4. HPE reports only the methods that are reported by both LPE and GPE.

$$HPE(M_c) := LPE(M_c) \wedge GPE(M_c) \quad (4)$$

The Hot Primitive Enthusiasm with Wrapper classes (HPE_{W+}) can be seen in Formula 5. HPE_{W+} is based on LPE_{W+} and GPE_{W+} where the Java wrapper classes are included in the *PrimitiveTypes* set.

$$HPE_{W_+}(M_c) := LPE_{W_+}(M_c) \wedge GPE_{W_+}(M_c) \quad (5)$$

4.4 Result Aggregation

When evaluating a Primitive Enthusiasm variant on a project, a set of methods is returned for which the formulated inequality was true. 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. This way, the classes can be ordered by the number of their reported methods.

4.5 Method Parameter Clones

After examining the nature of the Primitive Obsession, it is reasonable to assume that if a class suffers from Primitive Obsession, certain method parameters will appear multiple times in the parameter lists of its methods. They will have the same type and – usually – the same name because logically they correspond to the same data. These are the parameters that could be extracted to a value object instead of using them separately. To grasp this property the Method Parameter Clone (MPC) metric is presented, which performs the following steps for each class in a project:

1. Initially the MPC value for a class is set to zero.
2. For each method parameter in the class, create a $(type, name)$ pair.
3. Select only the pairs where the type is in the *PrimitiveTypes* set.
4. Increment MPC value by one for every $(type, name)$ pair that appears at least three times.

The reason that only three or more repetitions are counted is *The Rule of Three* (Fowler, 1999).

4.6 Static Final Primitives

Another aspect of the Primitive Obsession bad smell is the usage of class constant values as type codes. To check if a variable can be a candidate for further investigation the Static Final Primitive function is used which is described in Formula 6.

$$\begin{aligned} SFP(V) \quad := & \, isClassLevel(V) \\ & \wedge isStatic(V) \\ & \wedge isFinal(V) \\ & \wedge isUpperCase(V) \\ & \wedge typeOf(V) \in PrimitiveTypes \end{aligned} \tag{6}$$

The SFP function will return true for a variable (V), if it has all the following properties, thus it might serve as a type code:

- V is a class level variable,
- V is static (e.g. has a `static` modifier in Java)
- V can be assigned only once (e.g. has a `final` modifier in Java),
- the name of V contains only upper case characters, numbers, and underscores,
- and the type of V is included in the previously defined *PrimitiveTypes* set (i.e. primitive).

Otherwise SFP returns false. With this function it is possible to filter out static final primitive variable usages in methods.

The Static Final Variable Usage (SFPU) function is defined in Formula 7. It has three parameters: M_c and V denote the function and variable under investigation, respectively, while F is a filter function. In the formula, U_V denotes one usage of variable V. The $U_V \in M_c$ relation means, that the U_V variable usage (access or modification) is preformed in the M_c method. By applying the F filter function it is possible to select a subset of the U_V variable usages. The *SFPU* function collects the access and modification statements for variable V in the M_c method for which both the *SFP* function and the supplied F filter function returns true.

$$\begin{aligned} SFPU(M_c,V,F) \quad := & \{U_V \mid \\ & U_V \in M_c \wedge SFP(V) \wedge F(U_V)\} \end{aligned} \tag{7}$$

A concrete application of the *SFPU* function is the calculation of the number of static final primitive variables which are used as case labels in `switch` statements. This can be achieved by defining an F filter function for the *SFPU* that determines for a given U_V variable usage if it is a case label. By using the *SFPU* and the given F filter function a class level metric is constructed, which is named Static Final Primitive - Switch Case Usage (SFP-SCU). The calculation is done for each class to see how many times its *SFP* variables appear as case labels globally in the project. This heuristical approach is based on the idea that type code variables most probably will appear in branching structures, especially in `switch-case` statements.

5 EVALUATION

To implement and evaluate the proposed formulas and their variants, the OpenStaticAnalyzer[2] was used, which is an open-source, multi-language, static code analyser 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.

The Primitive Enthusiasm variants, Method Parameter Clones, and the SFP-SCU metric was evaulated on three open-source Java projects. As there is no benchmark for Primitive Obsession detection manual validation was performed.

5.1 Eliminated Methods

The formulas introduced in Section 4 for Primitive Enthusiasm calculation do not take into account that some methods have only one parameter or none at all. Also there are certain types of methods that should be eliminated form the calculation of the metrics. During the implementation this was taken into account.

The original implementation of LPE skipped the investigation of constructors, the methods with empty parameter list, and methods that could be considered a class member setter method. Upon further work the restriction was enhanced to skip the processing of all the methods that had less than two parameters. A comparison on the two elimination approaches was performed.

5.2 Results

As the result validation was done manually, first the study of the analysed projects was done: *joda-time-2.9.9*[3], *log4j*[4] and *commons-math-3.6.1*[5]. Table 1 summarizes the major properties of the investigated systems to get an overall image about them.

5.2.1 Exclusion Strategy

As it was stated in Section 5.1, two strategies was evaluated for eliminating methods: to skip only setter methods (*Skip Setter Methods - SSM*) or skip every

[2]https://github.com/sed-inf-u-szeged/OpenStaticAnalyzer

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

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

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

Table 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

method with just one parameter (*Skip Ones - SO*). Table 2 sums up the details of the two strategies for the examined three systems.

Table 2: Comparison of the two elimination stategies: 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 with the SO strategy only a subset of the methods was processed for the calculation. During the manual validation it was 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.

5.2.2 Wrapper Classes

In the paper different variants of the Primitive Enthusiasm metric are defined, based on the content of the *PrimitiveTypes* set. LPE_{W+}, GPE_{W+} and HPE_{W+} take into account the wrapper classes of Java during the Primitive Enthusiasm calculation, whilst LPE, GPE, and HPE do not. To compare the effect of including boxing types into the *PrimitiveTypes* set, the number of methods with at least one (AL1) and at least three (AL3) privitive typed parameters were counted. During this calculation only the constructor methods were eliminated, the previously introduced skip strategies were not applied. With this it can be observed how the inclusion, or exclusion of wrapper classes changes the number of investigated methods.

Based on the results the contrast is much less than expected. These results are summarized in Table 3. It shows that the extension of the *PrimitiveTypes* set with boxing types has a very little consequence on the set of methods that use primitive types in their parameter lists. The difference is only one or two methods or none at all. Concluding these results, the SFPU, SPU-SCU, and MPC calculations was done by using the extended *PrimitiveTypes* set.

Table 3: Investigating 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

5.2.3 Primitive Enthusiasm Metrics

The Primitive Enthusiasm metrics suggest such methods for refactoring that use unusually lot primitive parameters. A result aggregation on the warnings is also performed according to Section 4.4.

Table 4 shows the number of reported methods of the three studied Java systems on the left side, while on the right side, the number of classes which have at least one reported method can be seen. The results of Table 4 were produced with the *Skip Ones* (SO) strategy.

To validate the results of the various Primitive Enthusiasm metrics, the given warnings were processed manually. 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 judgement of Primitive Obsession itself are subjective, deciding if a warning is true positive or not is hard to tell, therefore the the significance of the warning was investigated.

It is clear from the results of Table 4 that the Primitive Enthusiasm metrics report many methods and classes. However, the number of reported methods are less than 8% of the total number of methods in average. The other important aspect of Table 4 is that there is almost zero difference between the metrics which include and which do not include the wrapper classes. Considering the results of Section 5.2.2 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 and GPE_{W+} express how different the composition of the parameter list of a method is from the global average, whilst LPE and LPE_{W+} express only a class level distinction. The combined metrics HPE 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

Table 4: Comparison of Primitive Enthusiasm methods using the SO exclusion strategy.

	Number of reported methods						Number of reported classes					
	LPE	LPE_{W+}	GPE	GPE_{W+}	HPE	HPE_{W+}	LPE	LPE_{W+}	GPE	GPE_{W+}	HPE	HPE_{W+}
log4j	165	165	217	217	153	153	29	29	74	74	29	29
joda-time	301	301	429	431	230	231	92	92	104	105	65	66
commons-math	698	698	1192	1192	553	553	213	213	390	390	145	145

reported methods and classes was more than 25% percent. Based on this information it is recommended that first the HPE or the HPE_{W+} reports should be investigated by the developer as they report both class-localized and application-global suspicions.

Reviewing hundreds of methods can be tedious work, however, by integrating the warnings into an IDE this burden can be dramatically reduced.

5.2.4 MPC Metric

The main interest in the MPC metric was to study how well the reported classes align with the classes reported by the aggregated Primitive Enthusiasm metrics. For this evaluation the SO exclusion strategy was used and the wrapper classes were included in the *PrimitiveTypes* set.

MPC reported 6, 50, and 90 classes on *log4j*, *joda-time* and *commons-math*, respectively. The highest MPC count was 11 for *log4j*, meaning that in the `logMF` and `logSF` classes there are 11 parameter type - parameter name pairs that appeared at least three times in different parameter lists. Both of these classes were the top two results of the aggregated Primitive Enthusiasm metrics. They contain numerous methods that have a similar signature with multiple primitive typed parameters, so the metrics report a repetitiveness in the code that sometimes indicates a design flaw in the program. The `logXF` class, the base for `logMF` and `logSF` is also reported with an MPC count of 2 and has a high rank in the aggregated Primitive Enthusiasm results. The other three classes were also reported by at least one of the Primitive Enthusiasm variants.

The findings are similar for the other two projects too. The top ten classes in the aggregated HPE_{W+} results could be found in the MPC result set as well. Classes that have many methods with above the average primitive typed parameters usually have lots of method parameter clones too. Therefore the MPC metric can be a candidate for weighting the results of the aggregated Primitive Enthusiasm metrics and highlight more repetitive parts in the code.

The intersection of the aggregated HPE_{W+} and MPC result sets for *log4j*, *joda-time* and *commons-math* has the following size: 5, 31, and 56. The numbers show that by intersecting the result sets, a significant number of warnings were pruned from both metrics. Although we may lose some useful warnings, this way we eliminate many noisy results.

5.2.5 SFP-SCU Metric

The metric was designed to detect type codes which were introduced in Section 3. It follows from the nature of type codes that they appear in conditional statements. The heuristical approach was to spot them through their usage in `switch-case` statements. Table 5 shows the results of SFP-SCU. It is important to note that during the evaluation no exclusion strategy was used, thus all methods were considered. However, the effects of the SO exclusion strategy combined with excluding constructors were also investigated and the results were the same for the studied projects. The wrapper classes were included in the *PrimitiveTypes* set.

The number of classes reported by SFP-SCU is in the second column of Table 5. These classes have primitive-typed static final class members that are suspected to be type codes. The size of intersection between the results of the aggregated HPE_{W+} and SFP-SCU is presented in the third column, whilst the size of the intersection with the MPC metric in the fourth column. As this metric grasps a different aspect of Primitive Obsession than the Primitive Enthusiasm metrics no significant overlap was expected.

Table 5: Results of SFP-SCU. From left to right: the number of classes(NOC) found in the result of the SFP-SCU metric, number of classes in the intersection of the aggregated HPE_{W+} and SFP-SCU results, number of classes in the intersection of the MPC and SFP-SCU results

Project	NOC	NOC $\cap$ HPE_{W+}	NOC $\cap$ MPC
log4j	8	5	1
joda-time	13	4	2
commons-math	1	0	0

The constant variables, that can be found in the reported class of the *commons-math* project are used to distinguish between random number generation strategies. They have only one `switch-case` occurrence

and could be replaced, for example, with an `enum`. The *joda-time* relies more on static final variables than the other two projects. These variables are members in multiple classes with the same name, making it harder to understand the code. A generalized solution could be considered instead of the scattered constant usage. In `log4j` a typical example for type codes can be found in the `PatternParser` class, which also appears in both intersections.

Though no hidden dependency check was done to study the seriousness of the possible type codes it might be rewarding to refactor them for a more object oriented and readable solution.

6 CONCLUSION

In this paper several variants for the basic Primitive Enthusiasm metric were introduced and other three metrics were defined to grasp more aspects of the Primitive Obsession bad smell. The metrics were implemented in a Java static analyser, evaluated on three large systems and the results were analysed.

The Primitive Enthusiasm variants can find methods that use more primitive types in their parameter lists as the average. It is not just a readability issue but can be a sign for other bloater type smells as well. The SFP-SCU metric is useful for typed code detection. In the future the authors would like to consider if a static final variable can be seen outside its class or not by giving the usages outside its class or package a different weight than the inner usages. Additionally involving other conditional statements in the calculation besides `switch-cases` can be another improvement. The MPC metric reports classes that have repetitive, therefore possibly smelly method signatures. The metric could be refined with ordinal information among the parameter clones.

The findings showed that the new metrics can highlight many smelly and hardly readable code segments. In the future we would like to continue the study of these metrics and their combinations. The inclusion of enum constants in the *PrimitiveTypes* set could be an interesting experiment. Creating a Primitive Obsession benchmark is also a goal to provide a more objective comparison of the metrics.

ACKNOWLEDGEMENTS

This research was 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. The project has been supported by the European Union and co-funded by the European Social Fund.

REFERENCES

Fontana, F. A., Mariani, E., Mornioli, A., Sormani, R., and Tonello, A. (2011). An experience report on using code smells detection tools. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 450–457.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA.

Gál, P. and Pengő, E. (2018). Primitive enthusiasm: A road to primitive obsession. In *The 11th Conference of PhD Students in Computer Science*.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Gupta, A., Suri, B., and Misra, S. (2017). A systematic literature review: Code bad smells in java source code. In *Computational Science and Its Applications – ICCSA 2017*, pages 665–682, Cham. Springer International Publishing.

Mäntylä, M. V., Vanhanen, J., and Lassenius, C. (2003). A taxonomy and an initial empirical study of bad smells in code. In *Proceedings of the International Conference on Software Maintenance*, ICSM ’03, pages 381–, Washington, DC, USA. IEEE Computer Society.

Mäntylä, M. V., Vanhanen, J., and Lassenius, C. (2004). Bad smells - humans as code critics. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, ICSM ’04, pages 399–408, Washington, DC, USA. IEEE Computer Society.

Moonen, L. and Yamashita, A. (2012). Do code smells reflect important maintainability aspects? In *Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM)*, ICSM ’12, pages 306–315, Washington, DC, USA. IEEE Computer Society.

Roperia, N. (2009). Jsmell: A bad smell detection tool for java systems. Master’s thesis, Maharishi Dayanand University.

Yamashita, A. and Moonen, L. (2013). To what extent can maintenance problems be predicted by code smell detection? an empirical study. *Information and Software Technology*, 55(12):2223 – 2242.

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

Zhang, M., Hall, T., and Baddoo, N. (2011). Code bad smells: A review of current knowledge. *Journal of Software Maintenance and Evolution*, 23(3):179–202.