Internship report

École Polytechnique de Montréal, Québec, from July 19th to August 31st, 2017

Implementing Linguistic Antipatterns Detection Algorithms and Their Integration in a Popular Code Checking Tool

Author :
M. Jean RUBLÉ

Professors :
Pr. Yann-Gaël GUÉHÉNEUC
Pr. Foutse KHOMH

Acknowledgements

I would like to express my appreciations to both professors who gratefully welcomed and led me during this internship, Mr. Yann-Gaël GUÉHÉNEUC and Mr. Foutse KHOMH.

Contents

Introduction **2**

1 Project presentation **3**

1.1 Environment presentation 3

1.1.1 L'École Polytechnique de Montréal 3

1.1.2 Department of IT genius and software engineering 3

1.1.3 Ptidej team 3

1.2 Why I choose this internship 4

1.3 Presentation of the project context 4

1.3.1 Antipatterns and Linguistics Antipatterns 4

1.3.2 PMD 4

1.3.3 PMD plugin for Eclipse 6

1.3.4 WordNet 7

1.3.5 Stanford Parser 7

2 Presentation of my work during the internship **10**

2.1 Objectives 10

2.2 The beginnings 10

2.3 Implementation 12

2.3.1 CustomDictionary and CustomParser classes 12

2.3.2 Splitter class 12

2.3.3 Classes implementing rules 12

2.4 UML diagram 13

2.5 The internship benefits 13

Conclusion **15**

Acronyms

AST *Abstract Syntax Tree*

LA *Linguistic Antipattern*

LAPD *Linguistic Antipattern Detector*

POS *Part of speech*

DIG *Department of IT Genius*

CSE *Computer Science & Software Engineering*

PolyMORSE *POLYtechnique MOntréal Researchers in Software Engineering*

SP *Stanford Parser*

Useful links

PMD website:	`https://pmd.github.io/`
WordNet:	`https://wordnet.princeton.edu/`
Stanford Parser :	`https://nlp.stanford.edu/software/lex-parser.shtml`
Ptidej team website:	`http://www.ptidej.net/`
PolyMORSE :	`http://plow.soccerlab.polymtl.ca/`

Introduction

From the 19th of June to the 31st of August I worked as an intern in École Poytechnique, in Montréal, Québec. I was under the supervision of Mr. Yann-Gaël GUÉHÉNEUC, a full professor of the Canada research chair on Patterns in Mixed-languages Systems, and Mr. Foutse KHOMH, an Associate Professor and researcher in the Software Analytics and Technologies (SWAT) lab of École Polytechnique.

During my internship, I had the opportunity to work on *antipatterns* and *linguistic antipatterns* and to code Java rules for a code checking tool, **PMD**.

Beyond enriching my knowledges in computer sciences, this internship gave me the chance to work in a renowned IT department and to better understand the work of a researcher.

My main task was to implement PMD rules to detect linguistic antipatterns in code. I had a set of rules to implement and I had to check if all user cases were verified by the rules.

It has truely been an enlightening experience in a dynamic environment, and in a beautiful city.

Chapter 1

Project presentation

1.1 Environment presentation

1.1.1 L'École Polytechnique de Montréal

L'École Polytechnique de Montréal, founded in 1873, is an engineering school affiliated with Université de Montréal. It ranks first in Canada for the scope of its engineering research. The school offers graduate and postgraduate training, and is very active in research. The room which I worked in was located in the *Pierre-Lassonde et Claudette-Mackay-Lassonde* building, home to the Electrical Engineering Department and Computer and Software Engineering Department, which was inaugurated in September 2005.

1.1.2 Department of IT genius and software engineering

The DIG was founded June 1st, 2001 by its separation to the Electric Genius Department. The department counts many labs specialized in virtual reality, constraint programming, and the Ptidej laboratory in which I worked.

1.1.3 Ptidej team

The Ptidej lab is part of the CSE department of Concordia University. It is also part of the PolyMORSE group at Polytechnique DIG. The team develops theories, methods and tools to understand, evaluate and improve design patterns.

1.2 Why I choose this internship

During my first year at ENSIIE, I made my internship at Viareport, a french enterprise specialized in financial consolidation. It has been a great experience in a medium-size french company. I wanted for my second year to be in a foreign research lab, to have an idea of the researcher work and also to discover another country and of course another culture and to practice english and validate 8 weeks abroad.

1.3 Presentation of the project context

1.3.1 Antipatterns and Linguistics Antipatterns

During my internship, I discovered patterns and antipatterns, and more precisely linguistic antipatterns. Code patterns are models or structures that occur repeatedly in object-oriented design. They have been formalized, and are generally considered a good development practice. On the other hand, antipatterns describe common mistakes and developers issues that can cause a software project to fail. Among those antipatterns, there are linguistic antipatterns. Linguistic antipatterns have been defined by Mrs Venera ARNAOUDOVA in her thesis [Arnaoudova 2014]. She described more than 17 possible LAs such as "*set method returns*", for example. This LA occurs when a developer creates a set method that does more than only setting a parameter and returns something (by convention in object-oriented programming, a set method should only set a parameter to a value and return void). Venera also created a version of LA detector for **Checkstyle**, another code checking tool. I do not want to plagiarize her work, so I added a reference to her work in the bibliography and I also added her thesis to this report. **The work about antipatterns can be found at chapter 6**. I also added a folder with my source codes of this project and I recommand to read the chapter 6 of the thesis to understand it.

1.3.2 PMD

Presentation

PMD is an open source static Java code source analyzer. It reports bad programming practices, or code that can reduce performance, such as high cyclomatic complexity (the cyclomatic complexity gives the number of independent paths within. For example, an if statement creates two new independent paths: one for the *true* value and another for the *false* one). PMD also detects copy-pasted code, source of recurrent errors. It has several plug-ins for JDE like Eclipse, jEdit.. During my internship I used the Eclipse plugin. To use it, built-in rule sets must be selected, such as *Dead code* (unused local variables, param-

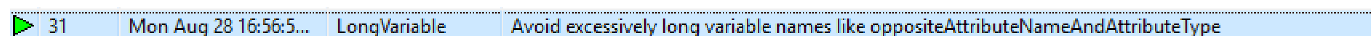
eters and private methods) or *Over-complicated expressions* (unnecessary if statements, for loops that could be while loops), and several more others. Then, custom rules can be wrote and used with Eclipse.
PMD uses an **Abstract Syntax Tree** to check code mistakes. An AST is a tree representation of the syntactic structure of the source code. Each node of the tree is one of the source elements representation. During the compilation, each rule traverses the AST and checks for problems. Then a report compiling rule violations is printed in the console or under a formatted document.

Figure 1.1: PMD logo

PMD use case:
here we use a variable with a very long name:

```
private boolean oppositeAttributeNameAndAttributeType;
```

and this is the result of rule violation on the PMD report:



PMD rules writing process

Writing PMD custom rules can be done by two ways: using XPath or Java.

XPath rules: XPath is a query language for selecting nodes from an XML document. It is based on the AST of the XML document. An XPath expression has a location path consisting of a sequence of an *axis*, a *node test* and *predicates*. Axis specifiers indicate navigation direction within the AST. The main axes available are attribute indicated with @, descendant or self //, parent .., and self . As an example of using the attribute axis in abbreviated syntax, //a/@href selects the attribute called href in a elements anywhere in the document tree. Writing PMD rules is easy and very effective since XPath is based on the AST. It offers good performances, but the available tasks are limited. To create more powerful and customized rules, we use Java.

Java rules: to write Java rules for PMD, the developer has to figure out which entity he wants to spot. As an example, if the problem occurs for a while loop, we want to use the *ASTWhileStatement* object.

```
public Object visit(ASTWhileStatement node, Object data) {
    System.out.println("hello world");
    return data;}
```

Then, we use our custom rule with this code test:

```
public void doSomething() {
  while (true)
      x++;
}
```

At the end we should have an "hello world" in command line because PMD traverses the AST recursively and performs a calleback to our rule whenever it finds a WhileStatement.

XML Ruleset file

Once the custom rule is ready, we must add it to a ruleset that will specify to PMD the new rule. The rule name, a message (for the user), a Java class implementing the rule, and eventually a description and usage example.

1.3.3 PMD plugin for Eclipse

I worked with the *Neon* version of the Eclipse project and used the PMD plugin for this IDE. Using custom rules with this IDE can be summarized in six main steps:
- Create a plugin fragment as an extension to the PMD plugin
- Develop the rules classes implementations
- Create the rule sets files
- Test the fragment. The simplest way is to launch a test Eclipse environment. This will launch a secondary workbench with all the installed plug-ins and the fragment currently in development. In that workbench, we create a test Java project to test the rule set.
- Package the fragment
- Install the fragment in an Eclipse environment.

1.3.4 WordNet

WordNet is a large database of English created in the Cognitive Science Laboratory of Princeton University. Nouns, verbs, adjectives and adverbs are grouped into sets of cognitive synonyms (synsets), each expressing a distinct concept. It is similar to a dictionary. Each of WordNet's 117 000 synsets is linked to other synsets by means of a small number of "conceptual relations.". The most frequently encoded relation among synsets is the super-subordinate relation (also called hyperonymy, hyponymy or ISA relation). It links more general synsets like furniture, piece_of_furniture to increasingly specific ones like bed and bunkbed. Thus, WordNet states that the category furniture includes bed, which in turn includes bunkbed; conversely, concepts like bed and bunkbed make up the category furniture. All noun hierarchies ultimately go up the root node entity. Wordnet has limitations as it does not include information about the etymology or the pronunciation of words and it contains only limited information about usage. WordNet aims to cover most of everyday English and does not include much domain-specific terminology.

Figure 1.2: Screen shot of WordNet query result for "Hamburger"

1.3.5 Stanford Parser

The Stanford Parser is a natural language parser. It can detect which words go together (as phrases) and which words are the subject or object. As an example to describe the usage of the Stanford Parser, we will use this sentence:
The strongest rain ever recorded in India shut down the financial hub of Mumbai.

The output gives us:

```
    The/DT strongest/JJS rain/NN ever/RB recorded/VBN in/IN India/NNP
shut/VBD down/RP the/DT financial/JJ hub/NN of/IN Mumbai/NNP ,/,
officials/NNS said/VBD today/NN ./.

(ROOT
  (S
    (S
      (NP
        (NP (DT The) (JJS strongest) (NN rain))
        (VP
          (ADVP (RB ever))
          (VBN recorded)
          (PP (IN in)
            (NP (NNP India)))))
      (VP
        (VP (VBD shut)
          (PRT (RP down))
          (NP
            (NP (DT the) (JJ financial) (NN hub))
            (PP (IN of)
              (NP (NNP Mumbai)))))
    (, ,)
    (NP (NNS officials))
    (VP (VBD said)
      (NP-TMP (NN today)))
    (. .)))

det(rain-3, The-1)
amod(rain-3, strongest-2)
nsubj(shut-8, rain-3)
nsubj(snapped-16, rain-3)
nsubj(closed-20, rain-3)
nsubj(forced-23, rain-3)
advmod(recorded-5, ever-4)
partmod(rain-3, recorded-5)
prep_in(recorded-5, India-7)
ccomp(said-40, shut-8)
prt(shut-8, down-9)
det(hub-12, the-10)
amod(hub-12, financial-11)
```

```
dobj(shut-8, hub-12)
prep_of(hub-12, Mumbai-14)
conj_and(shut-8, snapped-16)
ccomp(said-40, snapped-16)
prep_during(walk-33, night-37)
nsubj(said-40, officials-39)
root(ROOT-0, said-40)
tmod(said-40, today-41)
```

The first part of the output gives the POS-tagged sentence. In English grammar, a part of speech (abbreviated form: PoS or POS) is a category of words (or, more generally, of lexical items) which have similar grammatical properties: noun, verb, adjective, adverb, pronoun, preposition, conjunction, interjection, article. Then the output gives a context-free phrase structure grammar representation and finally a typed dependency representation.

Chapter 2

Presentation of my work during the internship

2.1 Objectives

My main task during this internship was to implement a fragment for the PMD plugin of Eclipse to detect the linguistic antipatterns. I had to develop Java rules to deal with :
- LAs in methods,
- LAs in variables and parameters,
- LAs in documentation.
I had to use the Stanford Parser and Wordnet. Wordnet gave the meanings of the words in documentation and in methods signatures, and the relations between the words (antonyms or synonyms..) whereas the Stanford Parser was useful to have the POS of the words and to know what grammar type of word we were dealing with. So the first steps were to implement classes that would permit a communication between PMD and the SP and WordNet. But first of all, I had to know how to implement effective rules. My first PMD rules were written with XPath, but like I said in the first part XPath does not permit to implement a lot of functions. When I have been used to XPath, I decided to begin to write Java rules.

2.2 The beginnings

The process of writing rules is quiet simple. As we saw it before, we spot one AST object that we want to visit and to do that we use the *visit* method. First of all we declare a class that will represent our rule and that extends the *AbstractJavaRule* class or whatever other PMD object we want. So when I began to write my rules I first created one class

for each LA I wanted to detect, and for each rule I decided to visit the AST object *ASTClassOrInterfaceDeclaration* that is as we can guess the AST object for the classes or interfaces. To use the methods of the class I was visiting, I created a list of the class methods and for each method I was examinating I added a condition and a violation. So for the LA "“Is” returns more than a Boolean" for example, the rule looked a bit like this:

```
public class IsReturnsMoreThanABooleanRule extends AbstractJavaRule{
    public Object visit(ASTClassOrInterfaceDeclaration parent, Object data){
        List<ASTMethodDeclaration> methods = parent.getDescendant(
        ASTMethodDeclaration).class;
        for (ASTMethodDeclaration method : methods){
            if (detectIsReturnsMoreThanABoolean(method)){
                addViolation();
                }
            }
        return data;
        }
    }
```

and after this part of the code I created the "detectIsReturnsMoreThanABoolean" method that would give me a boolean verifying if the method in the for loop was in fact a Is method that returns more than a boolean. As one can imagine, this approach was really inefficient because:
- PMD traverses several times the same AST node, and the biggest one (the *ASTClassOrInterfaceDeclaration*). It traverses it one time for each rule.
- Furthermore the PMD documentation specifies that "Rules which use the RuleChain to visit the AST are faster than rules which perform manual visitation of the AST. The difference is small for an individual Java rule, but when running 100s of rules, it is measurable. For XPath rules, the difference is extremely noticeable due to Jaxen overhead for AST navigation." So as we perform a manual visitation, it will be less effective.
- We are creating a list of AST methods every time. It takes a lot of memory and it is extremely ineffective.
So, that was how I used to work at the beginning. Plus I had a lot of code and functions that I was using for several rules and that I had to write every time, it was a lot of time and performance.

2.3 Implementation

2.3.1 CustomDictionary and CustomParser classes

When I got used to Java rules writing process and fluent with PMD, I decided to add the SP and WordNet to the project. So I created two different classes, one for the dictionary and one for the parser. Both have a private constructor and a public method getInstance() to apply the singleton pattern. For the parser class, we have different methods to get the POS of the words in a sentence under different forms: if we have a plain text sentence in a *String* or a more sophisticated form under a *Vector*. When we use vector, parsing process is faster. I tested it with JUnit and every time I could verify that there was more than 500 ms difference between both methods. The CustomDictionary class provides methods to transform a String word into an *IndexWord* understandable for WordNet. Then it finds relations between those objects in other methods.
With those two classes I had finally a set of methods to use that would help me finding meanings of terms and relations between them. But the parser only works with readable phrases, it cannot figure out phrases with no spaces between words. It is also not sensitive to the case. So how to deal with methods ? We want the parser to know that the method name "isValid" has to be read as "Is valid". So I created a *Splitter* class that splits the terms at a precise point.

2.3.2 Splitter class

In Splitter class we can find different methods to deal with method names, with commentaries, and variables. The *methodSplitter* only splits string on upper case to lower case. *variableSplitter* splits on upper case to lower case and "_". We assume that method names and variables are written under the right conventions (otherwise PMD will detect it and the user will correct it. Also, there is a very low probability for a programmer to write a variable name with "!" or "?" inside). The *commentSplitter* splits on special chars thanks to a special char regex pattern. With the Splitter, I was able to split into an understandable sentence every method name, variable name and comment.

2.3.3 Classes implementing rules

To improve performance I grouped all the rules under two different files. The *Abstract-LAPDRule* class is an abstract class that provides several useful methods that are used by the two other classes. As an example, the *getNodeType* method permits to get the literal type from a variable or a method depending on what AST node we are using. Then the *AttributesLAPDRule* class uses the method with an *ASTVariableDeclarator* and the

MethodsLAPDRule uses it with an *ASTMethodDeclaration*. So, for both files, we have a visit method that traverses either the AST attributes or variables for the attributes rule and the methods nodes for the methods rule.

2.4 UML diagram

Please see page 14

2.5 The internship benefits

I learned a lot of things during this internship. First, I improved my skills in Java, I learned a lot of programming techniques and good practices to gain performance, such as patterns. I also learned a lot about code checking tools and how they can be really useful in the way that they correct bad practices and sometimes can prevent bugs and errors in the code. I also learned how to include tools such as a parser or a dictionary in a Java project and I will use this knowledge in my future works. Furthermore, I had a lot of difficulties that I overcame by finding new programming possibilities that I will use again in the future. The first difficulty I had was to understand how PMD works. In a few weeks I could create complex Java rules. Second difficulty was to create an entry point for the SP and WordNet and I found how to resolve it by using the singleton pattern and create simple methods. I also had to improve performance in a way that many users can use my plugin inside their projects, it is not possible for my code to take too much time before having an output. The biggest time consumer in my program is the Stanford Parser, the less time it can take is approximately three seconds for one sentence. So I had to design the other tasks to be extremely fast so that they do not make all the program too slow.

Moreover, I had the opportunity to work on a very concrete project, that has real applications and that will permit other programmers to learn about linguistic antipatterns and to produce a code of a better quality. Indeed, I submitted my work to PMD in a pull request, unfortunately (but also as expected) they did not favorably reply, but once I would have corrected it I hope my work will be enjoyed by many people around the world.

Finally, it has been such a great experience in a prestigious research lab, I discovered the job of researcher and the pros and cons of that job. I found out it can be really enjoyable in case of success, but also terrible when one does not find a solution and stays at the same point for days.

<<Java Class>>
CustomDictionary
utils
d: Dictionary
CONFIG_FILE_PATH: String
CustomDictionary()
getInstance():CustomDictionary
getD():Dictionary
areSynonyms(IndexWord,IndexWord):boolean
areAntonyms(IndexWord,IndexWord):boolean
haveAntonyms(IndexWordSet,IndexWordSet):boolean
stringToIndexWordSet(String):IndexWordSet
taggerToDictionaryPos(String):POS
hasIndexWordForPos(String,POS):boolean
-cd
0..1
<<Java Class>>
AbstractLAPDRule
rules
collectionKeywords: String[]
returningKeywords: String[]
ignoreForAntonymsKeywords: String[]
validatingKeywords: String[]
ignoreForConditionKeywords: String[]
aggregationKeywords: String[]
predicateKeywords: String[]
COLLECTION_KEYWORDS: StringMultiProperty
RETURNING_KEYWORDS: StringMultiProperty
IGNORE_FOR_ANTONYMS_KEYWORDS: StringMultiProperty
VALIDATING_KEYWORDS: StringMultiProperty
IGNORE_FOR_CONDITION_KEYWORDS: StringMultiProperty
AGGREGATION_KEYWORDS: StringMultiProperty
PREDICATE_KEYWORDS: StringMultiProperty
SIGNATURE: String
<clinit>():void
AbstractLAPDRule()
init():void
start(RuleContext):void
isCollection(String):boolean
containsCollectionKeyword(String,boolean):boolean
isReturning(String):boolean
shouldBeIgnoredFromAntonyms(String):boolean
isAggregation(String):boolean
isCommentedCode(String):boolean
getNodeType(Node):String
getNodeComments(Node,List<Comment>):Vector<Comment>
toString(List<Comment>):String
<<Java Class>>
CustomParser
utils
lp: LexicalizedParser
tokenizerFactory: TokenizerFactory<CoreLabel>
props: Properties
pipeline: StanfordCoreNLP
<clinit>():void
getInstance():CustomParser
CustomParser()
getTree(String):Tree
getTree(Vector<String>):Tree
hasPOS(String,String):boolean
hasPOS(String,Vector<String>):boolean
hasPOSInAnyForm(String,String):boolean
hasPOSInAnyForm(String,Vector<String>):boolean
getAllPOS(String):String
getPOS(String,String):String
getPOS(String,Vector<String>):String
-cp
0..1
<<Java Class>>
Splitter
utils
UPPER_CASE: String
SPECIAL_CHARS: Pattern
<clinit>():void
Splitter()
methodSplitter(String):Vector<String>
variableSplitter(String):Vector<String>
commentSplitter(String):Vector<String>
toString(Vector<String>):String
<<Java Class>>
AttributesLAPDRule
rules
collectionTypeSingleNameAttribute: boolean
booleanAttributeNameNotType: boolean
singleTypeCollectionNameAttribute: boolean
oppositeAttributeNameAndAttributeType: boolean
COLLECTION_TYPE_SINGLE_NAME_ATTRIBUTE_DESCRIPTOR: BooleanProperty
BOOLEAN_ATTRIBUTE_NAME_NOT_TYPE: BooleanProperty
SINGLE_TYPE_COLLECTION_NAME_ATTRIBUTE: BooleanProperty
OPPOSITE_ATTRIBUTE_NAME_AND_ATTRIBUTE_TYPE: BooleanProperty
<clinit>():void
AttributesLAPDRule()
start(RuleContext):void
getVariableComments(ASTVariableDeclarator,List<Comment>):Vector<Comment>
getVariableType(ASTVariableDeclarator):String
visit(ASTVariableDeclarator,Object):Object
detectCollectionTypeSingleNameAttribute(String,String):boolean
detectBooleanAttributeNameNotType(String,String):boolean
detectSingleTypeCollectionNameAttribute(String,String):boolean
detectOppositeAttributeNameAndAttributeType(String,String):List<String>
detectOppositeCommentAndAttributeSignature(ASTVariableDeclarator,String,String,List<Comment>):List<String>
<<Java Class>>
MethodsLAPDRule
rules
MethodsLAPDRule()
getMethodType(ASTMethodDeclaration):String
getNameFromPrefix(ASTPrimaryPrefix):String
getMethodComments(ASTMethodDeclaration,List<Comment>):Vector<Comment>
getCalledMethodsNames(ASTMethodDeclaration):List<String>
visit(ASTMethodDeclaration,Object):Object
detectFalseBooleanMethod(String,String):boolean
detectNotImplementedCondition(ASTMethodDeclaration,String,String,List<Comment>):String
detectIsMethodNotBoolean(String,String):boolean
detectSetMethodReturns(ASTMethodDeclaration,String,String):boolean
detectCollectionTypeSingleNameMethod(String,String):boolean
detectValidationMethodDoesNotConfirm(ASTMethodDeclaration,String,String):boolean
detectGetMethodDoesNotReturn(String,String):boolean
detectTransformMethodDoesNotreturn(String,String):boolean
detectSingleTypeCollectionNameMethod(String,String):boolean
detectGetMethodDoesNotReturnAttribute(ASTMethodDeclaration,String,String):boolean
detectOppositeMethodNameAndMethodType(String,String):List<String>
detectOppositeCommentAndMethodSignature(ASTMethodDeclaration,String,String,List<Comment>):List<String>

Conclusion and perspectives

As a conclusion, I would like to thank again my professors, it has truely been such an honor to meet them and work under their supervision.
This internship was really interesting. However, I am not entirely done with it because I aboslutely want my pull request to be accepted by the PMD team, so I have still a lot of work to do. Plus, Foutse included me in the team working on a new project linked with my work. It is based on neuronal network, the goal is to implement a tool simila to my plugin but far more intelligent since it will learn with the data it stores. The neuronal network is expected to exponentially divide the execution time, but it should also consume far more memory. That is why I am really enthusiastic about this project and grateful.

Bibliography

[Arnaoudova 2014] V. Arnaoudova. *Towards improving the code lexicon and its consistency.* PhD thesis, Polytechnique Montréal, 2014.