ECOLE POLYTECHNIQUE DE LOUVAIN

LINGI2252 - SOFTWARE ENGINEERING : MEASURES AND MAINTENANCE

Assignement 1

Professor :

KIM MENS

Students : (Group 6)

Benoît BAUFAYS 22200900

Julien COLMONTS 41630800

Program :

SINF21MS

Academic Year 2013-2014

1 PetitParser System

PetitParser is a parsing framework developped by Lukas Renggli. The goal of a parser is to build a data structure with data given as input. This framework is able to parse multiple languages like Java, MSE or Smalltalk. Anyone can specify easily other custom parsers. Since PetitParser is a framework, you add code to make it able to parse other languages. PetitParser is better than other parsers because it uses a combination of four parsing techniques which are :

- Scannerless Parsers
- Parser Combinators
- Parsing Expression Grammars
- Packrat Parsers

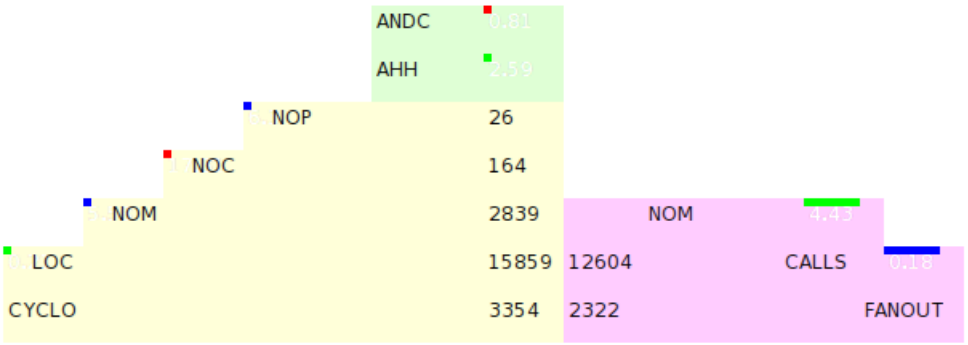
These different methodologies make PetitParser more adapted to the dynamic nature of Smalltalk. It's integrated into the Moose project. PetitParser packages are already imported into the image given for this course. If you need to import it in another image, you have to evaluate the following expression:

```
Gofer new
    renggli: 'petit';
    package: 'PetitParser';
    package: 'PetitTests';
    load.
```

2 Packages content

PetitParser contains approximately 300 classes defining 2839 methods. It is made of a big abstract core (package name is PetitParser-Core) and different concrete implementation layers. There are packages for each parsed languages split in a core package, an object package and a test package.

Three more special packages are implemented for the analyser part, writing tests and a graphical interface package used to write complex grammars.



This pyramid shows an overview of the model classes. With more the 15 thousands lines of code, PetitParser is quite a large framework.

3 Example of :

3.1 Object oriented programming style and techniques

Inheritance is used a lot in PetitParser, it's used more in packages core parts. For instance, there are a lot of parsers extending the main parser object called PPPARSER.

The use of inheritance seems complete : some methods are defined in an abstract way. For instance :

```
addSubclass: aClass
        "Ignored -- necessary to support disjoint class hierarchies"
```

We can still say that some classes seems not to be useful for the framework. For instance, the class PPSentinel (from PetitAnalyser-core) contains no messages and no variables. It extends PPEpsilonParser from PetitParser-Parsers package. There are no explanations for the existence of this class. We can make the hypothesis that it's there for further development but a comment saying why it exists should be written.

3.2 Adherence to accepted best practice patterns

Naming conventions are mainly respected in the all code :

- Variables : instance variable names are related to their role. A small description of what's the variable should contain is often available in the accessors. Parameters have a name related to their type. For instance :

```
initializeMessage: aString at: anInteger
        message := aString.
        position := anInteger
```

- Methods : method names seems to be respected, mainly accessors methods. For instance, accessors for nodes in the Java package :

```
expression
        ^ expression
expression: anExpression
        expression := anExpression
```

  Most of the methods are properly categorized. We can observe some uncategorized methods in PJFloatLiteralNode and PJCharacterLiteralNode classes. But they contain only accessors. We can also say that parameter names suggest a type. For instance :

```
PPMomento: position: anInteger
    position := anInteger (bon car suggestion type name)
```

- Classes : All class names are properly related to the object they represent. Hierarchy seems to be used in a good way. For instance, in the Analyser package, a rule can be divided in two subclasses : a replacing rule or a searching rule. Names are mostly properly written to understand hierarchy without reading any comment.

3.3 Bad smells in code

Code duplication

Duplicated method appears in PetitParser-Parsers package:

```
PPParser:firstSet
    "Answer the first-set of the receiver. Note, this
      implementation is inefficient when called on different
      receivers of the same grammar, instead use #firstSets to
      calculate the first-sets at once."
    ^ self firstSets at: self
```

This method is completely useless. It just call another one existing in the same package. Probably, it comes from development improvement. This method should bse deleted and every calls for it should be changed to `firstSets`.

Long parameters list

Some methods are related to this bad smell. For instance, method matchList in PParser class and RBProgramNode, both in PetitSmalltalk package, have respectively six and five parameters. This methods are both recursive and contained in an abstract context. Some of the submethods used have to be implemented in the subclass that will call matchList. They are also reported as too long. Since the code of this methods isn't easy to understand and there are no comment, it's hard to know what they actually do. With the recursive behaviour and just a few if conditions, it's probably hard to implement these methods in an other way. With more informations about them (comments), perhaps we could divide them into steps that would be easier to understand. Since no comments should be needed to get what the code do, these methods seems not to be properly implemented.

Long Methods

Quite a few number of methods are too long in PetitParser's packages. A large part of them can be explained as a normal way of writing code. Since PetitParser have to deal with programming languages, it has to know the complete syntax for each of them. For every languages, we have methods or number of class variables that could seems too long but it just contains all the words used in that language.

For instance, in the package PetitSQLite-Parser, for the PetitSQLiteGrammer class, the column-type method is reported as too long. Actually, it just contains all kind of columns existing for SQLite. This method cannot be split in parts. It would be even worse to divide it in multiple methods.

Another useful example is shown in PetitJava-Core package with a very large number of class variables. This number seems huge. But it is related to the large lexicon of Java. Maybe, it can be split into different classes sorted by kind of written line (Statements, declaration, expression,etc.). The large number of methods contained in this class could be also reduced in the same way.

In PetitParser-Parsers package, the PPDelegateParser class contains a long method named `morphicShapeSeen`. This method contains a lot of informations about displaying information such as colors, background, width and line spaces. It would be difficult to divide it in multiple parts. The same problem appears in PPChoiceParser class.

In the same package, the PPLimitedRepeatingParser class contains only one method which is too long. The ParseOn method appears in all parsers class, it is a subclass responsability from PParser higher class. In this case, there are two loops, their work could be done in a subroutine.

4 Framework

Since PetitParser is already a framework, we can't discuss how getting this project into a framework. All over the code, it is easily observable that this project is made to be a framework. A clear split-up is implement between the core part of the parsers and the different strategies related

to them and the languages that parsers have to understand. Adding more languages to PetitParser is a work that could be done without changing anything in the code already implemented.

5 First impression

Most of the code seems to respect a large number of rules studied in this course. The project seems to be properly thought. Implementing parsers with multiple strategies is a hard work and so it is to read it without taking part into the development. We observed a lack of comment on the code. There are explanations for main classes and kind of nodes but not many for strategies and long methods using loops. When a large number of variables are changed, even with good names, it isn't easy to get the goal that has to be reached.

They are a large number of tests. It is good for the application and for further development. People who want to use PetitParser framework can see how to write some of them for their own grammars or languages. The problem with the tests is still a lack of comments. We didn't get what a large number of them actually did. Some of them are written but aren't computed anymore. For instance:

```
testCharacterLiteral6
        "not clear how this must be supported
        (see http://java.sun.com/docs/books/jls/third_edition/
           html/lexical.html#3.10.6)"

        "self
                parse: '''\u03a9'''
                rule: #characterLiteral"
```

6 Conclusion

This project was helpful to see what is really important in developing large applications. The rules that have to be respected seems really simple at first sight but they are very useful to read the code

for further development. We also saw that checking packages without tools is a long and hard work. For a framework of this size, it could be possible to make a complete analysis but it would take a very long time. If we had a larger one, it would be almost impossible. We really get the interest of using tools.

The subject of the framework was very interesting for us. Parsers are used on a large number of applications and, even if it was hard to understand everything in the framework, we learned the basics.

For beginners in the SmallTalk language, reading a relatively large code, written by professionals, wasn't easy. We saw the advantages of this language in an application made expressively for his dynamic behaviour.