Primitive Enthusiasm: A Road to Primitive Obsession

Péter Gál, Edit Pengő

Abstract: Code bad smells usually indicate that there are low quality, hardly readable and maintainable parts in the source code. Static analysis tools can help programmers to identify bad smells and guide the refactoring process. Bad smell detection techniques have a considerable amount of literature although there are still a few types that need further study by the research community. The Primitive Obsession smell is one of them.

In this paper, we studied the definition of Primitive Obsession and based on that we introduced a method level metric, Primitive Enthusiasm, that can be used as an indicator for the smell. We implemented an algorithm for Primitive Enthusiasm calculation on Java code and analysed two real life Java systems along with a sample project. As no benchmark was available, we performed a manual validation of the results.

Keywords: Code smells, Primitive Obsession, Static analysis, Refactoring

Introduction

Refactoring means improving different attributes of the program without changing its external functionality [2]. Through code changes the original structure of the code will decay, making it harder to identify design elements, understand its behaviour, and to find bugs. The main goal of refactoring is usually to reduce the complexity of the code, to improve its readability, and therefore to make it more maintainable. There are several indicators that suggest the need of refactoring in the code. The code smells identified by Fowler can be such indicators [6].

Primitive obsession, that can be vaguely interpreted as the overuse of primitive data types, has lacked the attention of the research community. A recent literature review by Gupta et al. [3] concluded that currently existing smell detection tools and techniques do not provide support for this type of smell. They collected 60 research papers form 1999 to 2016 and found that four of Fowler's bad smells – including Primitive Obsession – were not detected in any of the papers.

Therefore, we decided to study Primitive Obsession and defined an indicator which can highlight the potential places where this bad smell could occur. We call this indicator Primitive Enthusiasm and it focuses on one major aspect of the smell. We implemented the indicator algorithm for Java source code, however, the method can be generalized to other languages as well. The algorithm was tested on two real life Java projects and on a sample program seeded with Primitive Obsession. A manual validation of the results was performed.

Defining Primitive Obsession

In most programming languanges, data types can either be categorised as primitive or complex types. The building blocks of complex types are primitive types, e.g., integers or booleans. The benefits of complex types is that they usually provide semantic information about the variables. For example, from three integer values we can form a meaningful structure of a date record. It is much easier to grasp the essence of a code fragment if the operations are carried out on small, straightforwardly named objects rather than using dozens of distinct primitive types.

Primitive Obsession smell means that the programmer does not create small objects for small tasks, instead s/he is obsessed with the use of primitive data types. However, this definition is not really exact and can be interpreted broadly. Chapter 3 of the Fowler book [2] makes the previous guideline clearer by providing a list of possible refactorings for Primitive Obsession. A small GitHub project [5] aids our understanding by showing these refactorings step-by-step in practice.

The code fragment in Figure 1 is packed with suspicious lines. Using type code like the constant class variables `ENGINEER` and `SALESMAN` instead of a State or Strategy class can be considered as Primitve Obsession. From the first three parameters of the `work` method, the developer could create a Shift class. A method having too many primitive data types in its parameter list can be a sign of Primitive Obsession. The extraction of class members to primitive local variables, like on lines 5 and 6, can also be recognised as Primitive Obsession.

```
class Employee{
static const int ENGINEER = 1;
static const int SALESMAN = 2;
public void work(int from, int to, int numberOfBreaks, Task task){
String taskName = task.getName();
int hardness = task.getHardness();
/* ... */
}
}
```

Figure 1: Sample code contaning three Primitive Obsessions

The sample code in Figure 1 shows that identifying Primitive Obsession usually needs semantic knowledge about the code. Moreover, the number of primitive types used in a software is application, problem, and coding guideline dependent. Thus, it is not possible to formulate an exact rule on how much the primitive types can be used before we report Primitive Obsession.

Indicator for Possible Primitive Obsession

Considering the challenges described in Section , we decided to restrict our definition of Primitive Obsession. In this section we present Primitive Enthusiasm, a metric based on this more exact definition. Primitive Enthusiasm can be used as an indicator to highlight potential candidates for Primitive Obsession.

Implementation for Java

The current implementation of Primitive Enthusiasm is for Java code, however, it can be adapted for other languages as well.

To calculate the metric, the algorithm needs to know which types should be considered primitive. We used the definition of primitive types found in the Java SE 9 language reference, Section 4.2 [7]. The following types are considered primitive: `boolean` `byte`, `short`, `int`, `long`, `char`, `float`, `double`. Additionally, we also consider the `String` as a primitive type. This is because strings in Java are immutable and could be considered as a quasi primitive type. This set of types is used as the $PrimitiveTypes$ set in the equations below.

For the calculation we used the API of OpenStaticAnalyser [1], which is an open-source, multi-language, static code analyzer developed at the Department of Software Engineering, University of Szeged.

Primitive Enthusiasm

With Primitive Enthusiasm, we limited the detection of Primitive Obsession to detecting the overuse of primitive types in method parameters. However, as it was previously stated, the need for primitive types can be application dependent, therefore we could not formulate an exact number on how much the primitive types can be used. Instead, we composed Primitive Enthusiasm as the function of the current and overall primitive type usage in the method parameter lists of a class. Our algorithm processes the examined program class-by-class and calculates the indicator for each method of the class.

Primitive Enthusiasm is a method level metric based on Formula 25 and 26. The definitions of the parameters are the following:

- $\boldsymbol{N}$ 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 the $\boldsymbol{M_i}$ method.

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

- $\boldsymbol{P_{M_i,j}}$ defines the type of the $\boldsymbol{j}$th parameter in the $\boldsymbol{M_i}$ method.

$$Primitives(M_i) = \langle P_{M_i,j} \bigm| 1 \leq j \leq |P_{M_i}| \wedge P_{M_i,j} \in PrimitiveTypes \rangle \tag{25}$$

Formula 25 describes how the primitive-typed parameters are collected for a given M_i method. That is, we select all those parameters from the M_i method that can be found in the previously introduced `PrimitiveTypes` set and it is returned as a list. Currently we do not care for the order of the parameters as this information is not relevant.

$$\frac{\sum_{i=1}^{N} |Primitives(M_i)|}{\sum_{i=1}^{N} \mid P_{M_i} \mid} < \frac{|Primitives(M_c)|}{\mid P_{M_c} \mid} \tag{26}$$

The left side of Formula 26 denotes the percentage of how much of the parameters of the current class are primitive types. We count 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 side of Formula 26 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.

If the percentage of primitive typed parameters in the current method is bigger than the percentage of primitive typed parameters in the class, the algorithm marks the method as Primitive Enthusiasm, indicating a potential place for Primitive Obsession. In case a method does not have any parameter then the calculation is skipped for that method.

It is important to note that we exclude Java setter methods, constructors, and lambda functions from the process. Setter methods are the conventional way to manipulate the primitive and non-primitive-typed class members therefore their number and their parameter type is determined by the data fields of their class. This way we eliminated reporting false positives on Java Beans. In our current approach, the methods are treated as setters if their name starts with `set`, they have only one parameter, and their return type is void. Additionally, as the metric processes method parameter types, it does not "look inside" each function, thus lambda functions are not taken into account.

Result Aggregation

The algorithm marks methods one-by-one with Primitive Enthusiasm, therefore on large projects, the list could grow ineffectively long. It can happen that it is not visible at first glance what are the most crucial places where Primitive Obsession can occur. To overcome this problem, we introduced an aggregation method that gives a class level view of the results. Thus, we rank the classes by the number of reported methods. This way we can easily see which classes might be heavily affected by Primitive Obsession.

Results

To validate usability of the proposed indicator, three projects were processed. The first project was the small GitHub project [5], which shows six variants, each with less and less primitive type parameters, as the method arguments are refactored into classes. On the first variant, our algorithm reported all five functions – the project only has five important functions – as potential methods suffering from Primitive Obsession. These five functions are indeed refactored in subsequent versions to use different types instead of primitive types.

The other two projects were the Titan [8] and ArgoUML [1] tools. For both projects there are multiple reports of Primitive Enthusiasm. Based on the initial manual validation of the results it can be stated that the class with most reports is indeed suspicious of Primitive Obsession.

Conclusion

In this work, a new indicator is presented to aid developers in detecting code elements possibly suffering from Primitive Obsession. The Primitive Enthusiasm idicator processes the method type parameters to report suspicios methods. Furthermore, an aggregation of the reported methods into a class level metric can aid the programmers to focus their attention.

We see many possibilities to improve our Primitive Obsession detection method. In the future, we would like to involve more aspects of the smell. For example, the detection of type codes, that were briefly introduced in the sample code of Section would highlight more places where this smell can occur. The usage of primitve class members should also be taken into consideration. The utilization of other metrics, like the Halstead [4] metrics which gives information about the complexity of the analysed method could also improve the reliability of our detection method.

Acknowledgment

This research was supported by the EU-funded Hungarian national grant GINOP-2.3.2-15-2016-00037 titled “Internet of Living Things”.

References

[1] ArgoUML UML modeling tool (accessed: 2018-03-28). `https://github.com/stcarrez/argouml`.

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

[3] 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 International Publishing.

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

[5] Refactoring from ’primitive obsession’ confusion to ’tiny types’ clarity (accessed: 2018-03-28). `https://github.com/stevenalowe/kata-2-tinytypes`.

[6] Satwinder Singh and Sharanpreet Kaur. A systematic literature review: Refactoring for disclosing code smells in object oriented software. *Ain Shams Engineering Journal*, 2017.

[7] The Java Language Specification, Java SE 9 Edition (accessed: 2018-03-28). `https://docs.oracle.com/javase/specs/jls/se9/jls9.pdf`.

[8] Titan distributed graph database (accessed: 2018-03-28). `https://github.com/thinkaurelius/titan`.