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 Analysis

## 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

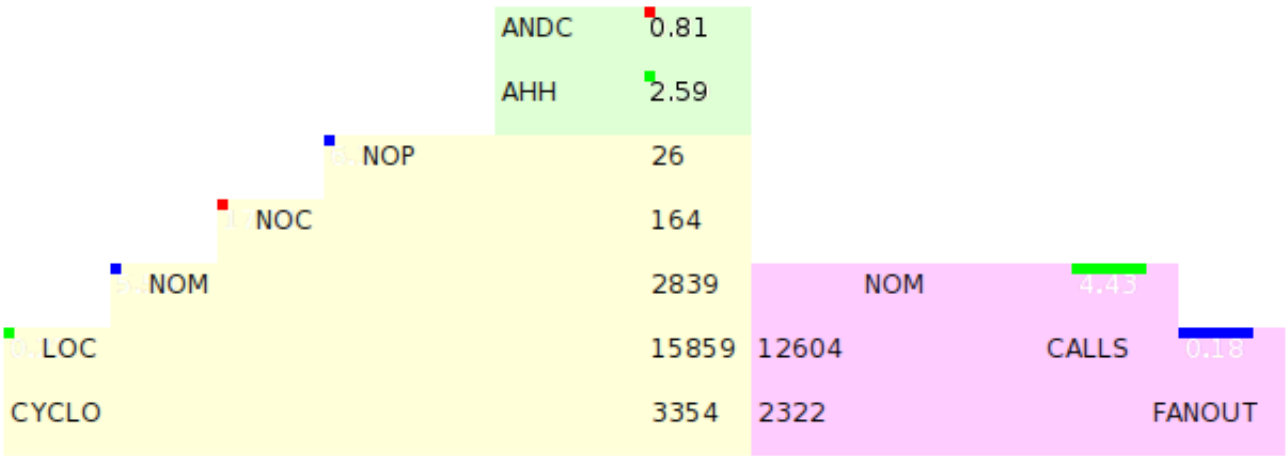


Figure 1: Overview Pyramid

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 squares symbolizes the number of methods and the width symbolizes the number of variables. From left to right, we show : PPAbstractParserTest,PPParser, PJASTNode, SQLASTNode and

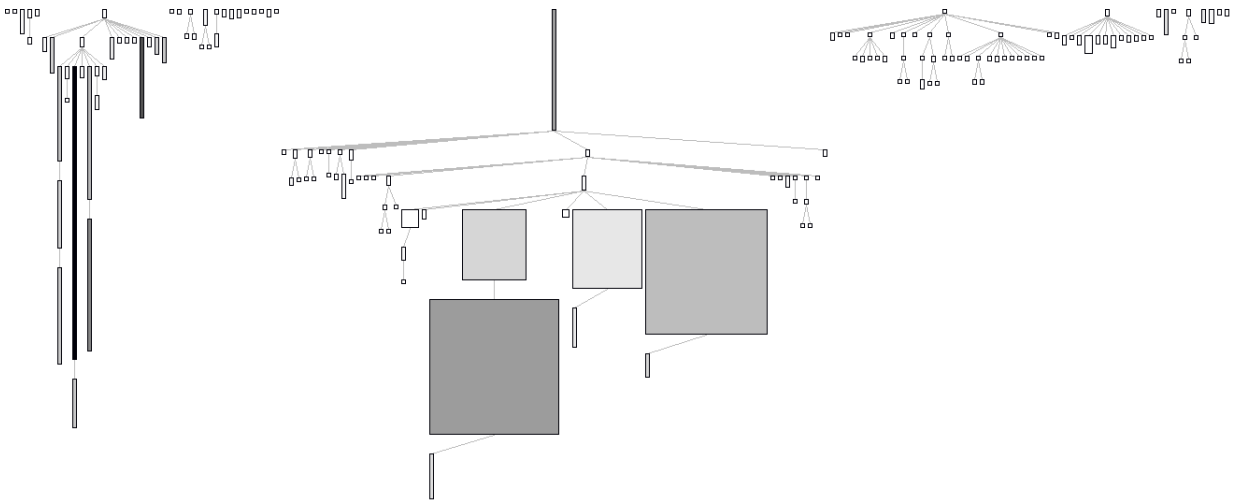

Figure 2: System Complexity

Petit Analyzer code. Other smalls squares 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 will be very large. 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 analyse 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 rectangles representing classes and his hierarchy. In each rectangle, we

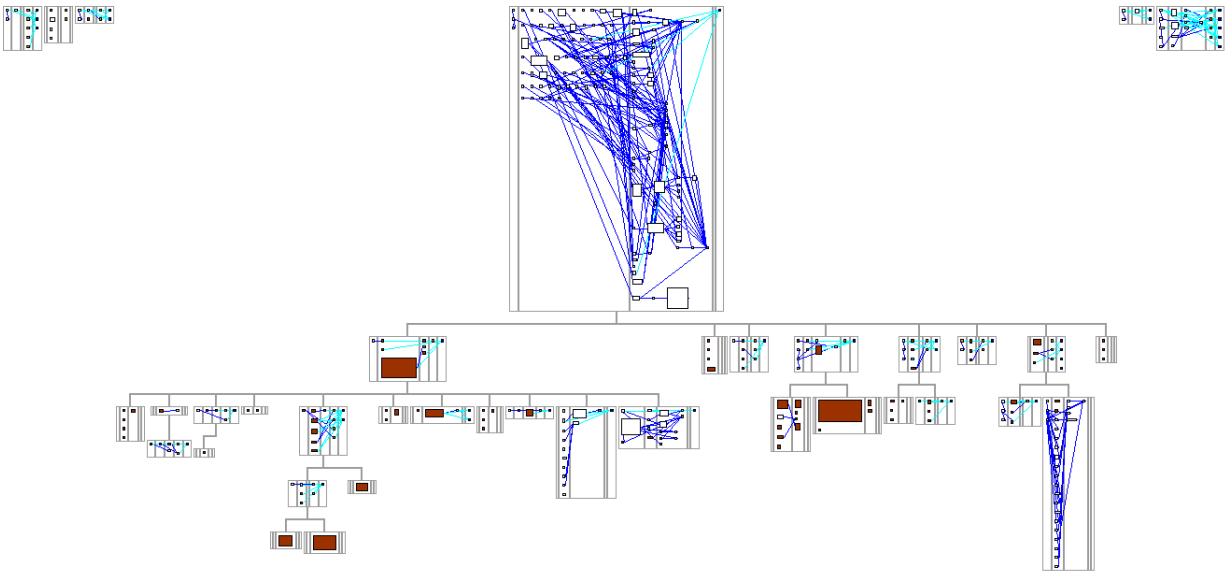

Figure 3: Blueprint Complexity of PetitParser packages

found 5 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 squares 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**

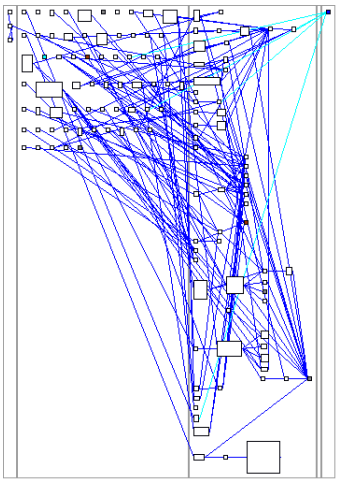

Figure 4: Blueprint Complexity of PetitParser class

In the schema, we can see more clearly the shape representing PETITPARSER class.
Based on the legend explained above, we found that it contains no accessors and one attribute, PROPERTIES which is a instance variable. Some methods, public or private, access directly to this attribute. In a theoretical viewpoint, it's not good to access directly, you should use accessors. With accessors, you can validate informations before setting your attributes. Of course, it's less important for private methods if you are very careful when you use an attribute. But, for public methods, you must use an accessor. Of course, it's a theoretical viewpoint and we look all methods accessing directly the attribute before getting more analysis. We 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 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 changes 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 gives the value associated to aKey if aKey is a key in property and answers the result of evaluating aBlock otherwise. Like the first method, we see that the integrity of the attribute is checked ;
- postCopy : this public method gives 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 tests 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 addition of 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 all methods 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 creates 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, even if we found no getter and setter, the most methods accessing directly the attributes 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 sign that the code is modular and it reduces the probability of code duplication. It is also the consequences of the PETITPARSER system : it uses structures with children and list of object. To access these structures, methods use these sub-structures and methods related to them.
Finally, we see some squares coloured in brown. It means that the method overrides code. Since we pointed out in the first report that PETITPARSER has a good use of inheritance, it seems quite logic.

## 2.4 Duplication side-by-side

What is dis fuk ??

## 2.5 Bad smells

With the Moose tools, we can compute metrics about almost everything but they are not all relevant to detect bad smells. To detect more precisely some bad smells, we wrote some queries.

### Number of lines of Code

First, we want to see methods where the number of lines of code is bigger than 50, we can execute this 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.

### 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 two accessor 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. Actually, when we browse the code, we see that this class has one getter, one setter and one method to show the attribute because it corresponds to a message with a specific format which is 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 directly

attributes and not use accessors. Particularly, in the class PPFailingParser, we have three accessors and some methods accessing directly the attribute.

At the opposite of the previous query, we can list 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

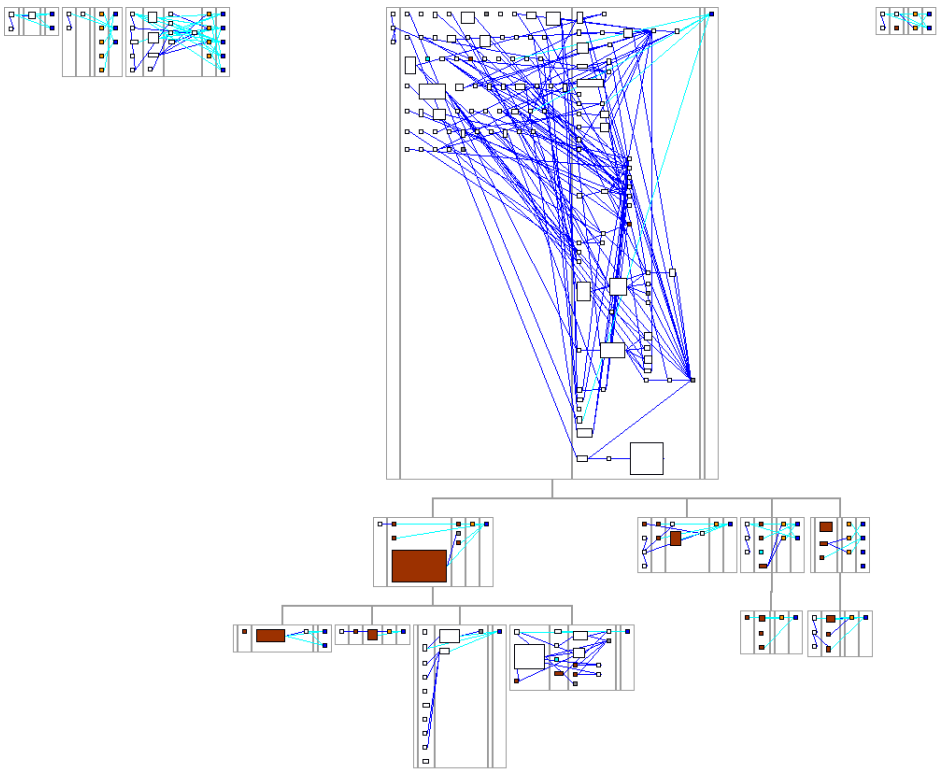

Figure 7: Blueprint of classes with less accessors than 2* attributes

rectangle, we have already analysed this in a previous section because it represents the PPParser class. For the other classes, we see that some of them have a lot of light blue links, like the rectangle on the left of PPParser. It is the PPToken class every methods (private or public) accesses directly to attributes. When we analyse the code of this class, we see that almost all methods just read attributes but when there are accessors to do this job. It's also not very good for maintenance. If you change your attribute (other type or the way you store key/value), you must change the code for alls methods accessing directly to this attribute. According to this blueprint schema, we see that it will be a big work.

### High parameter number

An other interesting metric is the number of parameters for every method and, in particular, finding methods with a high number of them. This is a very interesting metric because methods receiving a lot of parameters can be difficult to read and we can assume that lot parameters indicates a method performing too much work. We can also assume that all parameters can be grouped in a larger object, where all variables needed are stored. Using an object rather than parameters is very useful if you change the way you represent parameters in this object. If it is only passed in parameter, you must change more than if you use an object.
To find methods with a lot of parameters, we used this query :

```
each numberOfParameters > aNumber
```

Because the number of parameters can vary depending on the program, we have performed this query for aNumber form 0 until we have no methods. We found that 6 is the highest parameters number.

| aNumber | Number of methods |
|---|---|
| 0 | 2216 |
| 1 | 472 |
| 2 | 105 |
| 3 | 32 |
| 4 | 10 |
| 5 | 3 |
| 6 | 1 |

We clearly see that the number of methods decreases when we increase the number of parameters. It is also very interesting to see that the highest parameter number is 6, a small number. To go deeper in our measurements, we analysed the code of the method with 6 parameters : "matchList" in the PetitAnalyzer package from PPParser.
The header of this method is :

```
matchList: matchList index: matchIndex against: parserList index:
    parserIndex inContext: aDictionary seen: aSet
```

As said before, this is not very readable. Also, we found a long method (35 lines) with no comments which does not help in understanding. A manual analysis of the code explains us that it is a method testing if two list match. So, we could group parameters about lists in one object. With this solution, we can reduce the parameters number but mainly increase the cohesion in adding method directly in the object. It is also more efficient when you want to use this method : with an object, you give just the object, you don't have to pass each parameter and try to find every parameter before using it. It is very relevant because we found that this method is accessed in 51 methods.

**Brain and god classes**

A brain method or class is a method or a class that do too much work. In general, we try to avoid such methods in programs because such methods are very specific, impossible to reuse and they are very difficult to maintain. In general, it's the sign that your program is not very modular and you can improve your program by splitting these methods in several smaller methods.
To find these methods or classes, we can search for long methods but it's not enough. We search also methods with a lot of conditionals.

```
each numberOfConditionals > aNumber
```

Again, aNumber is a variable and we can adjust it to perform better analysis. If we take 5, we found 5 methods, whose the MATCHLIST method and PARSEON method. It's not a suprise because we have found this method in the large emthods metrics and in the high parameters number metric.
In this 5 methods, we found 3 methods from PETITPARSER package. According to the literature, we can qualify a class a "god class" if it is a complex class with low cohesion and with a high number of outer class accesses. To detect god class, we can use a query but each property of a god class has still been analysed. So, we can say that PPParser is a god class.

## 2.6 CityMap


# 3 Conclusion