ECOLE POLYTECHNIQUE DE LOUVAIN

LINGI2252 - SOFTWARE ENGINEERING : MEASURES AND MAINTENANCE

Assignement 2

Professor :

KIM MENS

Students : (Group 6)

Benoît BAUFAYS 22200900

Julien COLMONTS 41630800

Program :

SINF21MS

Academic Year 2013-2014

1 Introduction

For the first assignment, we had to analyse the code quality of the Petit-Parser framework project. Since we had to perform this analysis manually and Petit-Parser is a quite large framework, it was a hard job to go through all the code and analysis wasn't perfect because there were too many lines of code to read through.

In this report, we will use some Moose tools to perform an automated code analysis of the same system. Our conclusion in the first report were optimistic about Petit-Parser even if we found some bad smells. Let's see if our previous conclusions are still valid after a deeper analysis with some metrics.

2 Overview

2.1 The Moose Pyramid

The first tool given by Moose that is visualizing feature which gives a pyramid containing general metrics of the overall system. In this pyramid, you can observe three main blocks: one on the top

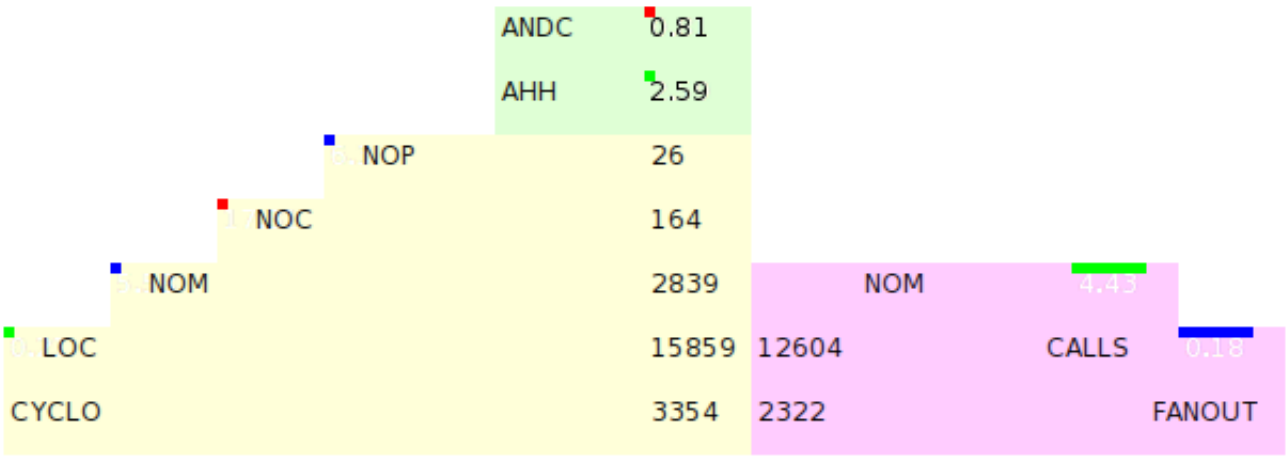

Figure 1: Overview Pyramid

and two at each side of the bottom.

The block on the top, in green, gives metrics about inheritance measurements. The first metric is the average number of derived classes. The little red square at the corner of this metric cell tells us that the computed number given is higher than what is known as a normal value. This could be explained. Since PETITPARSER is a framework for parsing several languages, his model is quite complex and need a large number of derived class dedicated to each kind of parsing strategy. The second metric is the average hierarchy height. Moose tells use that the value computed is close to average. Mixing the results of these two values, we can conclude that PETITPARSER uses a large number of classes which are derived in a number of levels which are still reasonable.
The block at the bottom left side gives several metrics about the size. The first metric shown is the number of packages. The number computed is lower than what would be expected for such a framework. But, as we already navigate manually through the code, we can say that strict rules are respected to build packages. These packages sometimes contain a large number of classes, especially the ones with tests and with parser types. The number of packages metric can be a bit distorted for this kind of framework. The second metric represents the number of classes. This time, the number is too high. To understand why this result could be a bad smell, we went back to the system browser. We saw that packages are mostly divided in two categories. There are packages that are almost empty. A very short number of classes are implemented. There are also packages with a way too many classes. After searching why these packages are too big, we found that the classes they put in them were still logically managed. For instance, *PetitParser-Parsers* package contains a large number of classes but these classes must be in this package. They are describing the different parsing strategies for the different types of parsers. The next metric is the number of methods. The number is a bit lower than what is expected. This observation comes from the fact that we have a large number of derived class. Some methods are written in a class higher in the hierarchy and don't need to be defined in all subclasses anymore. The last metric we'll take a look at is the number of lines of code. PETITPARSER seems to have this number in the average for Smalltalk projects.

2.2 System complexity

Before going deeper in the code, it is very interesting to identify which classes are worst than others. To visualize this, we give, below, the System Complexity of PETITPARSER. The height of the arrow symbolize the number of methods and the width symbolize the number of variables. From left to right, we show : PPAbstractParserTest,PPParser, PJASTNode, SQLASTNode and

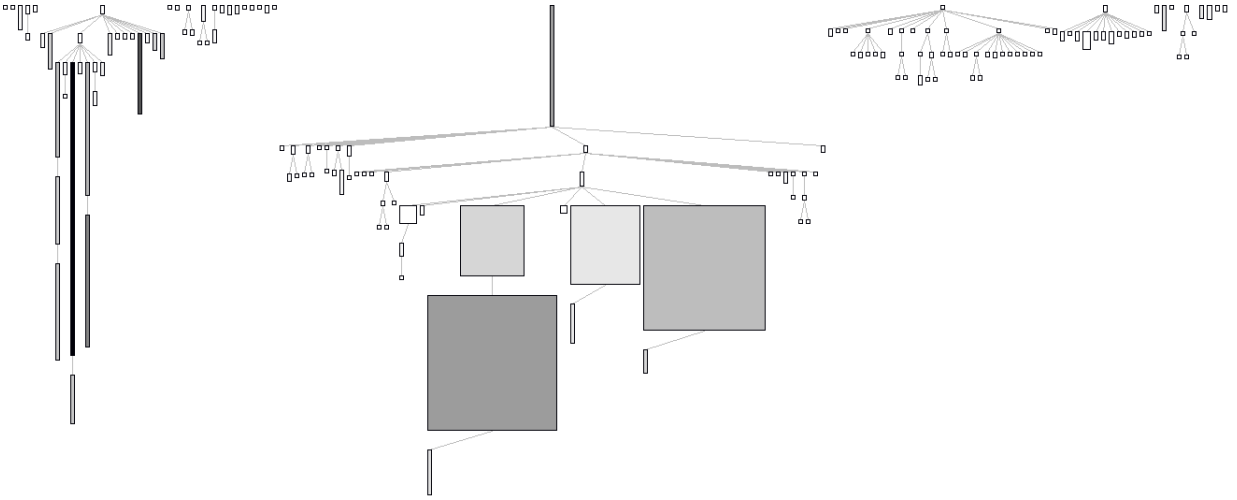

Figure 2: System Complexity

Petit Analyzer code. Other smalls arrows are objects from PetitParser or from SmallTalk. We see that PPAbstractParserTest and PPParser are worst than others. Actually, if we browse the code of this two classes, we understand the schema : for the test part, the creator of PetitParser added all his testing methods in one class. In other part, for the PPParser classes, and particularly for PPJavaSyntax and PetitSQLiteGrammar, he put all the variables in the same class. When we know the syntax for Java and SQlite, we understand that the number of variables must be very important. This graph also shows that the core of the PetitParser system is kind of efficient regarding to the complexity of this system.

2.3 Blueprints Complexity

An other tool to analyze PETITPARSER is Blueprint. With this tool, we can show invocation of methods, accesses to attributes and how methods and attributes interact.

As we have seen before with the System complexity, the test classes are not very efficient and we prefer to focus on the core of PETITPARSER. So, you can find below a schema representing the Blueprint Complexity of PETITPARSER classes and its core system.

This schema contains arrows representing classes and his hierarchy. In each arrow, we found 5

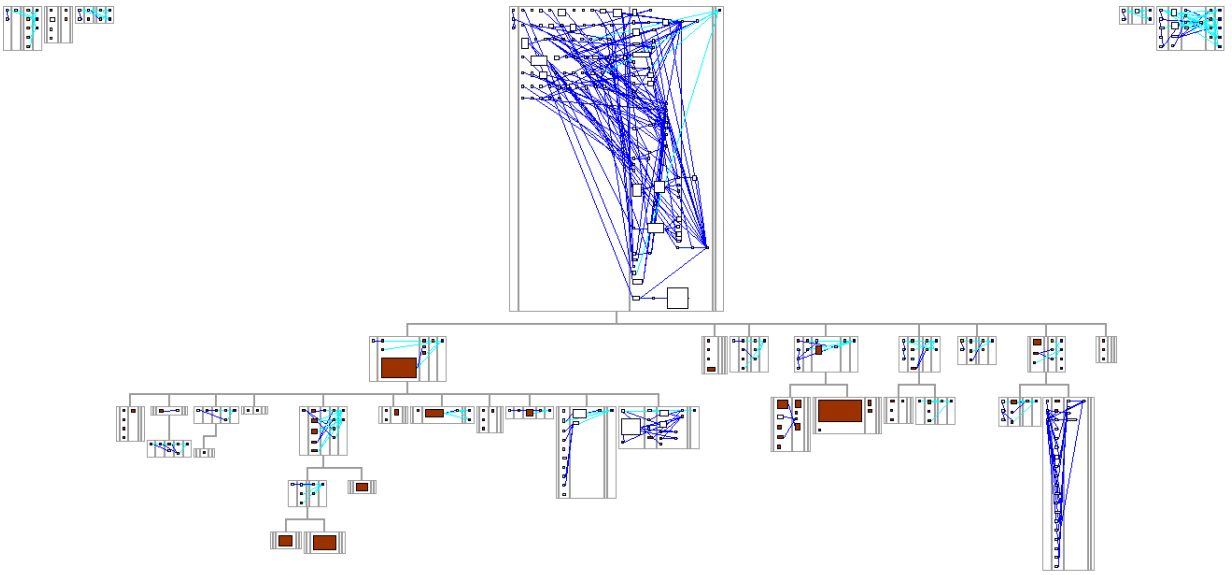

Figure 3: Blueprint Complexity of PetitParser packages

columns corresponding of differents entities we found in a class. So from left to right, we found initialization, public methods, private methods, accessors and attributes. In each column, we have also arrows representing the entity of this type in the class. We found also, crossing columns, dark and light blue links. The dark link represents the invocation of methods and the light one represents access to attributes.

In this schema, at first glance, we see that the arrow, and his tree, in the middle is the main class

of PETITPARSER. The class call PETITPARSER and, indeed, it is the main class of the project.

Blueprints Complexity of PetitParser

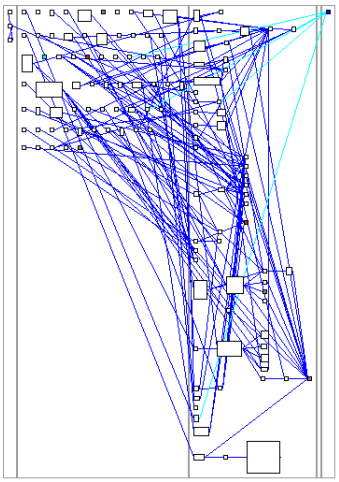

Figure 4: Blueprint Complexity of PetitParser class

In the schema, we can see more clearly the arrow representing PETITPARSER class. Based on the legend explained above, we found that it contains no accessors and one attribute, PROPERTIES which is a instanceVariable. Some methods, public or private access directly to this attribute. In theorical view, it's not good to access directly, you must prefer to use accessor. With accessor, you can validate the information before setting your attributes. Of course, it is less important for private method if you are very carefull when you use an attribute. But, for public method, you must use a attribute. Of course, it's a theorical view and we look all methods accessing directly the attribute before getting more analysis. We have found 5 methods :

- removeProperty: aKey ifAbsent: aBlock : this private method remove the property with aKey. If a key is found, it returns the associated value and the the result of evaluating aBlock otherwise. According to the code, we can say that this method is a safe method for the attribute : it removes the key if, and only if, the key exists. With this solution, the integrity of the attribute is checked;

- propertyAt: aKey put: anObject : this private method change the object (anObject) linked to the key (aKey) in the property. if aKey is not found, it creates a new entry. Again, we see that this method is a safe method and the integrity of the attribute is checked;
- propertyAt: aKey ifAbsent: aBlock : this private method give the value associated to aKey if aKey is a key in property and answer the result of evaluating aBlock otherwise. Like the first method, we see that the integrity of the attribute is checked ;
- postCopy : this public method give a copy of the property. To perform this copy, it is normal to access to the attribute but, for a public method, it's preferable to use a getter, event if we don't change it ;
- hasProperty: aKey : this public method test if aKey is a key in the property. Also, like the previous public method, the integrity of the attribute is checked but it's not good to access directly to an attribute in a public method.

In ddition to checking the code of methods, it is interesting to note that this methods are well used by this class or children classes.
Even if the integrity of the attribute is checked in every method accessing directly to the attribute, we see that almost method performs a test on property. Why the creator doesn't use the test method in other methods listed below ?
Finally, we see that some methods are setter for property. Because property contains keys and values, the creator can not do a "simple" setter to change the property but he create method to add or remove (key,value). So, in conclusion, we can see that it's important to check the code after reading this schema and, evenf if we found no getter and setter, the most methods accessing directly the attribute are setter and getter for the structure of property.

For the dark blue links, we see that a lot of methods are linked together. It's a signe that the code are modulable and reduce code duplication. It is also the consequence of the PETITPARSER system : it uses structures with childrens and list of object. To access these structures, methods use these sub-structures and methods related to them.

Finally, we see some brown arrows. It means that the method override code. AND ????

2.4 Duplication side-by-side

Moi, Moose me renvoit un schéma vide, WTF ? - Pareil :)

2.5 Bad smells

With the Moose tool, we can have metrics about almost everything but they are not all relevant to detect bad smells. To Detect more precisly some bad smells, we have created some query to get bad methods.

Number of lines of Code

First, we want to see methods where the number of lines of codes is bigger than 50, we can query :

```
each numberOfLinesOfCode>50
```

With this query, we can see that, in the PETITPARSER System, we found 7 classes with more than 50 lines of codes.

Figure 5: Complexity of big classes

In the schema below, we can see that the class PPParser is the bigger one, with 758 lines of codes. To go depper in the analysis, we can perform the query for this class and we see

plante chez moi

Accessors

An other interresting metric is the number of getter and setter for one attribute. In mai ncase, you must have one getter and one setter for every attribute. We see before that, for some attributes, we don't have any accessors.

```
each numberOfAccessorMethods >= ( 2 * each numberOfAttributes )
```

This query return 22 classes and we can visualize these with the blueprint schema.

Here we can notice that almost classes have 2 accessors methods for one attribute, except the

Figure 6: Blueprint of classes with more accessors than 2* attributes

PPFailingParser class with one attribute and 3 accessors. In fact, when we analyze the code, we see that we have one getter, one setter and one method to show the attribute because the attribute is a message with a specific format who are not very readable. Like earlier in this report, we see a lot of light blue links, that symbolize an access to a attribute without accessors. As the query used indicate that we have, at least, two accessors, it is a bad smell to access direclty attributes and don't use accessors. In particulary, in the class PPFailingParser, we have three accessors and some methods access directly the attribute.

At the opposite of the previous query, we can listing all classes with less than 2 accessors per attribute.

```
each numberOfAccessorMethods < ( 2 * each numberOfAttributes )
```

This query return 15 classes and we can visualize these with the blueprint schema.

The number of classes are smaller but, again, we see a lot of light blue links. For the biggest

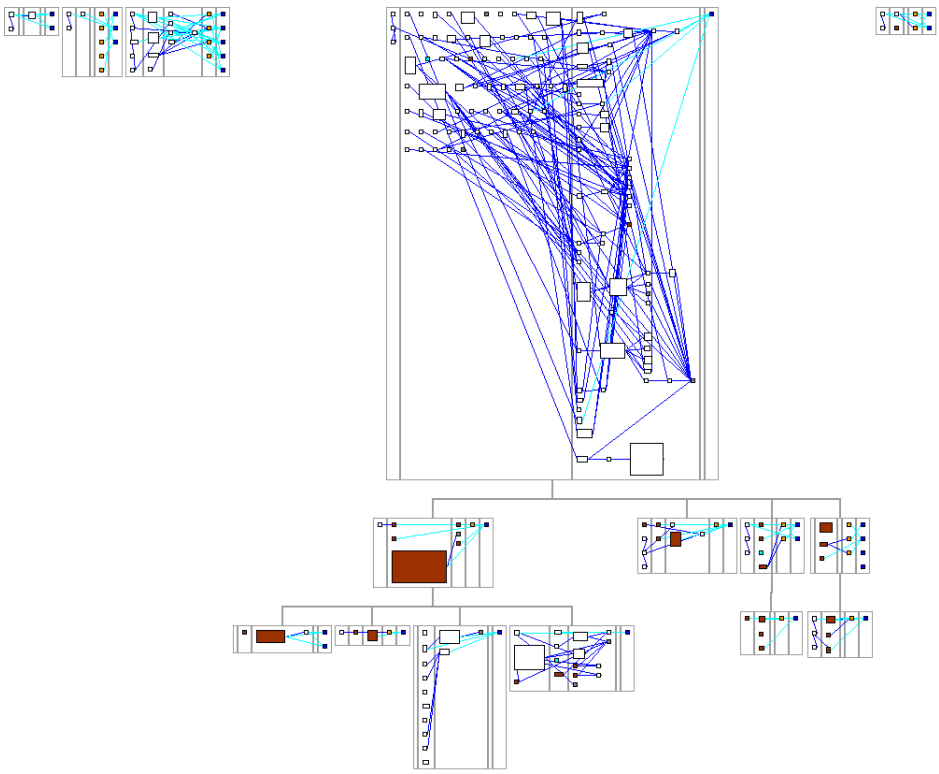

Figure 7: Blueprint of classes with less accessors than 2* attributes

arrow, we have already analysing this in a previous section because it represent PPParser class. For the others classes, we see that some of them have a lot of light blue links, like the arrow on the left of PPParser. It is the PPToken class every methods (privete or public) access directly to attributes. When we analyze the code of this class, we see that almost methods just read attributes but when you have accessors to do that, why access directly ?

duplicatior
side-
by-
side

CityMap