Université catholique de Louvain
École Polytechnique de Louvain
Pôle d'ingénierie informatique

# Extending Pythia with stuctural source code checks for Java

Author : Steven An
Promoters : Kim Mens
Sébastien Combéfis
Reader : Virginie Van den Schrieck

Master's thesis submitted for the
Master in Computer Science
option Artificial Intelligence

2013-2014

# Acknowledgements

I would like to thank my promoters Kim Mens and Sébastien Combéfis for their help, availability and constructive feedback through the whole year. Thanks to all my friends and family for their support.

# Table of contents

# Chapter 1

# Introduction

## 1.1 Context

Pythia is a platform that allows teachers to share their exercises online. The main idea behind it is to make projects and homework submissions as well as corrections automatic. Students can submit their work as soon as it is done and then it is forwarded to the teachers. Since the code is uploaded on the platform, it can automatically be checked by implemented checkers. Students can then correct their mistakes based on the provided feedback. In other words, when teachers create exercises, they can implement corresponding checks. Once students have worked on those exercises, they receive feedback that is generated automatically by those checks so that they can correct their code by themselves until no more errors are detected. To summarize, Pythia helps teachers as well as students by facilitating interactions between them. It is an interactive platform that assists beginning coders.

## 1.2 Problem

For now, Pythia incorporates mainly functional and performance checks meaning that it is able to check if the submitted code works as expected by the teacher and if its performance indicators such as the execution time are good enough. To improve the platform, we could also add some coding checks to detect bad coding styles. Providing such verification could help students to improve their style or to adopt some coding standards promoted by teachers. Making such automatic checks would lower the teachers' workload. Furthermore, it could also help beginners to realize the importance of coding style to improve readability and to respect coding conventions. In other words, the problem is the lack of coding style checks.

## 1.3 Motivation

Pythia is a growing project that will become more and more important for programming courses taught at UCL. It is evolving and is the intermediate agent in projects and homework processes. It would be very interesting to help improve Pythia. Learning how to code is not

an easy task for everyone at first sight and we all make mistakes that should be corrected as soon as possible when they occur. Functional checks are already implemented however there is a lack of checks for bad coding habits. Coding properly is really important to improve the readability and to match coding standards. Furthermore, coding style checks could detect common beginners errors that students can directly correct themselves. As explained in [1], schools and teachers tend to use a project-based learning approach. It means that students learn to write code by implementing projects. The correctness of students' solutions are only checked through the functionality of their implementation while the style is completely left behind. While it is not that bad for small exercises, problems could emerge on larger projects especially if students have to work in groups. A group member could noticed that he has a bad coding style if another member tells him he does not understand his code. However, since good coding style is not taught, the student could be unable to correct himself. The paper "*Tool Support for Learning Programming Style*" [2] also points out the importance of coding style for the maintainability of program. If a code is hard to read, it would require more time to be understood and so to perform a software maintenance.

## 1.4 Objectives

As suggested by the title of this work, the goal is to extend Pythia with structural coding checks that are not used yet by the platform. In other words, the purpose of this thesis is to assist students by providing them good coding checks through the Pythia platform. Since those checks will be used with an educational purpose in mind, it is crucial that they provide comprehensive help messages to show what went wrong. With the provided feedback, we expect students to understand basic mistakes in their code so that they can improve themselves. From a teacher's point of view, such checks provide some kind of processing that make the correction of students' code easier; since coding errors are detected, students can correct them before submitting their code to the teacher.

## 1.5 Approach

In order to achieve our goal, we decided to look for tools that already handle coding style checks and that we could use to integrate in Pythia. We found three Java tools that fits what we were looking for: Findbugs, Checkstyle and PMD which provide some checking. We chose Checkstyle and extended it so that we obtained a set of checks that we could use.

To validate our choice, we created three exercises. A first exercise was submitted to students to make use of that set and simulate the situation where our work would be added to Pythia. We also tried ourselves to do an exercise and used those checks. As a third exercise, we created a larger exercise and asked more experienced developers to resolve it.

Our approach is not the same as in [1]. The authors asked 50 students to do an exercise then they used multiple tools to provide feedback. Students discussed their results to see what went wrong. They were able to do several iterations with the same set of students. In our case, we did not have as much students and we could not ask them to do several iterations. So we made three distinct exercises and we also decide to focus on one tool. The work described

in [2] is also a motivation for us because the authors also used Checkstyle on two students' project and they noticed improvements in the number of errors detected. In fact, they made an experiment during two years. On the first year, they asked students to do a project. They then used Checkstyle on the students' solutions and gave them the generated feedback. Since there were a lot of mistakes, they decided to teach students some style conventions. The next year, students made a second project and Checkstyle was used as well. Again, the results showed improvements as the number of errors decreased.

## 1.6 Contributions

This thesis provides an overview of some selected Java tools: Findbugs, Checkstyle and PMD. In order to do so, we tried to understand how each of them works and compared their approaches. We have extracted implemented checks that are interesting for us and we have extended a tool in order to provide a set of checks that could be implemented on Pythia to assist Java beginners. As a validation, we have also evaluated how this helps students by providing an exercise with a set of coding checks. The main contribution of this work was to see how we could extend Pythia with some checks that might be interesting to use on such platform and how a tool like the three mentioned could be used.

## 1.7 Roadmap

We began this thesis by introducing the selected analysis tools and we explained how each of they works by providing detailed examples of implemented checks. We also talked about Pythia and what the platform can handle at the moment. Those introductions defined the background (Chapter 2) of our work. Then in the Chapter 3, Related work , we have had a look at other platform that are similar to Pythia so that we compared them, see what were the differences between them and we also identified what features do they share. After that, in the Chapter 4, we explained the problem we tried to answer. We also described what exactly are coding style checks compared to functional checks in the section 4.1. Using what we learned on analysis tools, we talked about their different approaches to handle the detections of coding flaws. Based on that, we implemented a solution to the problem that is described in the Chapter 5 (Implementation) and explained the integration on Pythia. The validation part in the Chapter 6 explained how we tried to evaluate the interest of integrating such checks. To do so, we have created a small exercise that was then submitted to students to see how they interact and we collected their feedback about our work. We also tried Checkstyle ourselves in an additional experiment. As a conclusion, we summarized all those steps in the Chapter 7 and we shared our final thoughts regarding our results.

# Chapter 2

# Background

Before talking about what has been done in this thesis, it is important to define the background of this work. What tools were used? How do they work? This chapter will answer those questions by introducing some analysis tools as well as Pythia, a platform used by teachers and students here at the UCL.

## 2.1 Analysis tools

The first part of this chapter will focus on three analysis tools that have been chosen for comparison: Findbugs, Checkstyle and PMD. Those are probably some of the most popular analysis tools used by Java programmers. To really understand how they work, an implemented example will be discussed for each of them.

### 2.1.1 Findbugs

#### Overview

As suggested by its name, Findbugs [3] [4] is a tool that checks Java code and uses predefined detectors to find patterns that can lead to bugs. It works on bytecode and because of that, understanding implemented bug patterns as well as extending Findbugs can be difficult for inexperienced programmers. This first tool focusses on potential bugs which means that it mostly looks for functional errors instead of bad coding style.

Errors detected by Findbugs are grouped in nine categories, including Bad Practices or Dodgy code (for the other categories, please consult the Findbugs website [3]).

Each of those bugs are also evaluated on a 1-20 scale: scariest (rank 1 to 4), scary (rank 5-9), troubling (rank 10-14) and of concern (rank 15-20). A last important thing to note is that Findbugs performs a static analysis so that it works without executing the input code.

### Example

The following example comes from an article [5] explaining how to write a custom detector for Findbugs. The author implements an "*unguarded logging*" detector. In other words, he wants to verify that programmers check whether the condition "*Logger.isLogging()*" is true before using the instruction "*Logged.log(...)*". As most programmers are lazy and because it is not convenient to look through the code of projects, he wrote a detector so that Findbugs does this verification automatically.

When writing a Findbugs detector, two methods have to be implemented, even though auxiliary methods can be added if needed. The *visit(Code)* method is called when the body of a method is analyzed and *sawOpcode(int)* is called for each *opcode* that belongs to that body. An *opcode* or operation code tells to the Java Virtual Machine which operation to perform. In other words, the JVM uses instructions that consists of an *opcode* followed by operands depending of the action defined by that *opcode*. For example, to store the int at the top of the stack into the first register, the JVM will use the instruction "*istore_1*". In fact, the machine will use the *opcode* "*0x15*" to refer to the "*istore*" action because it is more convenient to use hexadecimal numbers for all operations instead of words.

For this Findbugs' example, the detector needs to keep track of three values: the program counter where the logging instruction has been found as well as the boundaries of the '*if*' that checks the corresponding condition. As shown in the code fragment 2.1, the *visit(code)* method just initializes those parameters and then call the superclass' implementation which knows how to "visit" the parse tree of a Java program.

**Code 2.1:** The visit(code) method

```
public void visit(Code code) {
    seenGuardClauseAt = Integer.MIN_VALUE;
    logBlockStart = 0;
    logBlockEnd = 0;
    super.visit(code);
}
```

The sawOpcode(int) contains three parts that correspond to the three if-statements in the code fragment 2.2. That is because in this case, the detector needs to check three *opcodes*.

The first part saves the value of the program counter PC when "*Logger.isLogging()*" is found. In order to do so, the method looks for the *opcode INVOKESTATIC* which corresponds to a method call. Class and method names are used to identify the method called.

The second part checks that this call is part of an if-condition and if so, it saves the boundaries of the "*if*". The *opcode* to seek is *IFEQ* and if that *opcode* is close enough to the previous *INVOKESTATIC* that means that the "*Logger.isLogging()*" belongs to that if-condition. The writer explains that he found values +3 and +7 empirically.

The last part verifies that the next "*Logged.log(...)*" found belongs to the range computed. This time again, the method waits for an *INVOKEVIRTUAL* that corresponds to a "*Logged.log(...)*" call. This time the method call is found using *INVOKEVIRTUAL* instead of *INVOKESTATIC*. The difference is that *INVOKESTATIC* is used to call class methods that are declared as static while *INVOKEVIRTUAL* invokes an instance method. Then, if

the program counter is out of the if-boundaries that has been previously computed, a bug has been detected. A *BugInstance* is created with parameters that define the bug.

**Code 2.2:** The sawOpcode(int) method

```
public void sawOpcode(int seen) {
    if ("cbg/app/Logger".equals(classConstant) &&
      seen == INVOKESTATIC &&
      "isLogging".equals(nameConstant) && "()Z".equals(sigConstant)) {
          seenGuardClauseAt = PC;
          return;
    }
    if (seen == IFEQ &&
      (PC >= seenGuardClauseAt + 3 && PC < seenGuardClauseAt + 7)) {
          logBlockStart = branchFallThrough;
          logBlockEnd = branchTarget;
    }
    if (seen == INVOKEVIRTUAL && "log".equals(nameConstant)) {
        if (PC < logBlockStart || PC >= logBlockEnd) {
            bugReporter.reportBug(
            new BugInstance("CBG_UNPROTECTED_LOGGING", HIGH_PRIORITY)
            .addClassAndMethod(this).addSourceLine(this));
        }
    }
}
```

Note that, after implementation, new detectors need to be added to the Findbugs package before using them.

### Remarks

Findbugs can easily be used through its graphical interface (see Figure 2.1). However, as mentioned above, Findbugs is not easy to extend. Writing a custom detector can be really difficult because manipulating Java bytecode is not intuitive at all. Findbugs' detectors need to analyze programs *opcode* by *opcode* and for that reason, corresponding patterns to look for become quickly long to write. The expressiveness of Findbugs is limited by the expressiveness of the bytecode. Findbugs cannot exploit information that got lost when moving from source code to Java bytecode. For example, detectors cannot work on variable names. Variables are handled using *opcodes* such as *istore* and *iload* so that corresponding values are just stored and loaded whenever they are needed. The bytecode that corresponds to an instruction like "*int i = 1;*" will just store the value 1 in a register.

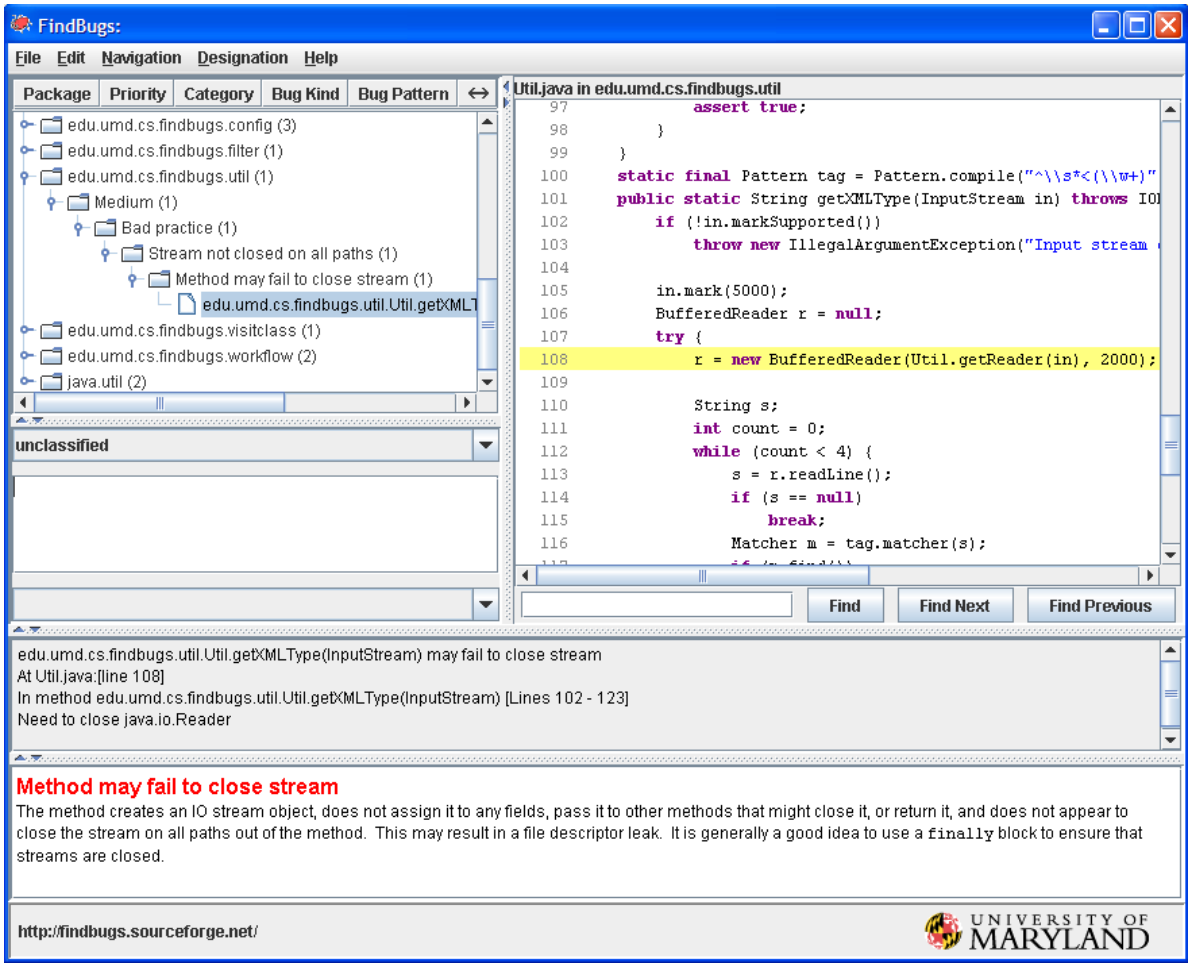


**Figure 2.1:** Screenshot of Findbugs' interface with reported bugs [3]

### 2.1.2 Checkstyle

#### Overview

A second analysis tool is Checkstyle. This one looks for violations of coding conventions. It is a really good tool to ensure coding standards and thus to improve consistency and readability in projects. Unlike Findbugs, Checkstyle works with abstract syntactic trees (AST) that are more easy to use. Another difference is that Checkstyle is more focused on coding style than on functional errors making it even more interesting for the purpose of this thesis. That being said, Checkstyle is less powerful than Findbugs since it cannot detect some of the bugs that Findbugs would report. In fact, as the code is represented as an AST, it is up to the user to interpret from that representation what the code really does unlike bytecode which is directly what the machine performs. In theory, Checkstyle could detect bugs as well as Findbugs but it is less convenient for functional checks. On the other hand, Checkstyle is better for coding checks because it can for example work on variable names that are lost in bytecode. All the checks provided by this tool are divided into several groups based on what they are looking for such as Class design, Coding or Naming conventions (see the Checkstyle website [6] for further details).

There are up to 15 categories because Checkstyle comes with a lot of different checks. As people do not really need all of them, users can specify in a configuration file which checks are needed.

#### Example

The following example (see code fragment 2.3) has been implemented by us for this thesis. The goal of this check is to find two nested if-statements so that the first 'if' only contains the second one and no other instruction. In that case, the two conditions could be combined to avoid the nested 'if'. As said in the previous section, Checkstyle use the AST. The nodes of an AST are called Tokens. When writing a check, at least two methods have to be implemented.

The first one, named *getDefaultTokens()*, specifies the starting node of the check, in this case, it is a "*LITERAL_IF*", starting from that node, the method goes through the AST until it finds the body of that '*if*'. To make sure that that block only contains the nested '*if*', there are two requirements. The first child has to be an "*if*" and the first and the last children have to be the same. In fact, the last child of the body is the "}" character so the last but one is taken instead. If both requirements are met then the check should notify the programmer that he could combine those nested "*if*". In the console, such notification looks like "/tmp/work/Hello.java:30: warning: You can combine those conditions.". The Java file that was analyzed is specified as well as the line of the detected error and after the warning keyword there is the message that corresponds to the checks and that is specified in a configuration file. Writing the message into the configuration file allows users to customize it according to his needs. For example, if a teacher want to use a check for french students, he can write a message in french without having to update the code.

**Code 2.3:** Nested if check

```
public class NestedIfCheck extends Check {
    @Override
    public int[] getDefaultTokens() {
        return new int[]{TokenTypes.LITERAL_IF};
    }

    @Override
    public void visitToken(DetailAST ast) {
        DetailAST objBlock = ast.findFirstToken(TokenTypes.SLIST);

        if(objBlock !=null) {
            DetailAST firstChild = objBlock.getFirstChild();
            DetailAST lastChild = objBlock.getLastChild().getPreviousSibling()
                ;

            if(firstChild.getType()==TokenTypes.LITERAL_IF &&
              firstChild.equals(lastChild)){
                  log(ast.getLineNo(),"nested.if.check");
            }
        }
    }
}
```

### Remarks

Checkstyle is easy to use and to extend. Because this tool relies on AST, custom checks work as a tree traversal. Each node is called a token and it is easy to get corresponding attributes thanks to implemented methods and a commented API. That being said, operating on a tree can quickly become complicated while dealing with the hierarchical organization of tree nodes. All Java keywords are represented by tokens so that the expressiveness of Checkstyle is quite good. However, its purpose is coding style therefore it suits more that kind of checks.

### Analysis of Checkstyle

Now that we explained why we chose to work with Checkstyle, we describe it more in detail in this section. Checkstyle is a source code based recommendation system which means that it goes through the source code and provides feedback to improve the quality of that code. The definition of the "quality of the code" depends on the system because all systems are not designed for the same purpose. In the case of Checktyle, it aims at improving the style of the code. To analyse the tool, we use the criteria that are described in [7] (see Figure 2.2); that paper explains the development decisions to make when building a recommendation system. After that, we look at all the detectors provided by Checkstyle and classify them for later use.

| | Requirements | Design | Implementation | Validation |
|---|---|---|---|---|
| **Approach** | 1. Intent | 3. Corpus | 5. Method | 7. Support |
| **User interaction** | 2. HCI | 4. General I/O | 6. Detailed I/O | 8. Interaction |

**Figure 2.2:** Development decisions as presented in [7]

### A Source Code Based Recommendation System

The authors of [7] compares different source code based recommendation systems by using 8 groups of development decisions that are classified along two axis. They identify development decisions that should be made when building source code based recommendations systems in order to address issues that may emerge. The first axis refers to the four phases of the development cycle: Requirements, Design, Implementation and Validation. The other axis specifies if the decisions are related to the approach used by the system or related to the user interactions. Following those axes, we first have to handle the approach and user interaction decisions for the requirements phase then those for the design phase and so on. Those decisions should be made while building a source code based recommendation system but here we used them to analyse Checkstyle.

#### Intent (Approach-Requirements)

This group contains four decisions that define the purpose of the system.

**Intended User** The target audience of Checkstyle are Java developers from novices to more experienced ones. The tool only handles the Java programming language but as it provides a lot of detectors, it is useful whatever the level of the user is. Examples of users are those that just want to check their own code, teachers that want to checks students' code or managers that want to keep some programming conventions between a group of developers. There is no assumption about the level of the user but the tool still needs to be configure accordingly.

**Supported Task** Checkstyle provides feedback, it analyses the code and gives recommendations about the style. As it does not raised errors but warnings, we can consider that it provides code suggestions.

**Cognitive Support** The tool answers two questions to improve the coding style. By providing feedback that contains specific detector messages, it answers the *what*: What is not good according to the Checkstyle point of view and what the user should change in his code. For some detectors, it also answers the how: How can the user correct himself. In fact, some messages as implemented in Checkstyle are just observation (for example *"6" is a magic number*) while others hint a solution (for example *Missing default case in switch statement*). It is also interesting to note that the description of some detectors on the website answers to the *why*: Why did the Checkstyle developers have implemented a specific detector (for example *Rationale: Some developers find inline conditionals hard to read, so their company's coding standards forbids them.* [6]).

**Proposed Information** As we just said, the tool describes what to change in the code and for some detectors how to change it. The feedback also specifies the line where the problem is found so that is is easier for the user to know where he should correct himself.

Human Computer Interaction (User interaction-Requirements)

This group of decisions describes the interactions between the user and the system.

**Type of System** Checkstyle can be used as a command line tool but also as an Eclipse's plug-in. Using that IDE makes it easier to configure and to use since there is a graphical interface.

**Type of Recommender** Checkstyle is an advisor since it suggests what to change. The two other types are "finder" and "validator".

**User Involvement** To use Checkstyle the user has to configure it so he has to chose the detectors and write a configuration file unless he uses the default one. Write the configuration file is not only writing the set of detectors but also tuning detectors' variable if there is any (for instance the limit of the length of code lines). The user also needs to parse the output because the tool provides the line where an error has been found but the user still has to look for the corresponding line in his code. However, if Checkstyle is used through Eclipse, lines with errors are directly marked.

Corpus (Approach-Design)

The corpus is the data that are required for the source code based recommendation system to provide its recommendations.

**Program Code** The recommendations are computed from a source code a developer is working on.

**Complementary Information** The tool only analyses the source code and no other input. In other words, Checkstyle provides information about the source code and nothing else so there is no complementary information.

**Correlated Information** Since Checkstyle only works on the source code, it does not have to handle correlated information between several data input. If a source code based recommendation system analyses more than just the source code, information that comes from those other inputs needs to be correlated with the information from source code. It is not the case with Checkstyle.

General Input/Output (User interaction-Design)

Decisions that are related to the user interaction but for the Design phase this time.

**Input Mechanism** Checkstyle does not require any other input from the user than the source code and a configuration file.

**Nature of Input** The input is the source code.

**Response Triggers** Checkstyle provides its feedback only when the user calls the tool. It does not work proactively and the feedback is only updated when the tool is called again.

**Nature of Output** The output contains the lines of the detected errors (if there is any) and the corresponding message of the detectors. It is generated as text in the console. In Eclipse, the integrated feedback makes it possible to click on the warning messages list and jump to the corresponding lines.

**Type of Output** The type of output provided by Checkstyle is suggestions to improve coding style.

Method (Approach-Implementation)

This section groups the decisions about the software recommendation process.

**Data Selection** Checkstyle makes use of almost everything in the parse tree of the source code: variables, values, scopes, types, expressions, instructions, code blocks, methods and classes. Depending of the set of detectors that are chosen, the level of details can be very specific.

**Type of Analysis** Checkstyle uses a syntactic approach as it transforms the source code into an abstract syntax tree and analyses it.

**Data requirements** The only requirement is that it should be Java code as it is the only supported programming language. That code should also be parseable since Checkstyle works on the corresponding parse tree.

**Intermediate Data Representation** The tool uses a tree-based approach since it works with AST.

**Analysis Technique** The technique is based on pattern matching. A detector is implemented so that it looks for specific tokens in the AST and acts accordingly. The technique used is then to visit the abstract tree and look for the corresponding pattern.

**Filtering** No filtering is performed by Checkstyle, there is no pre-processing before the analysis of a source code.

Detailed Input/Output (User interaction-Implementation)

Decisions about the detailed input/output required by the source code based recommendation system.

**Type of Input** Checkstyle does not need additional information except the configuration file that specifies which detectors to use and so what to look for in the code.

**Multiplicity of Output** The tool provides a single output that contains the lines with errors if there is any.

Support (Approach-Validation)

We do not know how the Checkstyle's developers validated it so we did not discuss those decisions. We can imagine that it is validated through feedback users can send to its development team but we have no information.

Interaction (User interaction-Validation)

It defines the interactions of different types of users.

**Usability** In general, the tool is very easy to use especially through the Eclipse graphical interface.

**System Availability** The source code of the implementation of Checkstyle is available on its website [6].

**Availability of Recommendation Data** Results of validation are not available.

#### Conclusion of the analysis

Even though the purpose of the development decisions was to make sure that source code based recommendation systems' developers do not forget something while building the system, it is still interesting to use them for our analysis. They cover all the important aspects of a recommendation tool. Analysing Checkstyle in that way allows us to have a better understanding of the tool and provide a good overview of it.

### Classification of detectors

Checkstyle's detectors are already grouped into categories based on what they are looking for. Examples of those categories are metrics, naming, imports, etc. Since there are more than 100 detectors, it is not always easy to find the detector we need. For that reason, we decided to make other classifications that we found interesting and that we used in chapter 6.

#### Novices and experienced users

The first classification we described in this section is to identify detectors that could be useful to help novices programmers such as learning students, and those for more experienced developers. Since our work is more focused on beginners, we briefly describe all the detectors for novices while we just explain some of the others. Novices' detector are the ones that can help beginners, there are not detectors that novices can use by themselves as some of those detectors can be tricky to configure. In other words, the novices' list is interesting for a teacher that wants to check students code while if we give that list to beginners, they might not be able to understand how to use it. The purpose is just to identify a set that focuses on errors that are not too complicated since we do not expect students to write overcomplicated code. Detectors are listed in alphabetical order.

Detectors for novices:

**AvoidStarImport** It is better to avoid star in import so we can see if students know what they really need to import. An example of star import is "*import java.util.**".

**AvoidInlineConditionals** Inline conditionals could be tricky for novices and we might want to avoid them.

**BooleanExpressionComplexity** This detector can be tune to set a limit on the number of Boolean operators in Boolean expression. For example, the complexity of "(boolean1==boolean2) && (boolean2==boolean3)" is 1.

**DeclarationOrder** It is used to defined an order on declarations.

**DescendantToken** This detector is a bit tricky to use but it allows to set limits of a certain type of token as child of another type of token. For example we can use it to make sure that in each method, there are from 2 to 5 local variables. To do so, we need to configure the detector so that it counts the children of *method definition tokens* that are *variable definition tokens* and set minimum and maximum limits to 2 and 5. It requires some knowledge about Checkstyle's AST but can be useful in some situations.

**DefaultComesLast** It can be useful if we want to promote the convention of writing the default case of switch statements at the end.

**EmptyStatement** It checks empty statements.

**ExecutableStatementCount** It counts the number of definitions (of methods and constructors) and initializations. A limit can be defined.

**FallThrough** A detector that check switch statements; each case should contains a *break*, *return*, *throw* or *continue*.

**FileLength** It is used to set a limit on the number of lines the file can contain.

**IllegalType** It can be useful to prevent the use of specific type.

**Indentation** It checks that the indentation is correct, it is possible to configure the number of spaces expected between indentation levels.

**InnerAssignment** Assignments in subexpressions are not very convenient to read.

**LineLength** A detector that checks the length of each instruction lines, a limit can be defined.

**MagicNumber** It checks magic numbers in the code. It can be configured so that some values are allowed.

**MethodCount** A user can configure this detector to define maximum number of methods.

**MethodLength** It is used to define the maximum length of methods.

**MissingCtor** A detector that make sure we write a constructor.

**MissingSwitchDefault** It is a good habit to always write a default case in switch statements.

**ModifiedControlVariable** It general, we prefer not to allow novices to modify control variables in loop.

**ModifiersOrder** Define an order for modifiers could improve the readability.

**MultipleStringLiterals** We might want to avoid writing the same String literal several times.

**MultipleVariableDefinition** It checks that there is no more than one variable definitions per line.

**NestedForDepth and NestedIfDepth** It can be configured to set a limit on nested for/if depth.

**ParameterNumber** It is used to make sure that methods and constructors definitions do not have too many parameters.

**ParametersAssignment** We might want to prevent novices to change values of parameters.

**NeedBraces** It can be used to make sure that novices write braces for *do*, *if*, *else* and *while* block. Syntactic sugar allows us to omit braces if the body is a single instruction but we find that it might be better to force students to write them in all cases (until we are sure they understand how those blocks work).

For example: write "if(!bool) { i++; }" instead of "if(!bool) i++;"

**OneStatementPerLine** To improve the readablity, it is better to not write more than one statement per line.

**RedundantImport** Novices sometimes import the same package without notice it.

**RegexpSingleline, RegexpSinglelineJava and RegexpMultiline** Those detectors can be used to look for specific expression in code. By tuning the limits, it is possible to force some expressions to be used as well as prevent others to be written.

**ReturnCount** It is used to put a limit on the number of return statement in methods.

**SimplifyBooleanExpression** Novices sometimes complicate Boolean expressions.

**SimplifyBooleanReturn** A detector for complicated Boolean return statements.

**StringLiteralEquality** Novices often use "==" instead of the equals methods when they compare Strings.

**TrailingComment** We might want to avoid trailing comment as some developers think that it is a bad practice. A trailing comment is a comment that is written on the same line of an instruction. For example: "*int iter = 0; //This is an iterator*".

**UnusedImport** While implementing a class, novices can try several implementation before they find the solution and forget to remove imports that are no more used in the final version.

**UnnecessaryParentheses** It can happen that we write more parentheses that needed.

**Detectors of Naming Conventions** All the detectors that check the naming can be useful if we want to promote a format.

Detectors for experiences users:

**CovariantEquals** It checks that if an "equals()" method is written, it overrides the "equals" from "java.lang.Object".

**CyclomaticComplexity** A detector that can be used if we want to set a maximum limit to the cyclomatic complexity of methods. Cyclomatic complexity refers to the number of different paths that can be followed from the start of a code to its end. For example, a method that just contains a return instruction has a complexity of 1 while a method with an if-statement has at least a complexity of 2 (depending on whether or not the condition is true, the execution will follow the path with the body of the "*if*" or not).

**IllegalImport** If we do not want some imports to be used, we can set them as illegal with this detector.

**IllegalToken** A detector that prevents from using specified token. If for example we do not want switch statement in the code, we can use this detector.

**JUnitTestCase** It checks that "setUp", "tearDown" and "suite" methods are used correctly (with correct signature).

**TodoComment** This detector looks for "TODO" in the code. We can use it to make sure we do not forget one.

We do not describe the 75 detectors left.

### Instructions, Methods and Classes

This classification is based on which level the detectors are working on. We identify three level: instructions, methods and classes and higher. As we have already described a good part of the detectors, we do not explain them more because it would be long and we would repeat ourselves. So, we simply show the distribution among those three levels. The purpose of such classification is that depending of the size of an exercise, we would not need to consider all the detectors.

| Level | Instruction | Method | Class and higher |
|---|---|---|---|
| Number of detectors | 53 | 14 | 54 |

What is interesting to see here, is that Checkstyle is more focused on instructions, classes and higher levels. With the detectors from the instructions level, we have a lot of checks that are very precise but also very specific. For small exercises that just contain a method to fill, almost a half of all the provided detectors can be ignored. The other half is more relevant for projects. On a platform such as Pythia, most of the exercise are methods to be filled by students.

### Code critics groups

As a third classification, we have followed the one that is used in [8]. In this paper, the authors discuss code critics which are detectors that belong to the Pharo Smalltalk IDE. Those code critics are classified in seven groups: Unclassified, Style, Idiom, Optimization,

Design flaw, Potential bug and Bug. As already said in the previous section, we do not go into details here as we just give the distribution.

| Groups | Unclassified | Style | Idiom | Optimization | Design flaw | Potential bug | Bug |
|---|---|---|---|---|---|---|---|
| Number of detectors | 28 | 50 | 10 | 13 | 13 | 7 | 0 |

As expected the main part of the detectors focus on the style. All other groups are more or less covered. There is no detector for bug, only for potential ones. As we already know, Checkstyle does not look for bugs since its purpose is to improve the coding style.

**Remarks**

Those three classifications are very subjective because other people would probably find different distribution. For each classification, some detectors could be put in a different group. There are also detectors that could belong to several groups but we decided to put them in only one. In the end, we think that those distribution are interesting to see as it helps to understand what Checkstyle focus on.

### 2.1.3 PMD

#### Overview

Our last analysis tool is PMD and offers a solution between what Findbugs and Checkstyle provide. It will check source code and finds bad practices that can lead to bugs. It is less powerful than Findbugs because finding functional errors is not its main purpose. PMD also works with ASTs and it comes with a lot of rules but it is easily customizable as well. According to its website [9], PMD looks for Possible bugs, Dead code, Suboptimal code, Overcomplicated expressions and Duplicate code. PMD includes rules for several languages but as the scope of this thesis is limited to Java, only the Java rule set has been looked at.

#### Example

Code fragment 2.4 comes from the PMD website [9]. This is the implementation of a rule that verifies that methods should not have more than one "*return*" instruction. PMD uses ASTs that are similar to those used by Checkstyle. Again, when writing rules, the visit() method has to be implemented.

First, this rule starts from nodes that are class or interface declarations. If it is an interface, there is nothing special to do. If it is a class, we call the second visit method. Abstract method are ignored as well. For other methods, the rule's implementation simply counts the number of child nodes that correpond to "*return*" instructions. If there is more than one child, then the rule iterates over them and raises the violation on the last "*return*".

**Code 2.4:** The only one reutrn rule

```
public class OnlyOneReturnRule extends AbstractJavaRule {

    @Override
    public Object visit(ASTClassOrInterfaceDeclaration node,
      Object data) {
        if (node.isInterface()) {
            return data;
        }
        return super.visit(node, data);
    }

    @Override
    public Object visit(ASTMethodDeclaration node, Object data) {
        if (node.isAbstract()) {
            return data;
        }

        List<ASTReturnStatement> returnNodes =
          new ArrayList<ASTReturnStatement>();
        node.findDescendantsOfType(ASTReturnStatement.class,
          returnNodes, false);
        if (returnNodes.size() > 1) {
            for (Iterator<ASTReturnStatement> i =
              returnNodes.iterator();
              i.hasNext();) {
                Node problem = i.next();
                // skip the last one, it's OK
                if (!i.hasNext()) {
                    continue;
                }
                addViolation(data, problem);
            }
        }
        return data;
    }
}
```

### Remarks

As shown in the example, PMD uses ASTs that contain abstract nodes. Those are extented for each kind of instructions that can be found in Java source code. Abstract nodes are a bit difficult to use because it contains some fields and methods that are not intuitive. PMD remains an expressive tool that is able to find some bugs as well as checking code style.

## 2.1.4 Other tools that worth to be mentioned

In this section, we talk about two tools that we did not use nor analyse in detail but that are still interesting to notice. We just provide an small overview.

### An Eclipse Plugin [10]

The Eclipse plugin that is described in [10] is an interesting tool. The authors have implemented a plugin for the Eclipse IDE in order to help Java developers to improve their style. To achieve that goal, they focus on 4 points.

**Promote the use of CamelCase and English word** CamelCase is a very common naming convention and the English language is probably the most used by developers. For those reasons, the authors believe that it is a good practice to follow those guidelines for naming. The idea is that they define a style for the users so that if everyone uses that style, the readability of the code is improved since everyone codes the same way. To check the names in the code, they split the word using the uppercase letters and then they look for occurrence in the English dictionary.

**Help to set the correct variable scope** The scope of variables should be the lowest. There is no reason to create a variable outside of a method if it is only used in that method. For each variable, the plugin compares the declaration level with the levels where the variable is used.

**Promote commented code** The plugin looks for declaration of classes, methods and variables but also loops and conditional branches in the code. For each of these lines, the tool asks the user to write a comment.

**Correct of code according to the style** For instructions block with bad indentation or with misuses of braces, the tool rewrite the block correctly.

Like the other tools that we have described, this plugin helps users to improve their coding style. The three first points are also handled in Checkstyle. The main difference is the last point. The other tools we have considered so far only provide recommendations without modifying the code while this plugin corrects the code. Depending on the case, being able to change the code directly can be very interesting.

### A static analysis framework [11]

After explaining how important a good coding style is, the authors of [11] describe how they have implemented a framework to help Java beginners. They have first identify nine poor programming practices such as *too many loops*, *not enough methods*, *unused variables*, etc. They also find four common errors such as *confusion between instance and local variables*. After that, they have created a framework so that users can correct those mistakes. So they developed a static analysis framework that works like Checkstyle in the sense that it works with abstract syntax tree and the user can chose the framework's checks that he wants to use.

An interesting feature is that it is possible to perform a structural similarity analysis. To do so, the user has to provide a model of the solution so that when a student code is analysed, a comparison can be made (see figure 2.3). The framework comes with a graphical interface so that it is very easy to use.

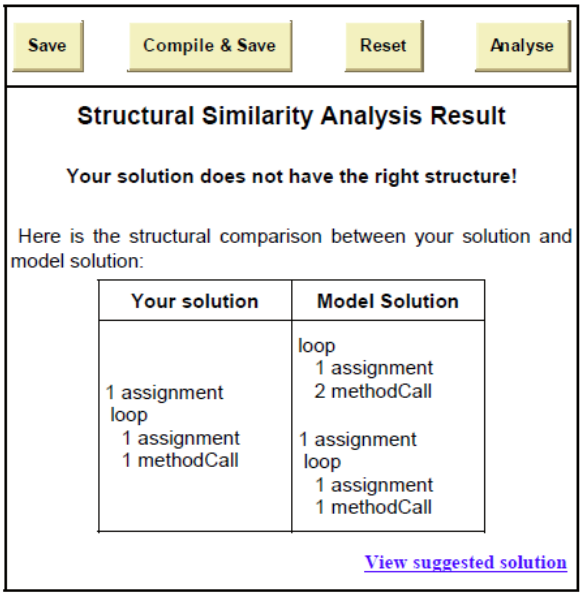


**Figure 2.3:** Structural similarity analysis [11]

## 2.2 Pythia

All the research made in this thesis would serve Pythia [12], a platform developed here at UCL, by adding new features. Thanks to Pythia, teachers can write programming exercises and put them online so that students can make them at home using a web interface. Submitted code can be checked automatically so that students directly know if their code works as intended or not. It is important to note that Pythia does not depend on any programming language so that it is a really complete and practical tool. Using this platform facilitates the entire process of exercises/homework/projects given to students by teachers.

### 2.2.1 A platform for students and teachers

As mentioned before, Pythia is an online platform that serves teachers and students at UCL. It is built so that it can handle several programming languages which allow teachers to use that platform whatever the language they want to teach is. Pythia provides both theory and exercises.

Teachers provide theory by giving their courses to one of the Pythia developers that then upload them on the server (see section 5.2.5 for more details). Once it is done, students just need to subscribe to those courses to access them. In fact, they need to create a account on the platform and once they subscribe to a specific course, they can either read it on the corresponding webpage or download it in a PDF format. The practical part consists of modules that contain a set of exercises that are related to courses (see Figure 2.4). Exercises are displayed such that students can directly see the name of the exercise, the difficulty rated on a five-stars indicator and a status. Several states have been implemented so that students can clearly know their progress. If nothing is displayed as status, that means that the student never clicked on the exercise. As soon as the student opens the webpage of an exercise, the state is set to "*started*". While looking at the status, it is really easy to see what has been done as well as if there are still some things to do.


**Figure 2.4:** The list of exercises related to the course "[LSINF1101] Introduction à la programmation"

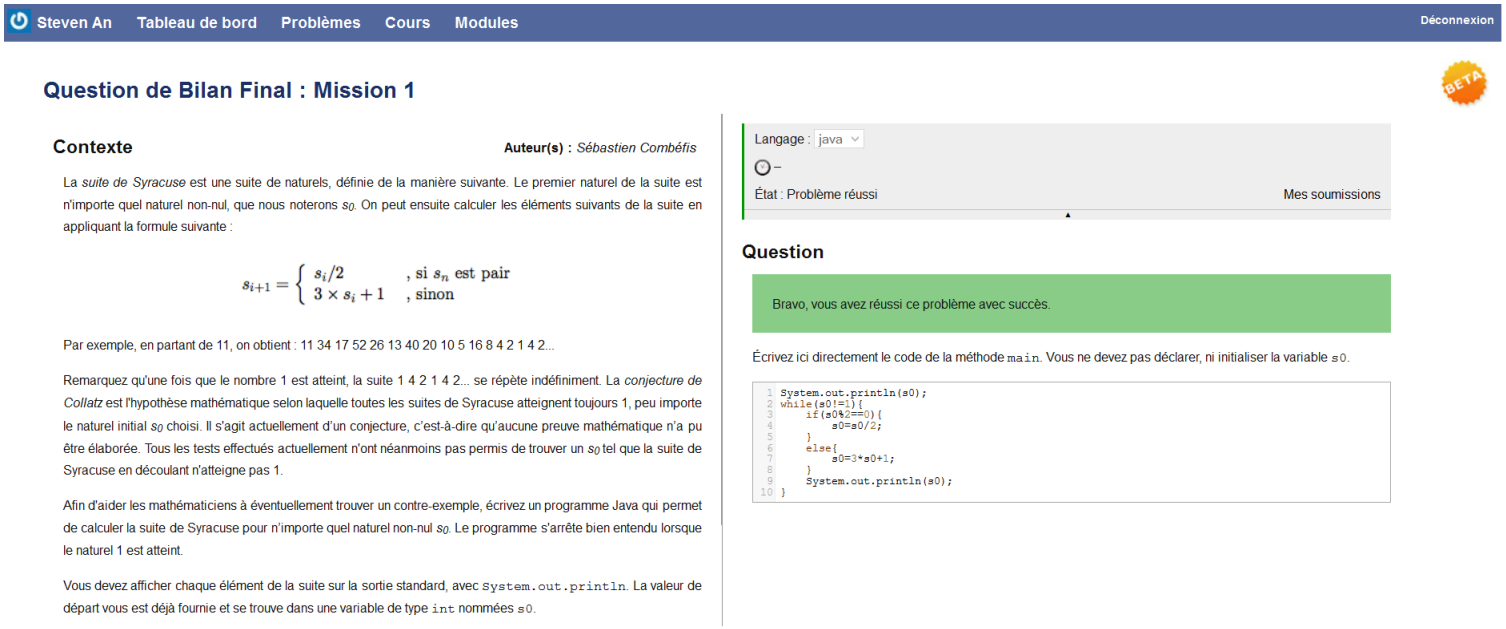


**Figure 2.5:** An exemple of exercise from the course "[LSINF1101] Introduction à la programmation"

The Figure 2.5 shows an example of an exercise provided on Pythia. The text on the left part of the screen explains the exercise and so, what students have to do to resolve the problem. The right part is where students write their code so that their method or their program performs the action described in the left part. Depending on the exercise, there can be more explanation or some specifications to complete the description with more technical instructions or other implementation details such as methods signatures (for exercises that ask students to complete the body of a method). As shown in Figure 2.5, several features are implemented.

First, there is a drop-down list to select the programming languages chosen by the student for his implementation. So it is possible for a teacher to write an exercise for which they allow several languages to be used. A clock icon is used when a due date to resolve the exercise is defined. The status of the exercise is also given on each corresponding page. To write code, there are coding frames with implementation instruction as mentioned before.

Students can also click on two buttons. The first one allows students to save their code. Such feature allows students that cannot finish an exercise directly to save what they start and resume later from where they were. In this case, the status will be set to "*saved*". The second button allows students to submit their code so that it can directly be automatically checked (if it is implemented). Then the webpage will provide some feedback. For example, teachers can implement unit test and if so, an error message such as "*test with input 3 failed*" or "*With input A your method should return B but it returns C instead*" will be displayed. Teachers could also implement performance checks if an exercise needs the code to perform with an execution time that is not too long for example.

Once submitted for checking, the status becomes "*corrected*". When the student decides to correct his error(s), he can chose if he wants to start from what he has already written or if he wants to delete his code and reset the exercise. Finally, when the exercise is correctly done, the status is set to "*resolved*". Note that for all status, the date is saved so that students can see when they were working on exercises. Last feature is the submission menu which allows students to see all their previous submissions (if there is any) with the corresponding feedback generated at the time of each submission.

### 2.2.2 The architecture of Pythia

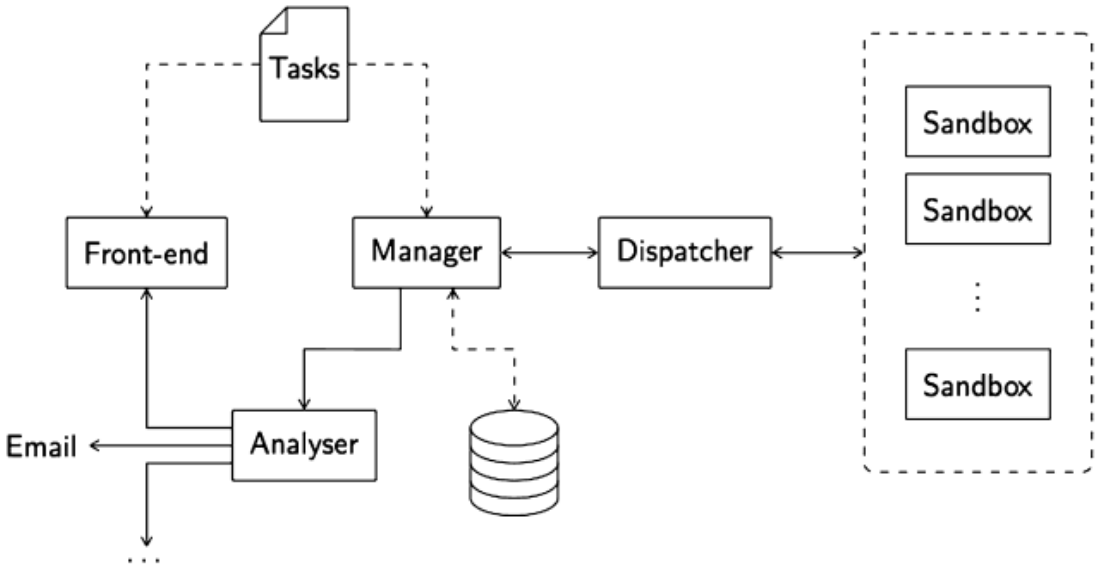


**Figure 2.6:** Pythia's architecture (Figure taken from [12])

The Figure 2.6 is a simplified representation of the architecture of Pythia (for more details, see [12]). The first element in that figure is the front-end. It groups all interfaces through which students can interact with the platform. The main interface is the website. When students resolve exercises on the web interface with their code, they provide some input using this front-end. The theory of courses is also given through this component.

On Pythia, exercises are considered as tasks that will be executed. Each task is defined with a set of parameters such as the environment to use in order to run the code or a time value before raising a time-out error if the task takes too long to end.

The manager, as its name suggests, manages all tasks. In fact, Pythia exercises are written as program templates that are then completed with codes submitted by students. The action of taking the code and filling the template is performed by the manager. He also forwards complete programs to sandboxes through the dispatcher.

Programs are executed on those sandboxes. A sandbox is an environment that is isolated and used to test an untrusted code. Programs written by students are not always safe and we cannot be sure that they will not damage the platform. Using a sandbox is a preventive measure.

The dispatcher is the component that links the manager and sandboxes. When it receives tasks, it queues them so that if no sandbox is available, those tasks will be put on hold until one is free.

Once the code is executed, it is then analysed. The analysis consists of a set of checks such as unit tests, performance indicator computation or structural code checking for instance. The checking has to be implemented on Pythia beforehand so it requires teachers to know how they want to check if their exercises are correctly implemented by the students. For example, if a teacher ask a student to write a method that takes an input and returns an output, he can write unit tests. Those ones would give a set of input instances to the method implemented by the student and check the output. Since the teacher knows what should be the result for each of those input instances, he can compare the output from the student method with the values that he expects to have. Of course, he can write that test in a script so that it is executed automatically. The result of such test can then by displayed on the web interface to the students as a feedback so that they can use it to correct themselves (if needed) directly or it can be sent by email or saved on files for other purposes.

Last thing to note is that is that Pythia is in beta for now which means that there is still work to do and room for improvements. With this thesis, we hope we can promote another kind of checks that have not been used on Pythia so far.

# Chapter 3

# Related work

Since there are more and more tutorials on the web and people tend to learn by themselves, it is not a surprise to see online learning platforms being developed. Indeed, Pythia is not the only one of its kind and several other platforms exists. In this chapter, we will introduce some of them and provide on overview of how they work.

## 3.1 Project Euler

The Project Euler website [13] counts more than 400 problems (see example in Figure 3.1) that will challenge both mathematical and programming skills in order to be resolved. In fact, to reach the solution, people need to reason mathematically then implement their reasoning since computations are too complex to not use a computer. Once the correct answer is encoded for an exercise, a solution to the problem is given as illustrated in Figure 3.2. This particular example shows that Project Euler seems to have some kind of performance test since the solution begins with "*To get a more efficient solution...*". Mathematical reasoning are explained with formula that are then translated into pseudo-code. On Project Euler, there is a post for all exercises so that the community can share their thoughts, their point of view or different reasoning approaches. Users can also check their progression which is tracked by the site. A lot of statistics are computed providing even more data to participants. Unlike Pythia, Project Euler is aiming for people that already have both mathematical and programming knowledge, it is more about reasoning than learning but it is still worth to mention.

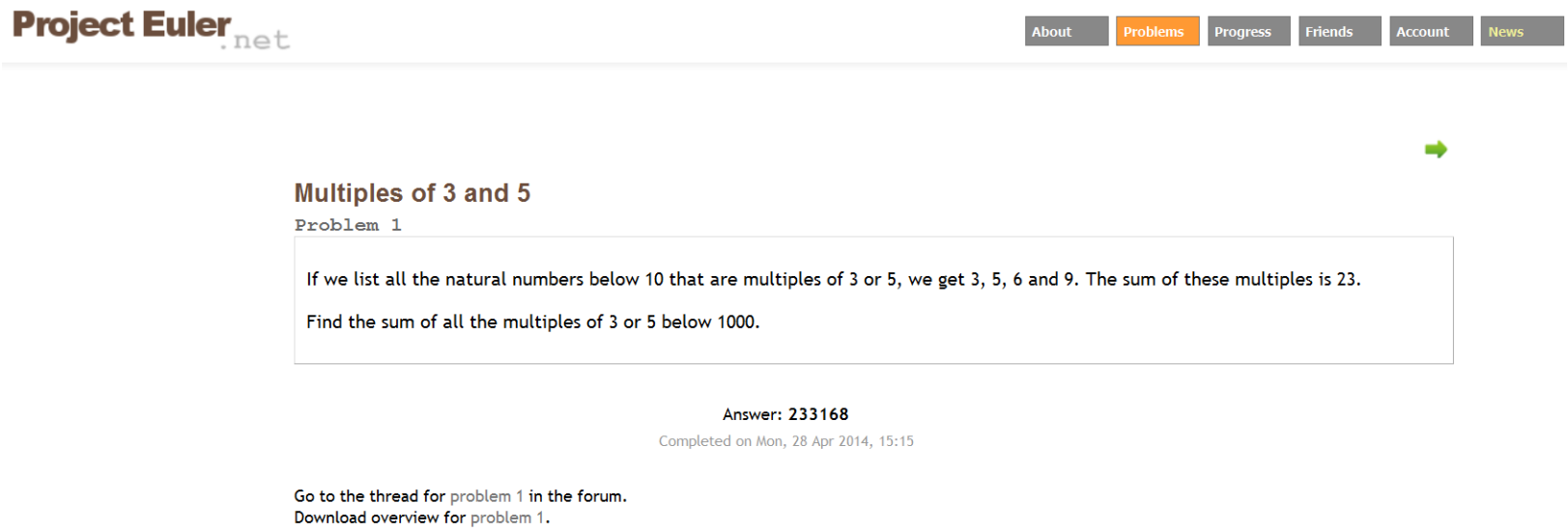


**Figure 3.1:** An example of problem found on Project Euler

To get a more efficient solution you could also calculate the sum of the numbers less than1000 that are divisible by 3, plus the sum of the numbers less than1000 that are divisible by 5. But as you have summed numbers divisible by 15 twice you would have to subtract the sum of the numbers divisible by 15.

If we now define a function:

```
Function SumDivisibleBy(n)
    Details to be filled in
EndFunction
```

Then the answer would be

```
SumDivisibleBy(3)+SumDivisibleBy(5)-SumDivisibleBy(15)
```

Let's look at the details of our function and take as example n=3.
We would have to add:
3+6+9+12+.....+999=3*(1+2+3+4+...+333)
For n=5 we would get:
5+10+15+...+995=5*(1+2+....+199)
Now note that 199=995/5 but also 999/5 rounded down to the nearest integer.
In many programming languages there exists a separate operator for that: `div` or `\`.
If we now also note that 1+2+3+...+p=½*p*(p+1) our program becomes:

```
target=999

Function SumDivisibleBy(n)
  p=target div n
  return n*(p*(p+1)) div 2
EndFunction

Output SumDivisibleBy(3)+SumDivisibleBy(5)-SumDivisibleBy(15)
```

**Figure 3.2:** A part of the solution provided for example in Figure 3.1. This feedback comes from the Project Euler website.

## 3.2 RubyMonk

RubyMonk [14] is, as its name suggests, a platform that teaches Ruby. Users follow a course that leads them through all the Ruby language can do. Theory is given with concrete examples. Then beginners can resolve small exercises based on both those examples and the related theory (see Figure 3.3). For all of those exercises, the site provides unit test to check the correctness of code. Once Ruby beginners have read all chapters, they can try to resolve problems that requires them to make use of all what they learned. Like for exercises in the theory, functional checks are implemented so that users can check their code and know if they are doing it right or not. On Pythia, we want to provide more things such as checking structural regularities in source code and not only functional check like RubyMonk does.

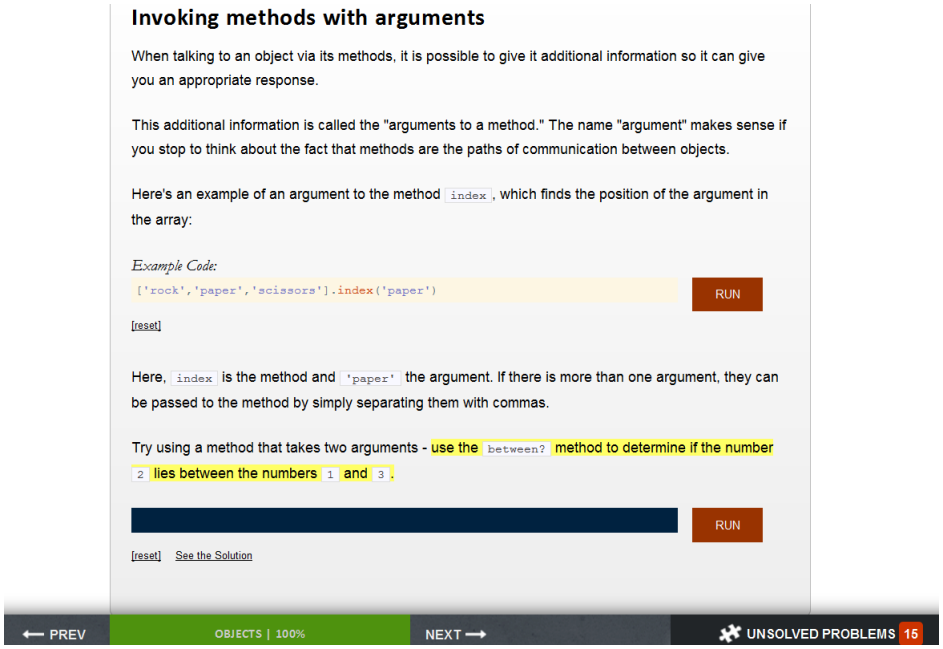


**Figure 3.3:** Exercises are included in the theory to make sure that beginners understand what they read. This screenshot comes from the RubyMonk website.

## 3.3 Try Python

Try Python [15] is like RubyMonk but for the Python language. Webpages are divided in two parts: one is used to explain the theory and the other is the Python interpreter (see Figure 3.4). Code is directly included in the theory with a button that moves those instructions blocks into the interpreter to execute them. Python beginners can then see the output of those provided code fragments. Thanks to Try Python, users can become more familiar with the programming language without needing to open a console as the interpreter is included. If they want to try their own code, they are free to type what they want as well while keeping the theory next to it.

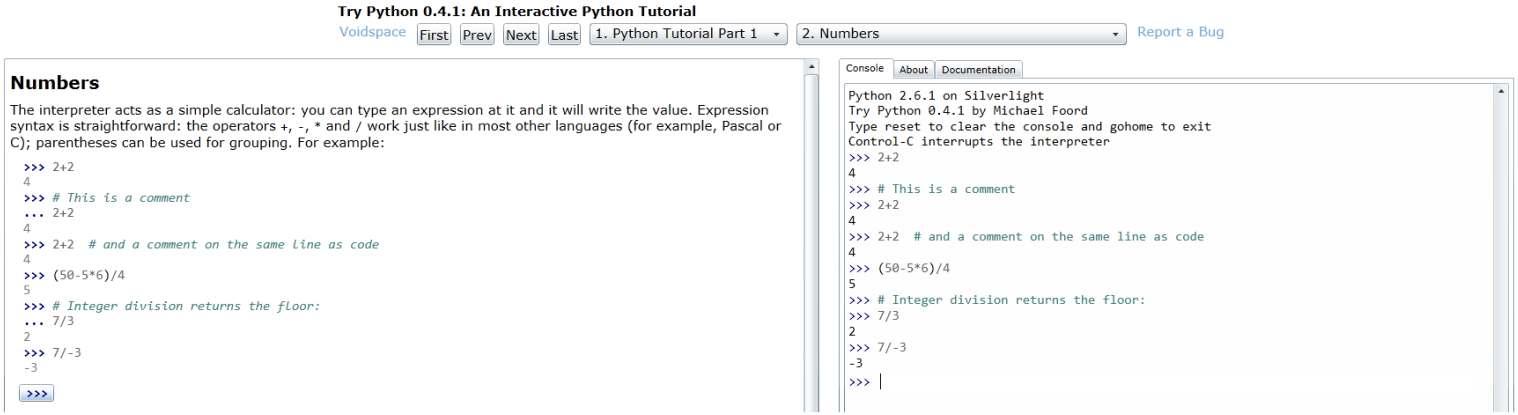


**Figure 3.4:** Code fragments explained in theory can directly be executed on the Python interpreter put on the right. This screenshot comes from the Try Python website.

## 3.4 Code School

A last example of a learning platform is Code School [16]. It provided courses for Ruby, JavaScript, HTML/CSS and iOS applications development. Code School like previous examples provides theory to learn as well as exercises to resolve. Functional checks are included to make sure users achieve what they are asked to do. Each course starts with a video giving an overview of the main concepts (see Figure 3.5). Then beginners have access to more complete theory while they are resolving exercises. In case they are struggling to find the answer, they also can have a look at some hints that guide them to the right path as illustrated in Figure 3.6.


**Figure 3.5:** Courses start with an introduction video

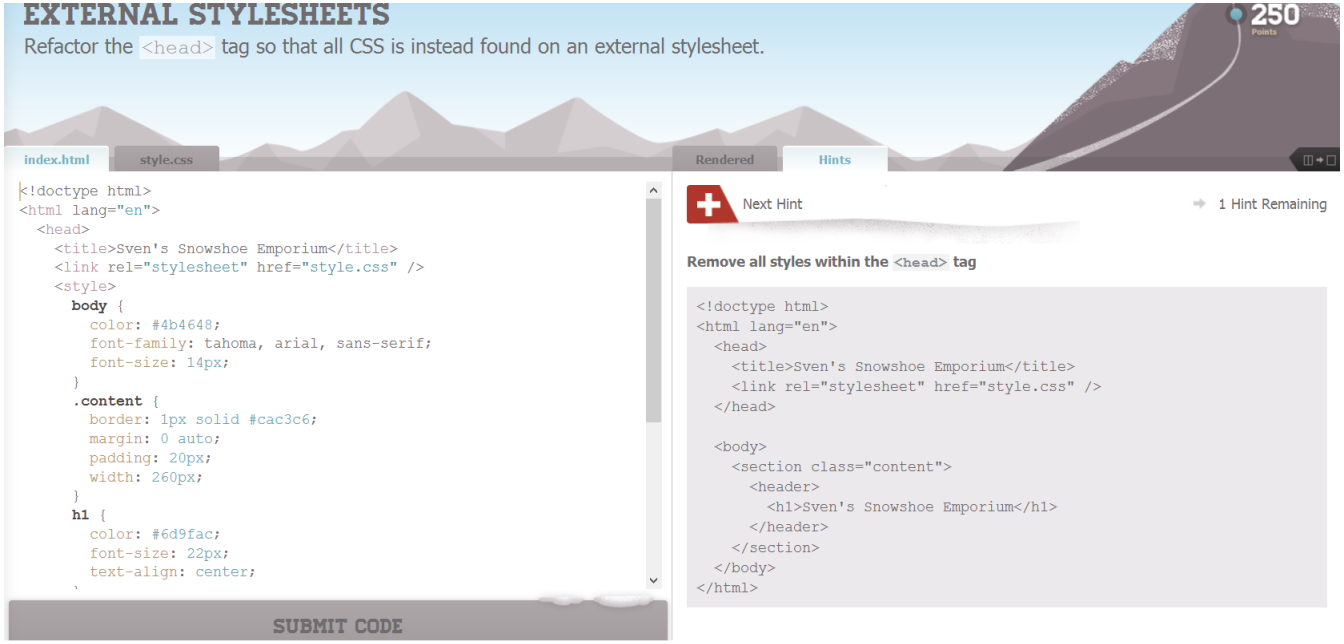


**Figure 3.6:** Hints are provided if needed. This screenshot comes from the Code School website.

## 3.5 Remarks regarding Pythia

Most of the previous examples are focusing on one language but there exists similar platforms for all programming languages. It is interesting to note that Pythia shares some features with all those platforms. It provides a set of courses on different programming languages (see Figure 3.7). Exercises are provided with the theory for students to check if they understand what teachers try to teach them. Functional checks ensure correctness of code written by students. Depending on what is asked to be done, it is also possible to compute performance indicators if needed. Pythia provides a good set of features we meet in other platforms. Considering all those examples, learning platforms need a mix of theory and exercises that can be automatically checked to evaluate students understanding.

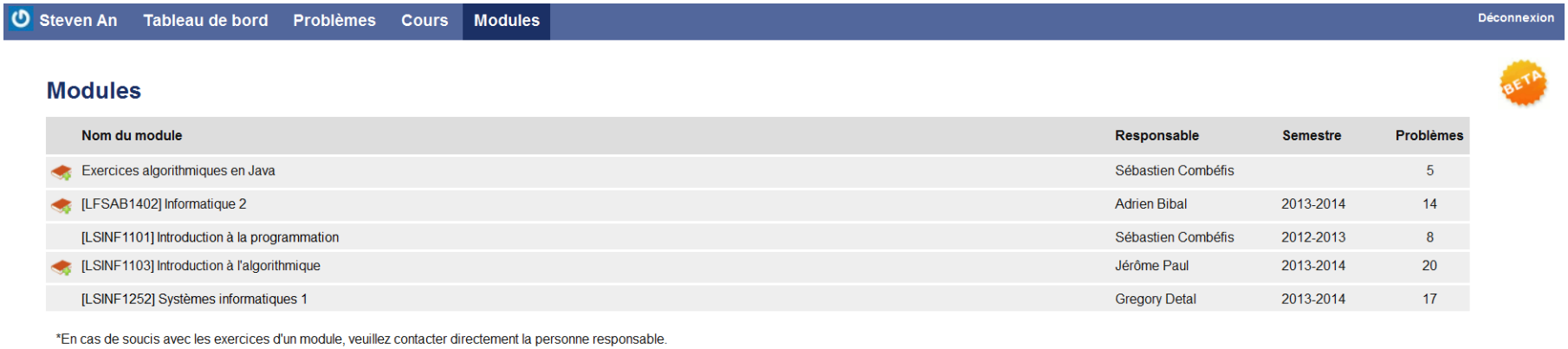


**Figure 3.7:** Pythia provides exercises for several programming languages

# Chapter 4

# Problem statement

Now that all materials needed have been introduced, this chapter focusses on the problem we try to solve. First, it is important to define what this thesis aims at : good coding checks. Then, Java bytecode and AST approaches will be discussed using what we have learned in chapter 2.

## 4.1 Functional checks and good coding checks

When thinking about checks that would help Java beginners, several types of checks come to mind. At this point, the two main group of checks should not be confused : functional checks and good coding checks.

### 4.1.1 Functional checks

On one hand, there are functional checks. Those ones try to find errors in code that makes the program not working as intended. For example, if we ask a student to write a method that computes the square of a number and his code provides the cube, a functional check could detect that he used the number 3 in his formula instead of 2. A better approach would be to write unit test that will try several values as input and check the output. This example is very basic but it is just to show what this kind of check is looking for. Does the method/class/program really perform the task it is written for ? Does it have the expected behaviour ? Functional checks helps answer those questions.

### 4.1.2 Good coding checks

On the other hand, there are good coding checks. They want to teach good habits while writing programs. Unlike functional check, they will not verify if the code is correct or not. They just help beginners write code that is readable and matches standard rules. For example, if a student writes something like "if(anyBoolean==true)", a good coding check could notify him that the "==true" part is useless and should not be written. Of course, if a program does not satisfy good coding checks, it could still work. However, detecting this kind of mistakes

will help the teacher to realize that maybe his student does not really understand how the Boolean type works. Then he can provide specific explanation to help a Java beginner and remedy the gap in his knowledge.

### 4.1.3 The importance of using coding checks

Why is it so important to provide coding checks ? Even if bad coding could not prevent a program to run as expected, Java beginners need to avoid it. Such checking is interesting to have on a teaching platform as Pythia is for few reasons.

First, good coding should be taught as soon as possible because when human beings learn to do something, they tend to build some habits in their mindset. After a certain point, it quickly becomes really hard to lose those habits even if they are bad. For instance, a person who always makes his writings corrected by someone else without looking at the corrections will always misspell some words. Good habits need to be taught during the learning process so that it becomes automatic and clear in the head of the Java beginner.

Second, as said in the previous paragraph, coding checks could help teachers to catch students' lacks in programming skills.

Third, automatic checking allows teachers to save time as it provides some kind of preprocessing by cleaning dirty code. Furthermore, it generally improves the readability of the code. From a student point of view, the beginner can directly use the feedback to correct his code as well as to learn by himself. It is also known that several students are sometimes too shy or fear to ask questions about something they do not fully understand. Last point is the fact that the feedback is provided at the time students do their homework, they do not have to wait the correction.

More generally, there are also some advantages while ensuring good coding style as it could enhance a program regarding some criteria [17] such as:

- Modularity: In the sense that for example, it could promote the declaration of explicit constructors or the use of accessors to access private variables.
- Typography: When we look at commenting practices (Is the source code well documented? Are the comments not too long?).
- Clarity: When we promote naming convention with checks (variable names are explicit, not too long, etc.).
- Effectiveness: For example, checks to promote variables with small scope.
- Reliability: An example is a check to force the programmer to write default blocks in switch statements.

## 4.2 Comparison between Java bytecode and AST approaches

As mentioned while introducing the selected Java coding tools in chapter 2, two different approaches were used to analyse code. Findbugs uses Java bytecode while Checkstyle and PMD use AST. Is one approach better than the other ? To provide an answer to this question, we compare those solutions according to several criteria.

### 4.2.1 Expressiveness

Using Java bytecode, it is possible to make use of all the information that is not lost in the bytecode. In fact, the process from source to bytecode tries to optimize the code by removing everything that is unnecessary for the program to work. For example, while reading Java bytecode, variable names are omitted. It only keeps what is needed to execute the task. For the rest, everything can be expressed since it corresponds to what the machine really executes. If there was something that could not be expressed, that would mean that the machine cannot perform the corresponding instruction. It is a bit different for AST. The expressiveness of that kind of tree depends on what is implemented for nodes in order to represent instructions. Imagine, we use AST with a node implementation that would only handle numbers. Then the expressiveness of such AST is limited to numbers. In the case of Checkstyle and PMD, because both tools want to be as complete as possible, both implementations provide complete API making them very expressive. Looking at this criterion, both approaches are almost similar except for variable names and other bytecode optimizations which make the AST approach a bit better.

### 4.2.2 Ease of use

Java bytecode is machine language and it is not very easy to use by inexperienced programmers. That is why higher-level languages were made. For that reason, writing checks based on bytecode is quite hard and not really intuitive. ASTs are more convenient to work with. As Checkstyle and PMD are implemented, each node has one type which corresponds to a Java instruction. Thanks to that, it is easy to look for patterns. The main drawback is that while using ASTs, we need to deal with tree structure as nodes are placed in a hierarchical model. Consequently, it is important to understand nodes level as well as parent/children links. Speaking of ease of use, ASTs are more intuitive and convenient to handle.

### 4.2.3 Conclusion

Based on those two criteria, it appears that both the bytecode(Findbugs) and AST(Checkstyle and PMD) approaches are pretty similar but the latter is better for us. Both approaches are expressive enough to provide lots of detectors. Working on bytecode seems more convenient for finding bugs because the code is optimized so it is easier to get to what a code really does. On the other hand, AST are more convenient for detecting bad coding habits. It is better for coding checks because it does not lose any information from the source code so that checks on variable name can be implemented. Another advantage is that AST keep the structure of the program which might be useful to identify bad patterns we want to look for. For those

reasons, we chose to use Checkstyle which is really easy to extend. We did not go with PMD because it is very similar to Checkstyle but less documented and so a bit less convenient.

# Chapter 5

# Implementation

This chapter explains how we implement our solution. We will describe how we decided to use Checkstyle and how we integrate it on Pythia.

## 5.1 Using Checkstyle

Checkstyle provides a lot of checks but depending on users needs, it is up to them to configure the tool accordingly. Following that idea, we went through all checks and extracted those that would handle errors we wanted students not to make. The first step is to identify which Checkstyle checks handle common errors that are often made by Java beginners. We do not look for advanced detectors since we are focusing on students that learn a programming language. However, the use of an analysis tool such as Checkstyle requires that we configure it according to the exercises we want it to check. In other words, Checkstyle's usage is case dependent. For example, it would not be useful to execute a check on method name if we just ask students to write instructions in the body of an "if-statement". In the validation chapter, we will create a concrete exercise and we will describe how we chose checks for that particular exercise. Among the implemented checks there are few that are generic and could be useful for all kind of exercises. An example of such checks is the "OneStatementPerLine" detector or at a less generic level the "SimplifyBooleanExpression" check. If CheckStyle does not provide a check that we need, we also have to possibility to write custom checks.

## 5.2 Extending Pythia

The integration with Pythia can be a bit tricky to understand for everyone who does not have the opportunity to dig into the Pythia implementation files. For that reason, we will not explain everything in detail but we will just give an insight of how it works by going through the most important file to explain.

### 5.2.1 Defining an environment

As explained in chapter 2, Pythia can handle several programming languages. In order to do that, we can define environments for which the platform will create a virtual machine. In our case, we need a Java environment with Checkstyle. In the Pyhtia environment folder, we create a new folder that contains a makefile and a rootfs-config.sh that specifies all the libraries that are needed in our environment to perform our check. Note that we also need to include the Checkstyle jar and configuration file as detailed earlier.

### 5.2.2 Writing a task

Exercises on Pythia are defined as tasks. Each of them has 4 parameters that have to be set to specify resource allocation. The corresponding task is shown in code fragment 5.1

- The time limit before the task will be considered as failed if no result is generated.
- The memory granted to the task.
- The space on the disk that the task can use.
- The size limit to print the output.

**Code 5.1:** An example of task resource allocation

```
{
  "environment": "nqueens",
  "taskfs": "nqueens-java.sfs",
  "limits": {
    "time": 60,
    "memory": 10000,
    "disk": 500,
    "output": 1555524
  }
}
```

Note that other files are created for a task but are omitted here since we do not want to go too deep in detail.

### 5.2.3 The server part

In order to create an exercice on the server of Pythia, two files have to be implemented.

The first one is a HTML file that implements the web page of an exercise. An example of such implementation is shown in the code fragment 5.2. It is the file we have implemented for an exercise that we used in our validation part (see section 6.1.2). For that exercise in particular, we have defined two main text area. One text area is the panel that contains the class template to be filled by students while the other text area is used to display the output of checks that have been implemented beforehand. There is a third text area for students to encode their matricule number so that we can identify them. We also add a button so that once their code is inserted, students can click on it to perform those checks. In fact, the submitted code is saved then is used in the second file that we described below.

**Code 5.2:** The implementation of the webpage for the Nqueens exercice described in the section 6.1.2

```
 <!DOCTYPE html>
<html lang="fr">
<head>

<title>Pythia</title>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.min.js"
    ></script>
     <script>



  $(function()
        {

function isEmpty( $el ){
      return !$.trim($el.val());
  }

        $('#send').click(function(){
$('#send').attr("disabled", "disabled");
$matricule = $('#matricule');
if (isEmpty($matricule)){
$('#matwarn').html("Matricule manquant");
$('#send').removeAttr("disabled");
}
else {
$('#matwarn').html("Merci de patienter");
var mytest =  encodeURI($('#code').val());
var myvar = '{"input":"'+mytest+'","id": "'+$matricule.val()+'"}';
    $.ajax({
    url: "/submit",
    type: "POST",
    data: myvar,
    contentType: 'application/json',
         success: function(data) {
              $('#back').html(data);
              $('#matwarn').html("");
$('#send').removeAttr("disabled");
          }
   });
}
            });
        });

    </script>
</head>
<body>
<p><textarea id="code" spellcheck="false" style="width:700px;height:500px;
    float:left;" class="form-control pythia-input" name="code">
/**
 * The Goal of this class is to check if a given board is a solution to the
     Nqueens problem.
 */
public class NqueensSolver {

        /**
```

```
         * In this main method, you should only check the size of the board (
             is there
         * a solution for all size ?) and then call you method.
         *
         */
        public static void main(String[] args){
                int[] board={2,0,3,1}; // An example of board
        }

        /**
         * This method should call all the two auxiliary methods needed to
             check the board
         * and print in the console if the board is a solution or not.
         * @param board is the board to check
         */
        public static void isSolution(int[] board){

        }

        /**
         * This method should check if there is any line conflict in the board
             .
         * @param board is the board to check
         * @return true if there is no line conflict
         */
        public static boolean checkLine(int[] board){

        }

        /**
         * This method should check if there is any diagonal conflict in the
             board.
         * Hint: The method "Math.abs" could be useful.
         * @param board is the board to check
         * @return true if there is no diagonal conflict
         */
        public static boolean checkDiagonal(int[] board){

        }

}
</textarea>
<textarea id="back" style="width:500px;height:500px;" disabled spellcheck="
    false" ></textarea>
</p><p>Matricule : <input type="text" id="matricule" /><span id="matwarn"
    style="color:red;"></span><br></p><p><button id="send" class="btn btn-
    default">Envoyer</button></p>
</body>
</html>
```

The second file is a Python file in which we describe how Pythia should interact with the web page. In our previous example, the submitted code was just saved so we define what should be done with that code in our Python file (see code fragment 5.3). For that exercise, we wanted to run Checkstyle on the code and that is what we write in the Python file. When our checks are performed, several outcome can happen. We could have a timeout or an overflow error and if so we print a message to notify the student. If checks are correctly executed, the

feedback generated is displayed on the exercise page so that students can see it and correct their code according to it. In our case, we also decide to save both the submitted code and the feedback so that we can analyse those results. In the python file, we decribe what checks to perform and what to display depending of the output of those checks. In our example, we decided to directly print the output because we have decided that it is clear enough. However, it would be possible to make other computations or analyses on the output and then display a message based on that output.

**Code 5.3:** The implementation of the Python file for the Nqueens exercice described in the section 6.1.2

```
{
        #!/usr/bin/python
# -*-coding: utf-8 -*
from bottle import request, route, run, response, static_file, error
from json import dumps
import json
import socket, sys, re
import codecs, time,urllib.parse

myvar = 1


def count(check):
    if (check == "TRUE") :
        succeed = codecs.open('succeed.txt','r+','UTF-8')
        value = succeed.read()
        succeed.write("1\n")
        succeed.close()
    else :
        tried = codecs.open('tried.txt','r+','UTF-8')
        value = tried.read()
        tried.write("1\n")
        tried.close()

@route('/', method="GET")
@route('/', method="POST")
def home():
    return static_file('index.html',root='/home/admin/server/')



@route('/submit', method="GET")
@route('/submit', method="POST")
def submit():
    global myvar
    ts = time.time()
    myvar += 1
    if 'application/json' in request.headers['Content-Type']:
        requested = request.json
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sock.connect(('localhost', 9000))
        filename = requested['id']+"_"+str(ts) +"_"+str(myvar)
        newjson = '{ "message": "launch", "id": "'+filename+'", "task": { "
            environment": "nqueens", "taskfs": "nqueens-java.sfs", "limits": {
            "time": 60, "memory": 32, "disk": 50, "output": 1024 } }, "input":
            "'+requested['input']+'"}'
        sock.sendall(newjson.encode('utf-8'))
```

```
        returned = sock.recv(15024)
        sock.close()
        returnedd = json.loads(returned.decode('utf-8'))
        infile = codecs.open(filename+'.in','w','UTF-8')
        s=urllib.parse.unquote(requested['input'])
        infile.write(s)
        infile.close()
        if returnedd['status'] == 'success' :
            outfile = codecs.open(returnedd['id']+'.out','w','UTF-8')
            outfile.write(returnedd['output'])
            outfile.close()
            if ".java" in returnedd['output'] :
                count("FALSE")
            else :
                count("TRUE")
            return returnedd['output']
        elif returnedd['status'] == 'overflow' :
            newjson = '{ "message": "Erreur lors de l'exécution !", "status":
                "failed", "output": "Votre implémentation génère un résultat dé
                passant la mémoire allouée à la tâche !"}'
            return (newjson)
        elif returnedd['status'] == 'timeout' :
            newjson = '{ "message": "Erreur lors de l'exécution !", "status":
                "failed", "output": "Votre implémentation dépasse la durée
                allouée pour la tâche !"}'
            return (newjson)
        else :
            newjson = '{ "message": "Erreur !", "status": "failed", "output":
                "Une erreur est survenue."}'
            return newjson
    else:
        return ("None")



run(host='ec2-54-194-189-76.eu-west-1.compute.amazonaws.com', port=8080, debug
    =True, reloader=True)

}
```

### 5.2.4 Remarks regarding the integration of our checks

While extending Pythia with coding checks, there are two approaches to make use of those checks. A first way would be to create exercises that would put students in situations where we expect them to make errors we want to correct. In other words, we make specific exercises just for coding checks. The second approach would be to add coding checks to other exercises so that students would get both feedback from functional check and coding checks. For our validation part described in next chapter, we decided to write a specific exercise so that the result are really focused on coding checks. That being said, if Pythia is extended with such checks in the future, teachers would have the opportunity to chose either way by telling to the Pythia development team what he wants.

### 5.2.5 Teachers interactions with Pythia

So far we have talked about how we implement our work on Pythia but not about how could teachers use that platform, how would they use Checkstyle, what do they have to do concretely. In fact, teachers do not interact directly with Pythia, they need to contact its development team to use the platform because they do not have all the access to it. The Pythia team serves as an middleman between teachers and the platform.

If a teacher wants to provide his courses using the platform, he needs to give corresponding PDF files to the Pythia crew which then uploads it on the server and make it visible on the web interface. Once it is done, students can subscribe to those courses and read them directly on the website or download the PDF.

For the exercise, it is almost the same. A teacher has to provide exercises in a text format but also specify what kind of correction (functional checks or performance indicators checks or other) he wants the platform to perform and what kind of feedback he wants to receive as well as how he wants it. Given all those specifications, the Pythia team implements both exercises and checks according to what the teacher ask them to do. For the feedback, the teacher can ask to receive it directly (for example by mail) or ask the team to make further computations on that feedback and then forward the results. The teacher is also free to chose the format of that forwarded feedback (for example, do they want the code as a Java file, the results as an XLS file, etc.).

If Checkstyle would be used on Pythia, it would require someone in the development team to be able to use that tool from the implementation of the configuration file to the implementation of customs detectors. Since the choice of checks to use is up to the teacher, he needs to know how the tool can be used and what it can look for. In the end, teachers would need to be aware of the possibilities that Checkstyle can offer while at least one member of the Pythia team would need a good knowledge about that tool.

# Chapter 6

# Validation

In this chapter, we describe the three experiments we made.

## 6.1 First experiment

| **Objectives:** |
| --- |
| Integrate Checkstyle to Pythia |
| Create an exercise for students |
| Write custom checks |
| Collect feedback from students |

The objectives for this first experiment were simple. If we wanted to use Checkstyle on Pythia, we had to explain how we can integrate the tool on the platform. To do so, we did not only had to explain how to install the tool on the platform but also how it could be used for an exercise. Since it is the first time we explain how we used Checkstyle, we also wanted to write customs checks so that we could evaluate if it is a difficult task or not. After the completion of the exercise by the students, we have collected their feedback.

### 6.1.1 Methodology

As announced in the previous chapter, we decided to use Checkstyle to provide a set of coding checks for Pythia. Using that tool has facilitated our task a lot since it already provides a lot of coding checks. In this section, we describe the process we went through to provide Pythia with such a set of checks. Keep in mind that we are focusing on Java beginners which means that we need basic checks.

#### Create an exercise for students

To evaluate students' coding style, we needed them to write programs which can be achieved by providing them exercises to resolve. In order to illustrate the need of coding checks, we created one exercise that then has been integrated in Pythia. With that exercise, we tried to

put students in situations where they were more likely to make a mistake for which we had a check. Therefore, students tried to resolve our exercise then checked their code with coding checks and corrected themselves using the feedback generated by Checkstyle.

#### Choose checks

In the corresponding Background chapter section, we said that Checkstyle groups its checks into several categories and that the most interesting category for us is the one of "Coding Problems". Looking at these specific checks, they are more than 40 and it is clear that depending on what is needed, it would not be appropriate to use all of them. Exercises that are implemented on Pythia are not large projects with hundreds of lines of code and do not require the use of complex data structures. When a teacher creates an exercise, he should be able to predict most of the code that will be provided by students. Thanks to that piece of information, Checkstyle should be configured accordingly. For example if we know an exercise does not require a switch case, there is no reason to use the "MissingSwitchDefault" check. A Checkstyle user should always be aware of what he wants the tool to look for. Beside case-dependent checks, there are some that are more generic and could be applied on all kind of exercises. Checks such as "OneStatementPerLine" are always useful regardless of what needs to be implemented for the exercise. Teachers that would want to use Checkstyle should be aware of what the tool can help them to check and based on that chose which check to use. To do so, he could either visit directly the Checkstyle website or either the Pythia development team could tell him (for example by providing a list of the detectors) what they can look for using that tool.

#### Integrate the exercise and checks to Pythia

As one of the goals of this thesis is to provide Pythia with a set of coding checks, the exercise that is described in the next section has been uploaded on that platform. For that purpose, we will explain how we can create a Pythia web page that host our exercises and display the feedback to students. We want both the exercise and the generated feedback to be clearly understandable by students.

### 6.1.2 The exercise

Now that all the steps of our solution have been explained, we can go further and talk about what has been done concretely.

As an exercise, we really want to observe if students have bad coding style while writing some expected patterns and so we do not want to let them too much freedom. To do so, we will explain in detail what they should do and what kind of instructions they should use.

#### The Nqueens problem

We chose to ask students to resolve the Nqueens problem. To limit the way they could manage to do it, we give them precise instructions. The expected program takes an array as input

and prints in the console whether that array represents a chess board that is a solution to the Nqueens problem. In order to keep this exercise simple, the input array is a one dimension array whose elements are integers that represents the line at which a queen is placed on the board. For example, if the first element is 5, it would means that the queen on the first column of the board is placed on the fifth line and so on. To resolve this problem, we ask students to fill a class template which contains a main method and three other methods. In the main method, students have to check the size of the input array since there is no solution for boards whose size is smaller than 4 and then call a first method. This one should only call the two auxiliary methods and prints if the board is a solution based of the results of two other methods. We defined those two auxiliary methods so that one checks if there is any line conflict thus if there are queens that are placed on the same line of the board, and the second one would check diagonal conflicts. The expected solution to the exercise is shown in the code fragment 6.1.

**Code 6.1:** The exercise resolved as expected

```
/**
 * The Goal of this class is to check if a given board is a solution to the
    Nqueens problem.
 */
public class NqueensSolverBis {

        /**
         * The user has to give an input board to check.
         * In this main method, you should only check the size of the board (
            is there
         * a solution for all size ?) and then call you method.
         *
         */
        public static void main(String[] args){
                int[] inputBoard={2,0,3,1};
                if(inputBoard.length<4){
                        System.out.println("There is no solution for the input
                            board.");
                }
                else{
                        isSolution(inputBoard);
                }
        }

        /**
         * This method should call all the auxiliary methods needed to check
            the board and
         * print in the console if the board is a solution or not.
         * @param board is the board to check
         */
        public static void isSolution(int[] board){
                if(checkLine(board) && checkDiagonal(board)){
                        System.out.println("The board is a solution !");
                }
                else{
                        System.out.println("The board is not a solution !");
                }
        }

        /**
         * This method should check if there is any line conflict in the board
            .
         * @param board is the board to check
         * @return true if there is no line conflict
         */
        public static boolean checkLine(int[] board){
                int line1, line2;
                for(int column1=0; column1<board.length-1; column1++)
                {
                        for(int column2=column1+1; column2<board.length;
                            column2++)
                        {
                                line1 = board[column1];
                                line2 = board[column2];

                                if(line1 == line2){
```

```
                                        return false;
                                }
                        }
                }
                return true;
        }

        /**
         * This method should check if there is any diagonal conflict in the
             board.
         * Hint: The method "Math.abs" could be useful.
         * @param board is the board to check
         * @return true if there is no diagonal conflict
         */
        public static boolean checkDiagonal(int[] board){
                int line1, line2;
                for(int column1=0; column1<board.length-1; column1++)
                {
                        for(int column2=column1+1; column2<board.length;
                            column2++)
                        {
                                line1 = board[column1];
                                line2 = board[column2];

                                if(Math.abs(line1-line2) == column2-column1){
                                        return false;
                                }
                        }
                }
                return true;
        }
}
```

### Choice and implementation of checks

In order to choose which checks we need, we have to go through the exercise again. This section introduce some detectors that we implemented ourselves. We do not expect teacher to do that but as we already discuss, that is the kind of work someone in the Pythia team should be able to do if Checktyle was used. The implementation of such detectors are not that hard since the API is well documented on the Checkstyle website [6]. The difficulty comes from the AST structure but as soon as the user is familiar with that structure, it is straightforward to implement custom detector.

#### Checking the number of method

Because we provide students a template to fill, we do not expect them to write additional methods. We implement a "MethodLimitCheck" (see code fragment 6.2) that we customize so that it raises a warning of the program contains more than 4 methods (as the template counts 4 methods already). Note that we set the default limit value to 2 but another value can be specified into the Checkstyle's configuration file (as shown in the code fragment 6.5). We have implemented it with the value 2 because it was more convenient to test it with that value, there is no further reason behind it. For all detectors that have a default value for something, if there is a mutator method in their implementation, that value can be changed directly using the configuration file. It is a really practical feature since it allows the user to not have to modify the implementation of the detector or to dig into its code. The tool can perform that check simply by counting the number of method definition tokens in the AST.

**Code 6.2:** The method limit check

```
public class MethodLimitCheck extends Check
{
    private static final int DEFAULT_MAX = 2;
    private int max = DEFAULT_MAX;

    @Override
    public int[] getDefaultTokens()
    {
        return new int[]{TokenTypes.CLASS_DEF, TokenTypes.INTERFACE_DEF};
    }

    @Override
    public void visitToken(DetailAST ast)
    {
        // find the OBJBLOCK node below the CLASS_DEF/INTERFACE_DEF
        DetailAST objBlock = ast.findFirstToken(TokenTypes.OBJBLOCK);
        // count the number of direct children of the OBJBLOCK
        // that are METHOD_DEFS
        int methodDefs = objBlock.getChildCount(TokenTypes.METHOD_DEF);
        // report error if limit is reached
        if (methodDefs > this.max) {
            log(ast.getLineNo(),
                "too many methods, only " + this.max + " are allowed");
        }
    }

    // Setter for the max limit
    public void setMax(int limit)
    {
```

```
        max = limit;
    }
}
```

### Checking the input

The main method should check the size of the input array before calling the next method. To do that, we expect and we want that students use an if-instruction. As it is a really case dependent check, we have to implement it as shown in code fragment 6.3. While writing a Checkstyle detector, we have to know from which kind of AST's tokens we want to start the search. In this case, we are looking for a method definition token. Then, we check the method name as we are looking for the main one. Once found, we just make sure that there is an if-instruction and if it is not the case, we raise a warning. Note that this check is not really accurate since students could write an if-instruction for anything else so that the check will not see any problem. We hope students would check it as expected but results describe in section 6.1.2 should tell if that is a good assumption.

**Code 6.3:** The input check

```
public class InputCheck extends Check
{
        @Override
    public int[] getDefaultTokens()
    {
        return new int[]{TokenTypes.METHOD_DEF};
    }

        @Override
    public void visitToken(DetailAST ast)
    {
                DetailAST main = ast.findFirstToken(TokenTypes.IDENT);
                if(main.getText().equals("main")){
                        DetailAST body = ast.findFirstToken(TokenTypes.SLIST);
                        if((body.findFirstToken(TokenTypes.LITERAL_IF)==null))
                            {
                                log(ast.getLineNo(),"input.check");
                        }
                }

    }
}
```

### Checking conditions

Java beginners tend to make conditions more complex than they really are. When they do not fully understand how Boolean works, they sometimes write "*==true*" or use "*==false*" instead of "*!*", the NOT operator. Checkstyle brings in a "SimplifyBooleanExpression" which, as its name suggests, would notify the developer if a Boolean expression can be simplified.

### Checking for-depth

To check diagonal conflicts, we expect students to use two iterators to go through the array; one to take each element one by one and the other to compare to other elements. We use the "NestedForDepth" check and we set the limit at 2 so that a warning is raised if the tool

detects nested-for with a depth higher than 2. We could also add a warning if students use a depth lower than 2 but as they are beginners, we do not expect them to find another way to implement the method but maybe they would come up with something correct with a lower depth that is why we chose not to set a lower bound.

### Checking nested if

We want the students' code to print that the board is a solution if both auxiliary methods return true (if both have detected no conflict). We can expect that students would use nested-if and print a result accordingly. With the check shown in code fragment 6.4, we want them to remplace"if(A){ if(B) {...} } " with "if(A && B){ ... }". It is not always good to merge conditions especially when they are big or complex but it is not the case here.

**Code 6.4:** The nested-if check

```
public class NestedIfCheck extends Check
{
        @Override
    public int[] getDefaultTokens()
    {
        return new int[]{TokenTypes.LITERAL_IF};
    }

        @Override
    public void visitToken(DetailAST ast)
    {
                // Go to the body of the if statement
                DetailAST objBlock = ast.findFirstToken(TokenTypes.SLIST);
                if(objBlock !=null){
                        // Get the first instruction of that body
                        DetailAST firstChild = objBlock.getFirstChild();

                        // Get the last instruction (using previous because
                            the last one is "}")
                        DetailAST lastChild = objBlock.getLastChild().
                            getPreviousSibling();

                        // Check if the first instruction is an "if statement"
                             and if there is no other
                        // instruction in the body. That would means that the
                            two conditions could be combined.
                        if(firstChild.getType()==TokenTypes.LITERAL_IF &&
                            firstChild.equals(lastChild)){
                                log(ast.getLineNo(),"nested.if.check");
                        }
                }
    }
}
```

### Auxiliary checks

We add two more checks to those that were just explained. The first one is the "NestedIfCheck". We are not sure students would use nested-if outside the case explained in the previous check but still if for any reason they use it, we do not want them to code it with a depth greater than 2. Maybe that check would never be used in this exercise but it is more as a precaution since students can be so unpredictable when they code. The other additional

check is the "OneStatementPerLine" that will ensure that students will not write more than one instruction per line.

### Writing the configuration file

After that checks are chosen or implemented, we write a configuration file (see code fragment 6.5). That file allows Checkstyle users to specify which checks they want to perform and they also can customize those checks by changing a variable if there is one (for example, in this case we can set the method limit check to 4) or modifying the message that is display when a warning is raised. We expect a Pythia member to be able to write that file but a teacher could be able to do it as well since it is not that difficult. The first line is mandatory, "Checker" is the root module of configuration file. The second line defines the severity of errors that would be detected by the detector that are specified in the rest of the file. Because we think that errors handled by our set of detector are not that critical, we chose to go with the "warning" severity level (more details are available on the Checkstyle's website [6]). Then, the rest of the file contains the set of detectors we have decided to go with.

For each of them, there are some attributes that can be changed here to customise them without going into to their implementation. There are two main attributes that should be noticed.

The first one allows user to customise the message that would be displayed if the detector finds what it has been implemented to look for. To do so, as shown for the SimplifyBooleanExpression (for example), we specify the message key of the detector (in this case: *"simplify.expression"*). It is a bit hard to find since it requires to look into the code but there are some tools/plugin to help the user. For example, we write that configuration file using Eclipse which finds the key by itself. Note that the line "*¡metadata name="net.sf.eclipsecs.core.comment" value="Updated message"/¿*" comes from the Eclipse and is used just to remind ourselves what modification we wanted to do (it is optional). Once the message key has been found, the custom message can be written as the value of that attribute.

The other attribute is used to set a value to a variable used in the implementation of the detector (for example the limit of methods in the MethodLimitCheck). Again, the name of the variable has to be found beforehand but it can also be easily done using Eclipse for example or by visiting the website since variables that can be customised are specified in the description of the detector. If a user writes a custom detector, he needs to keep the variable name in mind which should not be a problem since he choose it himself. Note that Eclipse is able to detect that variable so that the user can change it through it.

It is important to notice that since it is a XML file, special characters are written using their corresponding code.

**Code 6.5:** Configuration file

```
<module name="Checker">
  <property name="severity" value="warning"/>
  <module name="TreeWalker">
    <module name="SimplifyBooleanExpression">
      <metadata name="net.sf.eclipsecs.core.comment" value="Updated message"/>
      <message key="simplify.expression" value="Expression can be simplified.
          You can remove &quot;==true&quot; and use &quot;!&quot; instead of &
          quot;==false&quot;"/>
    </module>
    <module name="NestedForDepth">
      <metadata name="net.sf.eclipsecs.core.comment" value="Limit set to 2"/>
      <property name="max" value="2"/>
      <message key="nested.for.depth" value="Nested for depth shouldn't be
          higher than 2."/>
    </module>
    <module name="NestedIfDepth">
      <metadata name="net.sf.eclipsecs.core.comment" value="Limit set to 2"/>
      <message key="nested.if.depth" value="Nested if-else depth shouldn't be
          higher than 2."/>
    </module>
    <module name="InputCheck"/>
    <module name="OneStatementPerLine"/>
    <module name="NestedIfCheck"/>
    <module name="MethodLimitCheck">
                  <property name="max" value="4"/>
    </module>
  </module>
</module>
```

**Results**

For the exercise part, we let students deal with the exercise by themselves without giving any kind of assistance. The purpose is to simulate a real exercise with corresponding implemented coding checks. While preparing the Nqueens problem, we thought of the way we expect students to resolve it and then create checks that would help them correct mistakes we were waiting for. Since students have submitted their answers, we can have a look at their code and see if there is anything that we did not expect or that is worth to be noticed. Those results come from 10 students who accepted to help us by trying to resolve the exercise.

Most of the submitted codes are not that far from the solution we were expecting and that was shown in code fragment 6.1. The main difference comes from the implementation of the checkDiagonal method which is not a big surprise given the fact that it is probably the most complex method of the exercise.

Only 20% of the participants check the diagonals of the board as we do. Others use two conditions or do not really understand the "Math.abs" method that we recommend to use as they check "Math.abs((j-i))" then "Math.abs((i-j))". The most unexpected code comes from a student who decides to reimplement the board as an array with a size of 64. Because he forces a size, his implementation does not really behave as we wanted to.

Fortunately, students do not come with the same code as in our solution because if that was the case, we would not have much to learn from. In fact, only two students manage to provide a solution which does not raise any warning from our set of detectors. In general, students code have generated 1 or 2 warnings.

Almost all of them failed to check the input. As explained in the specifications of the exercise, we wanted them to check the length of the board in the main method. Givent the result, it is clear that our specification ("*is there a solution for all size ?*") and our detector's message ("*Try to check the input before using it*") were not clear at all and not precise enough. If that detector would be used in the future, its message should definitely be improved.

While looking at the output of the exercise, only 2 or 3 of the seven detectors proved to be useful as their corresponding warning was raised in the output of our checking. The reason is that the exercise is kind of easy and our specifications does not give students a lot of freedom in terms of how they have to code the class and also some of the detectors are more preventive. For example, while preparing the exercise we expected students to respect the coding convention of writing one statement per line and so, not to raise the warning from the corresponding detector which was the case.

It is also interesting to note that there is a student who submitted different codes several times even after he managed to resolve the exercise without any error. When we look at all the modifications that he made, we can think that he tried to understand how our tests worked and what are the limits. After having implemented the class correctly, he switched the final messages. At first, he printed that the board is a solution if no conflict is detected and then he changed it so that whenever a conflict is detected in the input board, his code considered it as a solution. As no functional checking is implemented, in both of those cases there is no problem from our customized Checkstyle. It is very pleasant to see that there is a student that is curious enough to try different things as he did.

### 6.1.3 A quick survey

After students have tried to resolve the Nqueens problem until no warning is raised, we decided to ask them some questions through a quick survey. Using their answers we wanted to evaluate two things.

At first, we wanted to know if both the exercise and our warning messages for our checks were well written. Were they clear enough, did students understand what we wanted them to do and what we asked them to correct (in the case warnings were raised).

The second thing we wanted to evaluate is the students' interest in such source code checking. They are the one who uses Pythia, the one that benefit the most from the platform with the teachers. For those reasons, their opinion is important to know. We asked them what they think about style checks and whether they think that such checks would be good to have in Pythia.

#### The feedback from students

Here are the questions we asked to students and their answers we received for our survey. Note that those answers come from students that tried to resolve the exercise and from which we receive their code.

#### How many warnings were generated for your first submission?

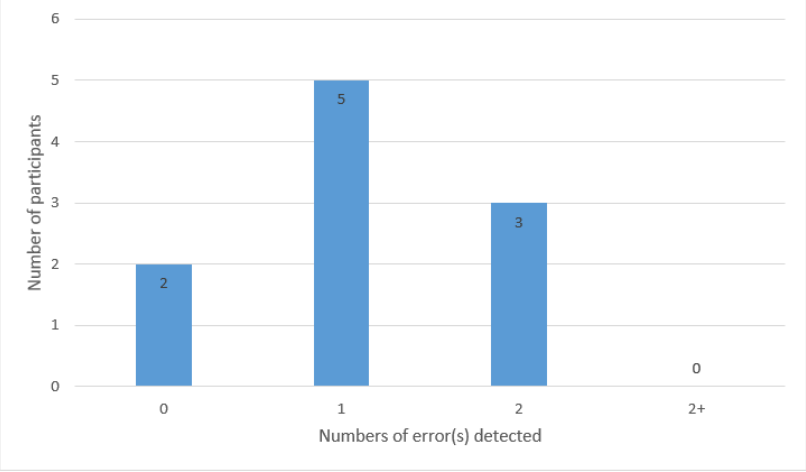


**Figure 6.1:** Number of error(s) detected for the first submission

As shown in figure 6.1, 80% of the students have generated one or more error(s) including the error that is detected with the InputCheck. It confirms that our specification for the main method was not good enough and that how we wanted them to implement it was not clear at all. We should have explicitly written that we wanted them to check size of the board and if it was too small to handle it there.

**Was the generated feedback explicit enough? Do you understand what we want you to correct?**

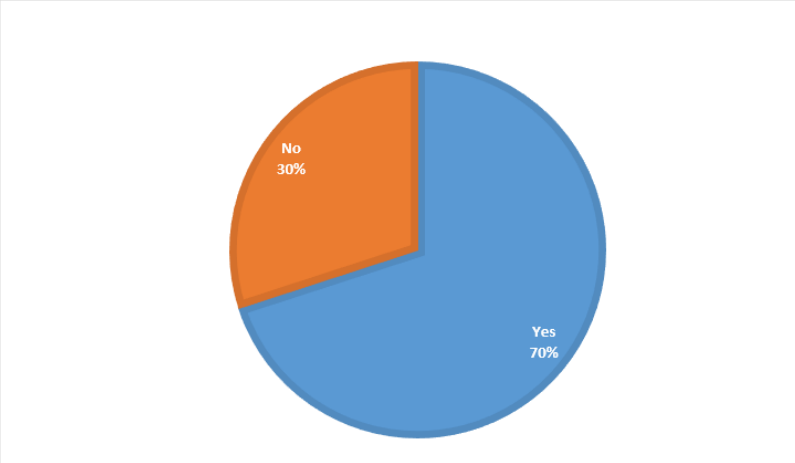


**Figure 6.2:** Was the generated feedback explicit enough?

The figure 6.2 shows that 30% of the participants think that our messages are not clear enough and so they do not really understand how to correct themselves. Using the values from the previous figure 6.1 and this one as well as looking at submitted codes, the message that is not good enough comes from the input detector which confirms what we thought and what we have already discussed in the section 6.1.2. If their answer is "no", we also asked them why. Two answers were particularly interesting. The first one is from a student who does not understand what is the problem while reading the input message which again confirms what we already know. The other one is from a student which managed to resolve the exercise but is wondering if its code is correct. In fact, when Checkstyle does not detect any error, it only prints *"Starting audit..."* and then *"Audit done"*. If a programmer is not familiar with Checkstyle, he might not understand what does that means. We should have prepared a more precise message to print when no error is detected and everything is fine according to the tool's point of view. The fact that a student wrote an incorrect code without being notified about that shows that we clearly should have added functional checks. In our exercise, we wanted students to write a program that solves the Nqueens problem but we have not check if it was the case. Even if the answers were positive, we feel that both what we wanted them to do and the feedback were not clear and precise enough.

**Do you think that such coding checks could help students to improve their coding style?**

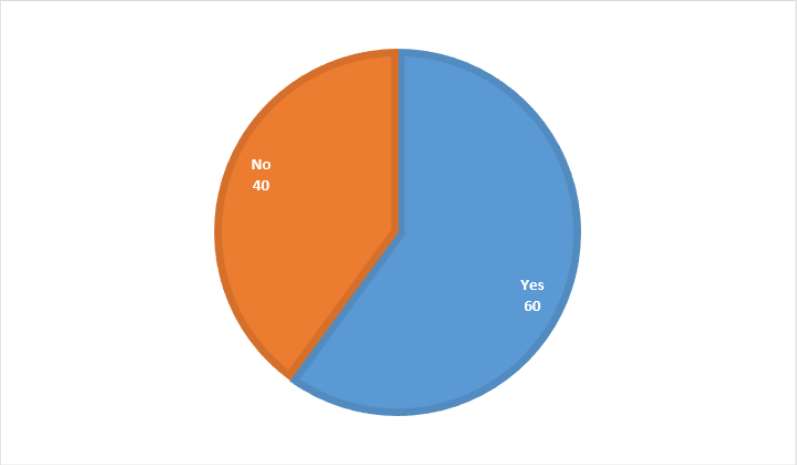


**Figure 6.3:** Do you think that such coding checks could help students to improve their coding style?

As shown in the figure 6.3, 40% of the students think that coding checks are not that useful to help them. However, even if 60% of positive answers seems a bit low, that number is not that bad given all the flaws we are already aware of and that we have already discussed (specifications not precise enough and lack of functional checks).

**Do you think that such coding checking have their place in the Pythia platform?**

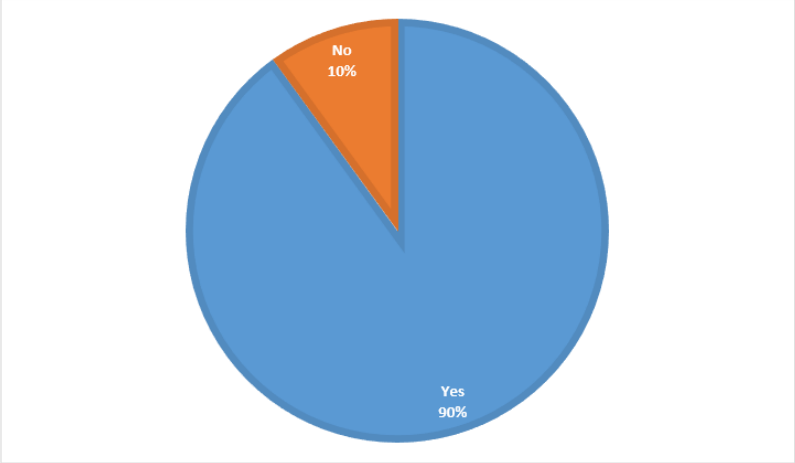


**Figure 6.4:** Do you think that such coding checking have their place in the Pythia platform?

Since students are the main Pythia users with the teachers, we think that they are in a good position to tell if extending Pythia with coding checksis legitimate or not. With this question we also want to know if they think that such extension is consistent from their point of view. For that question, we received 90 percent of positive answers as displayed in the figure 6.4.

Based on that result, we can think that extending Pythia in the way we want to would not surprise them. They would find it legitimate to integrate coding style check on the platform.

It is really a good result as it means that such checking would be welcome positively. Of course, the previous graph shows that there is still work to do before releasing that checking.

### 6.1.4 Conclusion

#### What we can take from our experiment

Both the exercise and the survey bring in positive and negative things but in the end our results are interesting.

First, the exercise is implemented successfully by the students that took part in our experiment. That is not a surprise since it is not very difficult and because with our specification we force the way to implement the class. By doing so, we were hoping that students would made errors we were prepared for.

Unfortunately, the easiness allows them to succeed almost immediately. The most common error is raised by the input detector which tried to check if students handle differently boards that are too small. We have already discussed several reasons like, for example, the fact that the specifications might not be explicit enough.

We also notice that the message of that detector is not good enough as students sometimes had trouble correcting themselves starting from that generated feedback. Because of both the lack of freedom and the fact that the exercise is small, all the detectors do not prove to be useful to the students as the expected errors were not made.

Looking at submitting code, there is a student that tried to see how our tool works and wrote some code that is not functional but is fine from Checkstyle point of view. It is not really a problem since it is not the goal of such coding but it seems that it would be better to use style checking additionally to functional checking.

In our case, a student that has a good coding style according to our detectors cannot know if its implementation works as expected for the exercise. While resolving an exercise, not having a verification might be frustrating. Moreover, a student might think that he is doing the exercise right while in fact his code is not good.

As we tried to teach programming, we should always make sure that while we ask students to implement something, their code has the behavior that we described in the specifications.

The answers provided by students for the survey are very interesting as well.

As already said, the exercise is maybe too easy as most of the students only raised one warning which is from the input detector. Overall, our feedback was clear and complete enough except for that detector. The answers seems to confirm the fact that that message in particular is not good enough. Also, we should have prepared a message to tell that the code is well written if no error has been detected. When preparing that exercise, we might be too focused on the error message and not enough on the success message.

Opinions are mixed about the usefulness of our checking. Again that could be explained by the fact that the exercise was too easy and small. Not enough errors were detected and because of that, students undervalued what coding checks could bring to them and how well it could help them in general (it might not be useful for all the students).

Answers to the last question tells us that extending Pythia in the way we want to, so with coding checks, would not surprise students. They seem to think that it would be a legitimate

addition to the platform. Knowing that students perceive it positively is pleasant since we tried to promote another way to help them improve their programming skills.

### The limits of our experiment

Regarding all those results we get from our experiment, we see that there are limits to our validation part.

The main one is the low participation rate of the students. We were hoping that students would be more cooperative and that more than 10 would provide us some feedback. That being said, we can understand that students prefer to spend their free time on doing something other than resolving an exercise with no reward. Looking for volunteers whatever the task is has always been difficult for all kinds of experiment with no reward. Our experiment was unfortunately no exception to that observation.

The exercise was too small and too easy. We will not talk too much about that as we already discussed it previously but it has limited the possibility of students making errors. Students also undervalued coding checks even if we are convinced of their utility. It also limits the number of detectors we could have used since there are not a lot of different ways to implement our class as we specified it. However, it was difficult to predict beforehand how the exercise would be resolved and perceived. We think that it might be the first time students use a tool such as Checkstyle and that is why we were afraid of creating something that might have been overcomplicated.

Thanks to a curious student, we think that it would be better to use our coding check in addition to functional check. In fact, we already know that but we wanted to see interactions between students and coding checks. We wanted students to be focused on their style and thought that adding functional might distract them from that. However it is clear that it would have been better to add functional checks to help students to know if they were doing it fine ands it is also better from a teaching point of view. Furthermore, it brings in another limit. Without functional checks, it is easy to write something to make the code fine from our detectors point of view. Detectors looking for specific patterns. If those patterns are not met, the code will be considered as well written. With the set of detectors we chose for our exercise, if a student submit the template we provided without filling in it, only the input detector would raise a warning. Then, if the student just add the corresponding “if” in the main method, no error would be detected which is really not good for our experiment. Fortunately, no student did that but that proves again the necessity to use both coding style checks and functional ones.

### Threats to validity

Event though our results are interesting, we can not affirm that the opinions that we collected are representative enough because we did not get the chance to have a lot of students. As we already mentioned, since the exercise was maybe too easy, it is possible that students did not really understand the interest of using Checkstyle or what we mean with coding style checks.

**Conclusion of our experiment**

To conclude this validation part, there is room to improve our exercise in order to respond to the limits we have just discussed. We have not had the opportunity to make another exercise because we did not have enough time to do so after the first one. Additionally, we did not think we would have had more participants. If it was not the case we would have made one that is more complex with more freedom for the implementation. We would also improve both the specifications of the exercise and the messages provided by our detectors. Also, if we prepare a bigger exercise, we would have to adapt our use of Checkstyle so that we would include more detectors as well. Finally, we can see that even with a low participation rate and a small exercise, we can learn a lot from the submissions and all we can take from the experiment we done is still useful and interesting.

## 6.2 Second experiment: Testing Checkstyle ourselves

| **Objectives:** |
| --- |
| Try Checkstyle ourselves |
| Use the tool with the default configuration file |

Since we could not use beginners students for this second experiment, we decided to try Checkstyle ourselves. Because we became familiar with that tool after we had analysed it and extended it with custom checks, we could have been a little biased if we would have implemented an exercise created by ourselves and knew which detectors were used. To counterbalance that, we decided to take an exercise from the Internet and used Checkstyle with its default configuration file which is called sun_checks.xml (it is provided with the tool). As suggested by its name, it is supposed to check if our code respects the Sun coding convention.

### 6.2.1 The exercise

We found an exercise on the Home and Learn website [18]. That exercise asks people to implement a Java class that takes a String of football results in input and prints a set of statistics computed by parsing that String. As an example, the author gives the following input String: "*Manchester United 1 Chelsea 0, Arsenal 1 Manchester United 1, Manchester United 3 Fulham 1, Liverpool 2 Manchester United 1, Swansea 2 Manchester United 4*". From that string, the output of the class to be implemented should consist of statistics for Manchester United and should look like:

- Number of wins = 3
- Number of draws = 1
- Number of defeats = 1
- Goals scored = 10
- Goals conceded = 6
- Number of points = 10

### 6.2.2 Our implementation

Our first implementation of the class is shown in the code fragment 6.6. It is maybe not the best way to do that but in general we do not expect students to write the perfect code as a first submission neither.

**Code 6.6:** Our implementation of an exercise found on the Home and Learn website [18]

```
package reminder;

public class FootballResult {

  public static void main(String[] args){
        tring results = "Manchester United 1 Chelsea 0, Arsenal 1 Manchester
            United 1, Manchester United 3 Fulham 1, Liverpool 2 Manchester
            United 1, Swansea 2 Manchester United 4";
    parseResults(results);
  }

  public static void parseResults(String results){
        int[] matchsResults = getStats(results);
        System.out.println("Number of wins : "+matchsResults[0]);
        System.out.println("Number of draws : "+matchsResults[1]);
        System.out.println("Number of defeats : "+matchsResults[2]);
        System.out.println("Goals scored : "+matchsResults[3]);
        System.out.println("Goals conceded : "+matchsResults[4]);
        System.out.println("Number of points : "+matchsResults[5]);
  }

  public static int[] getStats(String results){
        int[] res = new int[6];
        String[] matchs=results.split(",");
        for(String s : matchs){
                String[] goals = s.trim().split(" ");
                int manchester=0;
                int opponent=0;

                if(goals[0].equals("Manchester")){
                        manchester = Integer.parseInt(goals[2]);
                        opponent = Integer.parseInt(goals[goals.length-1]);
                }
                else{
                        manchester = Integer.parseInt(goals[goals.length-1]);
                        opponent = Integer.parseInt(goals[goals.length-4]);
                }

                if(manchester>opponent){
                        res[0]+=1;
                        res[5]+=3;
                }
                else if(manchester==opponent){
                        res[1]+=1;
                        res[5]+=1;
                }
                else{
                        res[2]+=1;
                }
                res[3]+=manchester;
```

```
                res[4]+=opponent;
        }
        return res;
    }
}
```

### 6.2.3 The analysis of Checkstyle

When we checked our code using Checkstyle, the tool detected errors at almost each line of our implementation. In this section, we explain the errors that were detected and we corrected them as Checkstyle wanted us to do. Our thoughts about the feedback are discussed in the next section.

The first error found was the lack of comments. Checkstyle wanted us to comment our class and our methods and notified us with the message "*Missing a Javadoc comment.*" Note that the message specifies that we have to use Javadoc comments and so we have to use the corresponding format or the error would still be detected. For example, comments for methods with parameters have to include "*@param ...*" and those for methods that return a value have to include "*@return ...*".

As we wrote our class in Eclipse, some tabulations were added in our code. Because of that, Checkstyle notified us with the message "*Line has trailing spaces*". In fact, the tool wanted us to use only whitespaces for indentation and not tabulation. We also received the message "*Utility classes should not have a public or default constructor*". After we have added a constructor, Checkstyle told us to declare our class as Final. Then, we had to change our parameters to final as well.

Checkstyle asked us to do not use magic numbers but we found it was not really convenient to correct that in our case so we ignored that check. The line that contains the input string was considered as too long and the detector specifies that the maximum length is 80. To correct that, we divided the String. The most detected error was caused by the fact that we did not write whitespaces before and after operator such as "=", "+=" or "==" but also after keywords such as "*if*" and "*else*". The conventions also forced us to write "{" and "}" on the same line of the instruction that preceded them and there have to be a whitespace before them. However, we met a case where for the same "}", we received two messages that were in conflict with each other. For the line: *opponent = Integer.parseInt(goals[goals.length - 1]); }*

We got:

- " - *'}' should be on the same line.*"
- " - *'}' should be on a new line.*"

It was impossible to correct that since the error remained whatever we tried. If we wrote it on the same line, the second message was displayed and if we wrote it on the next line, we got the two messages.

### 6.2.4 Remarks

The comments detector is an interesting detector because we know from personal experience and based on the students' code we received that students often forget to comment their code. However, we were forced to use the Javadoc format. While we understand that it was because it is the format promote by the Sun conventions, we feel that it might be a bit too strict in some cases. Writing lines that are too long is something we do not always think of so the detector was interesting as well.

Corrections regarding tabulations and whitespaces around operators and keyword are corrections that we do not think are that important. Of courses, in long formula, it can help to identify the different parts but in our exercise, it was not very useful.

Surprisingly, we met the case where two messages that were in conflicts. We do not really know how it happened and it seemed that even if we tried to correct the error, the tool did not take that into account. Maybe it was just a bug even if we tried to relaunch the checks, the error remained.

The detector of magic numbers is rather situational. In our exercise, the code will not change and it will not be used again so it was not really important to avoid them here.

### 6.2.5 Threats to validity

Because we are familiar with the tool and we knew the style was checked, we were certainly influenced when we were implementing our solution. Maybe if we did not know what were the objectives, we would have produced a different code.

### 6.2.6 Conclusion

We decided to run the default configuration file and as expected errors that were detected were not always appropriate for our exercise. Some detectors that were used seemed too strict. It proves that it is important to use a configuration file that fits the exercise but also fits what we want to look for. Coding checks are often situational so using all of them would force the user to correct things that are not really errors. The "bug" we found is a bit weird since the two detectors have opposite messages but still check for similar pattern (they both raised an error for the same instruction).

Writing the best configuration file is not an easy task. The process we went through in that exercise could be a way to deal with it. While creating an exercise, knowing what kind of errors to look for is not always clear. Use all the detectors on an implementation of the exercise and then removes those that we do not feel appropriate can be a solution to write a good configuration file.

## 6.3 Third experiment

| **Objectives:** |
| --- |
| Create a more complex and well-written exercise |
| Describe teachers' methodology |
| Collect feedback from the volunteers |

We have decided to make a third and last exercise so that we can consolidate what we have learned so far as well as to explain more in detail the process that should be followed by a teacher to create an exercise and use Checkstyle. So, there are two main objectives that we want to validate with that experiment. First, from the results and conclusions of previous exercise, we identified some flaws in the design of our exercise such as the lack of functional checks. In the current experiment, we try to correct those mistakes by designing a third exercise that does not suffer from those flaws. We expect to learn more and draw new conclusions. Our second objective is to describe the methodology of making an exercise that uses Checkstyle. In other words, we want to validate the approach from the teacher's point of view.

The 10 participants of this exercise were students that are in Master 2 who volunteered to take part in this experiment. We asked them to resolve the exercise and run our tests by using a script that we have written, as soon as they completed the class template of the exercise described in this section without trying to optimize their code; since beginners are the true target audience of this thesis but since the volunteers for this experiment are more experienced [1], we did not want them to write code that was too complicated. Besides that because they are not beginners anymore, it might be interesting to see if they have developed bad habits that the checks look for. The objective of this experiment was to provide an exercise that is more complete than the previous ones (with functional checks) and to collect the opinion of more experienced students. We expected positive feedback for both the design of the exercise but also for the interest of using Checkstyle. We did not use Pythia for this experiment because it was not convenient since we do not have access to the online server and so we could not upload an exercise online.

### 6.3.1 The process of making an exercise

How is a teacher supposed to use Checkstyle if he wants to create an exercise and check the style? The first step is obviously to imagine a problem to resolve which will become the exercise to do. In our case, we decided to make an exercise about Poker, the cards game. The Java class that we wanted students to implement receives a set of 50 Poker hands. Using that data, the class then has to compute a set of statistics such as the number of hands that contains a pair or a flush but also the best and the worst hand in the input set. The template and the specifications of the class are described in more detail in the next section.

Before making the design of the exercise , a teacher would have to decide which checks he wants to apply on the implementations that he would receive from his students. In our case, we want to have functional checks to ensure the correctness of the implementation but we also

[1] Since we made this experiment during summer holyday, we could not asked students to do it. We then looked for volunteers among friends that agreed to help us.

want to check that they respect some coding conventions by using Checkstyle. More details about those conventions are discussed in the next section. As we do not expect teachers to be as familiar with Checkstyle as we are, we will use the classification of detectors that we have introduced in the section 2.1.2. So we assume that we have access to our classification so that we only have to chose a set of detectors among those for novices.

### The methodology

Figure 6.5 illustrates the methodology that has just been discussed, from a teacher's point of view.

1. Decide on the objectives of an exercise: Why would a teacher make an exercise? The reason is just to try to achieve some goals or objectives. In general, teachers make exercises to check if students have understood the theory and are able to use it to resolve problems. In our case, we wanted to try our some detectors that we classified as for novices on a well-written exercise that is more complex and to collect the opinion of more experienced students, acting as if they are novices.

2. Select the checks to be performed: To see if the objectives are met, there is a need of checks to be performed. Our checks has been chosen from the novice set of detectors. We went thought the entire set of novice detectors and took those that would be interesting for the exercise.

3. Design the exercise: At this point, a teacher has to decide what the exercise would look like. It can vary from a simple method to a template of a class to complete, a multi-class project or an exercise where nothing is provided at all and students are free to implement it as they want. In this experiment, we provided a template to complete because we still wanted to control the implementation but we also let more freedom than in the first experiment since we let volunteers the possibility to remove some methods from the template and to write auxiliary methods.

4. Configure Checkstyle: The fourth step would be to create the checks to perform. Since we worked with Checkstyle, we just had to configure it and write additional detectors if needed. In the case the exercise is provided to students through Pythia, a teacher would also have to contact the Pythia team to create the corresponding web pages and upload the configuration file on the server (so that the platform can use that file).

5. Try out the exercise: The best way to identify some flaws in the conception of an exercise or maybe to improve it is for the teacher to resolve it himself. This step is important to check if the exercise is well designed or too difficult or if there is anything that the teacher might not have thought of beforehand. If it is the case, he might have to go back to one of the previous steps (for clarity, we only put one arrow back to step 1 in figure 6.5).

6. Provide to students: The teacher has to provide the exercise to the students so that they can resolve it. It can be done with printed paper or by mail or through the Pythia platform.

7. Collect code, results or feedback: Depending on the exercise, a teacher can collect different things. If the purpose of the exercise is that students try to write the code with the

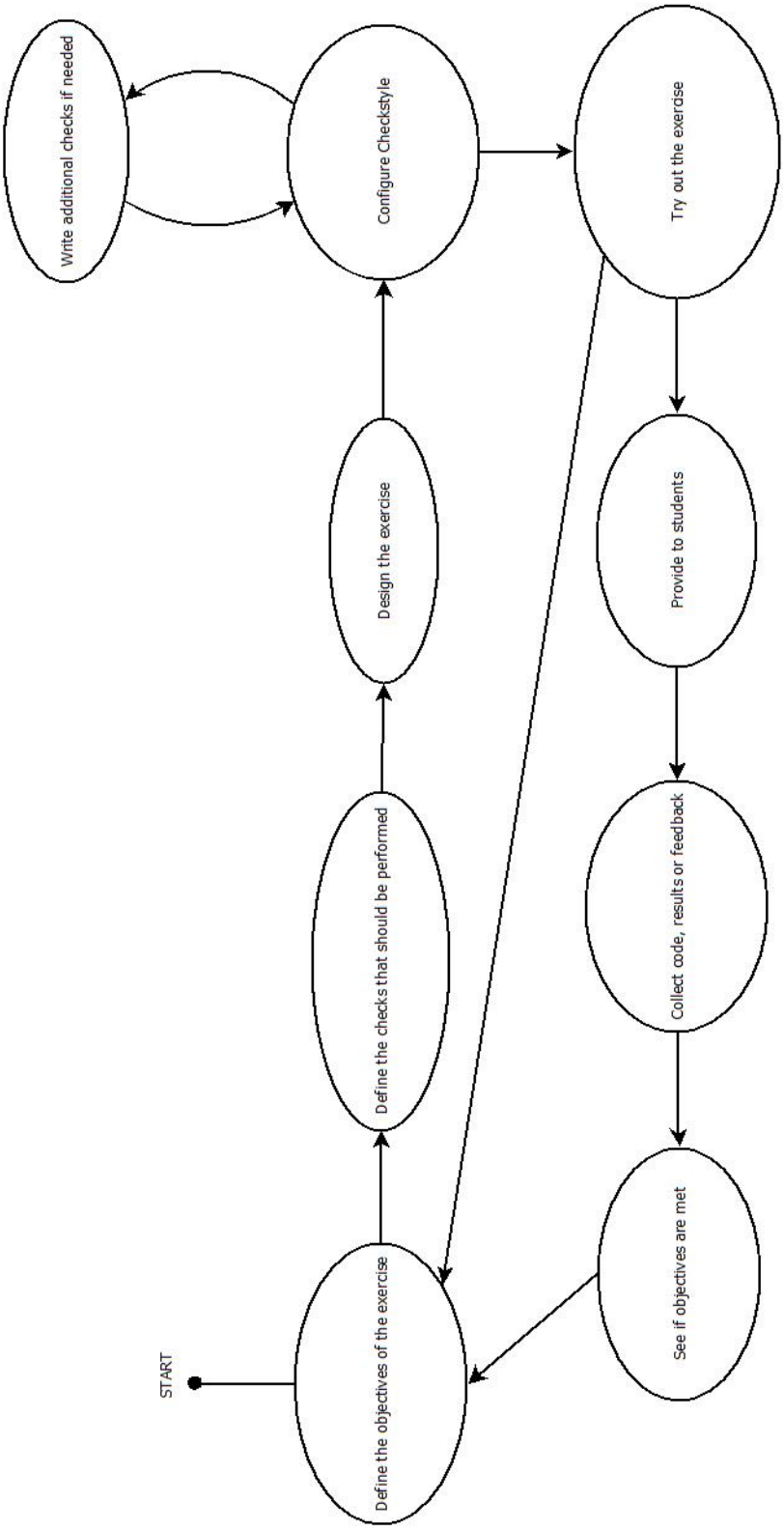


**Figure 6.5:** The process of making an exercise.

shortest execution time, he could just collect the durations, for example. In this case, we collected the output of the tests from the volunteers so that we could have a look at what mistakes they have made and which ones.

8. Check if objectives are met: If the objectives are met, the exercise is a success and if not a teacher can follow different approaches such as make another exercise or discuss with students what went wrong.

### 6.3.2 The Poker exercise

The template of the exercise is shown in the code fragment 6.7. The class to be implemented initially contains several methods. Note that Poker hands consists of combinations of 5 cards.

- The method "*computeStats()*" computes the statistics and prints them in the console. Statistics to compute are, the number of each combination found in the input file and the best combination of the set (the first found in case of draw).
- The method "*parseFile()*" is the parser. It should read the input file and store each Poker hand in an array.
- A method to handle each of the Poker combinations: those methods have to check if a Poker hand contains their corresponding combination.
- We also add several auxiliary methods to resolve the exercise: "*isBetterHand*", "*cardValueToInt*", "*getCardValue*", "*cardToInt*", "*getBestCombination*" and "*isBetterCard*". Those auxiliary methods can be removed if the participant prefers to implement his own solution. All the code will still be checked because we did not chose method dependent detectors.

For our functional checks, we asked students to store values in specific variables as explained in the specifications. We also added a difference with respect to the first exercise; auxiliary methods are allowed now but we still set a limit with our checks so that the amount of auxiliary methods used will not be too long. In fact, we wanted to give the students more freedom so that we might see interesting things that were not expected. However, we still have put some limits as we have used some metrics detectors. Also note that the template of the class is probably not the best to implement a solution but we wanted to divide the implementation in small and simple methods. We forced the use of one method for each combination because it is better for the readability and we wanted to keep a general structure, even though some combinations could have been handled in the same method. This is an exercise that we have designed ourselves but that was inspired by *the problem 54* on [13]; the problem is to compute the number of time the first player has a better hand than the second player (the input file contains 1000 pairs of Poker hands). We have modified and divided that input file so that it suits our tests.

**Code 6.7:** The PokerStats class

```
/**
 * PokerStats is a class that reads Poker hands in a file
 * and computes a set of statistics.
 */
```

```
public class PokerStats {

        /**
         * The following instance variable is used to set
         * the path to the input file. It will be used for
         * testing.
         */
        public String input_path;

        /**
         * The following instance variables should be used
         * to store the statistics that are computed from
         * the input file. There is one variable per statistics
         * to compute. The one_pair variable should store
         * the number of Poker hands whose best combination
         * is a pair, double_pair to store the number of Poker
         * hands whose best combination is a double pair and so
         * on.
         * The string best_combination should store the best
         * combination of the set.
         * If there is an equality, the first one found should be kept.
         */
        public int one_pair;
        public int double_pair;
        public int three_of_a_kind;
        public int straight;
        public int flush;
        public int full_house;
        public int four_of_a_kind;
        public int straight_flush;
        public String best_combination;

        /**
         * The constructor of the class
         * @param path is the path to the input file
         */
        public PokerStats(String path){
                one_pair = 0;
                double_pair = 0;
                three_of_a_kind = 0;
                straight = 0;
                flush = 0;
                full_house = 0;
                four_of_a_kind = 0;
                straight_flush = 0;
                input_path = path;
        }

        /**
         * The computeStats method is the main method of the class.
         * It should call other methods and assign the instance
         * variables with corresponding values.
         */
        public void computeStats(){

        }

        /**
```

```
     * The parseFile method parses the input file that is located
     * at input_path. The input file contains 50 Poker hands, one
     * per line. An example of Poker hand is "KS 4H 4D 9C AD".
     * Each pair of characters is a card: the first character is
     * the value (J = Jack, Q = Queen, K = King, A = Ace or a number)
     * and the second one is the suit (Hearts, Diamonds, Spades and
     * Clubs).
     * @return A 50-size array that contains the Poker hands.
     */
    private String[] parseFile(){

    }


    /**
     * The sortHand method receives a Poker hand as input and
     * returns that same hand but with the cards that have been
     * sorted from highest values to lowest.
     * For example, the returned string for "KS 4H 4D 9C AD"
     * should be "AD KS 9C 4H 4D".
     * @param hand is a Poker hand
     * @return the input hand whose cards have been sorted
     */
    private String sortHand(String hand){

    }

    /**
     * The containsOnePair method checks if a Poker hand
     * contains a pair. If it is the case, the method returns
     * the card value of that pair as a string, otherwise
     * returns an empty string.
     * @param hand is a Poker hand
     * @return an empty string or the card value
     *                       of the pair found.
     */
    private String containsOnePair(String hand){

    }

    /**
     * The containsDoublePair method checks if a Poker hand
     * contains two pairs. If it is the case, the method
     * returns the card values of those pairs separated with
     * a whitespace as a string, otherwise returns
     * an empty string.
     * @param hand is a Poker hand
     * @return an empty string or the card values
     *                       of the double pair found separated with
     *                       a whitespace.
     */
    private String containsDoublePair(String hand){
    }

    /**
     * The containsThreeKind method checks if a Poker hand
     * contains three cards of a kind. If it is the case,
     * the method returns the card value of those cards as
```

```
     * a string, otherwise returns an empty string.
     * @param hand is a Poker hand
     * @return an empty string or the card value
     *                       of the three kind found.
     */
    private String containsThreeKind(String hand){
    }

    /**
     * The containsStraight method checks if a Poker hand
     * contains a straight. If it is the case, the method
     * returns the highest card value as a string, otherwise
     * returns an empty string.
     * @param hand is a Poker hand
     * @return an empty string or the card value
     *                       of the straight found.
     */
    private String containsStraight(String hand){
    }

    /**
     * The containsFlush method checks if a Poker hand
     * contains a flush. If it is the case, the method
     * returns the highest card value as a string,
     * otherwise returns an empty string.
     * @param hand is a Poker hand
     * @return an empty string or the highest card value
     */
    private String containsFlush(String hand){
    }

    /**
     * The containsFullHouse method checks if a Poker hand
     * contains a full house. If it is the case, the method
     * returns the card value of the three kind and the pair
     * separated with a whitespace as a string, otherwise
     * returns an empty string.
     * @param hand is a Poker hand
     * @return an empty string or the card values
     *                       of the full house found.
     */
    private String containsFullHouse(String hand){
    }

    /**
     * The containsFourKind method checks if a Poker hand
     * contains four cards of a kind. If it is the case,
     * the method returns the card value of those cards
     * as a string, otherwise returns an empty string.
     * @param hand is a Poker hand
     * @return an empty string or the card value
     *                       of the four kind found.
     */
    private String containsFourKind(String hand){
    }

    /**
     * The containsOnePair method checks if a Poker hand
```

```
     * contains a straight flush. If it is the case, the
     * method returns the card value as a string, otherwise
     * returns an empty string.
     * @param hand is a Poker hand
     * @return an empty string or the card value
     *                       of the straight flush found.
     */
    private String containsStraightFlush(String hand){
    }

    /**
     * The isBetterHand method compares two Poker hands and
     * checks if the first one is better than the second
     * one (according to Poker Rules).
     * @param hand_1 is the first hand to compare with
     * @param hand_2 is the second hand to compare with
     * @return true if hand_1 is better than hand_2
     */
    private boolean isBetterHand(String hand_1, String hand_2){
    }

    /**
     * The cardValueToInt method returns the int that
     * corresponds to the value of the card. Values
     * such as 2,3,...,10 can be parsed to int but
     * J,Q,K,A have to be changed to a numeric value.
     * @param value is the value of the string
     *                        representation of the card
     * @return the numeric value of the card value
     */
    private int cardValueToInt(String value){
    }

    /**
     * The getCardValue method return the value of the card.
     * Given a card such as "AD", it should return the "A"
     * (it removes the suite).
     * @param card is a string that represents a card
     * @return the value of the card (2,3...,K,A)
     */
    private String getCardValue(String card){

    }

    /**
     * The cardToInt method parses the string that represents
     * a card and returns the corresponding numeric value.
     * If the input is "KH", the output should be 13.
     * @param card is the String representation of a card
     * @return The numeric value of the card
     */
    private int cardToInt(String card){
    }

    /**
     * The getBestCombination method looks for the best combination
     * and return an array. The first element is a String that
     * specifies the best combination found (i.e.:"Flush") and
```

```
         * the second one is the output of the corresponding method
         * (i.e.: the output of the containsFlush method).
         * @param hand is a poker hand
         * @return a 2-sized array with the combination found
         *                       and the output of the corresponding method
         */
        private String[] getBestCombination(String hand){
        }

        /**
         * The isBetterCard method compares two cards and
         * checks if the first one is better than the second.
         * It is useful since you might have to compare numbers
         * with Jokers, Queens, Kings and Aces.
         * @param card_1 is the first card to compare with
         * @param card_2 is the second card to compare with
         * @return true if the first card is better
         *                       false otherwise
         */
        private boolean isBetterCard(String card_1, String card_2){

        }

}
```

### 6.3.3 Configure the checks

Before the template is made, a teacher can focus on what he wants to check. As we said, in this case we wanted functional and coding checks. For the functional ones, we prepared several sets of Poker hands as input and we just wrote unit tests so that we called methods as implemented by students with those inputs and see if results are the ones we expected. For coding checks, a teacher could make use of our classification of detectors (see section 2.1.2). We decided to select some of the checks that we classified as interesting for beginners. Those checks for novices are good to detect frequent mistakes that students tend to make when they learn to code in Java. In fact, we went though all the novices detectors and we kept only those that might be useful for this experiment. The set of detectors that we used is shown in the configuration file below (see code fragment 6.8). Here is an explanation for the detectors that needed to be tuned. The others were chosen simply because we thought they would be interesting from the description we write in section 2.1.2.

- DescendantToken: We used it to limit the number of variable definitions in each method. The maximum limit was 10.
- LineLength: We have set a limit of 100 characters for instructions line because we did not want developers to write line that are too long. We defined the size of tabulations to 4.
- MethodLimitCheck: As we let volunteers write auxiliary methods, we did not want them to write too many methods so we set a limit of 20 given the fact that 10 methods were mandatory.
- MagicNumber: We allowed the use of the numbers -1, 0, 1, 2, 3, 4, 5, 11, 12, 13, 14 and

50 because those a numbers that we expected to find in the implementations; Numbers 0 to 5 because most of the computations are made on Poker hands that contains 5 cards, 11 to 14 are the corresponding number values of cards figures and 50 because the input set contains 50 hands.

- NestedIfDepth and NestedForDepth: We write arbitrarily limits that we thought was enough. We did not expect nested *for-loop* with a depth higher than 2.
- ReturnCount: We believed that having only one return statement in methods is a good practice but again we decided that arbitrarily.
- FileLength: To limit the length of the file, we set a limit of 900. Our own implementation which is not shown contains 800 lines so we decided to let a margin of 100 lines.

**Code 6.8:** The configuration file

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE module PUBLIC "-//Puppy Crawl//DTD Check Configuration 1.3//EN" "
    http://www.puppycrawl.com/dtds/configuration_1_3.dtd">

<!--
    This configuration file was written by the eclipse-cs plugin configuration
        editor
-->
<!--
    Checkstyle-Configuration: poketest
    Description: none
-->
<module name="Checker">
  <property name="severity" value="warning"/>
  <module name="TreeWalker">
    <module name="NeedBraces"/>
    <module name="AvoidStarImport"/>
    <module name="UnusedImports"/>
    <module name="RedundantImport"/>
    <module name="BooleanExpressionComplexity"/>
    <module name="DescendantToken">
      <property name="tokens" value="METHOD_DEF"/>
      <property name="limitedTokens" value="VARIABLE_DEF"/>
      <property name="maximumDepth" value="2"/>
      <property name="maximumNumber" value="10"/>
    </module>
    <module name="ModifierOrder"/>
    <module name="LineLength">
      <property name="max" value="100"/>
      <property name="tabWidth" value="4"/>
    </module>
    <module name="MethodLimitCheck">
                <property name="max" value="20"/>
    </module>
    <module name="AvoidInlineConditionals"/>
    <module name="MagicNumber">
      <property name="ignoreNumbers" value="-1, 0, 1, 2, 3, 4, 5, 11, 12, 13,
          14, 50"/>
    </module>
    <module name="ModifiedControlVariable"/>
    <module name="SimplifyBooleanExpression"/>
    <module name="SimplifyBooleanReturn"/>
```

```
    <module name="StringLiteralEquality"/>
    <module name="NestedForDepth">
      <property name="max" value="2"/>
    </module>
    <module name="NestedIfDepth">
      <property name="max" value="3"/>
    </module>
    <module name="ReturnCount">
      <property name="max" value="1"/>
    </module>
    <module name="MultipleVariableDeclarations"/>
    <module name="OneStatementPerLine"/>
    <module name="UnnecessaryParentheses"/>
    <module name="MissingSwitchDefault"/>
    <module name="DefaultComesLast"/>
    <module name="DeclarationOrder"/>
    <module name="JavadocMethod">
      <property name="logLoadErrors" value="false"/>
    </module>
  </module>
  <module name="FileLength">
    <property name="max" value="900"/>
  </module>
</module>
```

### 6.3.4 Results

Since our participants were experienced and the exercise was not really hard especially because precise specifications for the class and each of its methods were given, we did not expect them to fail the functional tests. They all manage to give the expected output for all our sets of input files. The results from Checkstyle are more interesting. The first time the participants ran the tool, a lot of errors were detected.

All participants made the following errors:

- Not writing braces for if-statement: The error was made when there is one instruction in the if-body. It is not a big mistake but we think that it is better to use braces for the readability. Furthermore, it prevents beginners to forget the braces when adding instructions in the body. It is not a surprise since volunteers were experienced programmers.
- Using magical numbers: Since we were expecting some value to be used, we configured the detectors to allow those expected values (see the configuration file). They used others numbers but they did not all use the same. That proves that it was possible to resolve the exercise without those numbers
- Missing Javadoc: They write their own methods but they all forget to comment them. The problem was not that they write comments that was not Javadoc but that they did not write comment at all. Forgetting to comment the code is probably one of the most common mistake and it is not a surprise to see that they all fail to do so.
- Too many returns: Writing several returns can be useful in some cases but we thought it is a good habit to use only one because it is better for the readability.

The other warnings raised were:

- Line too long: Some of the participants write some line that were just too long, above 100 characters.
- Depth of nested for too high: We allowed the use of a depth of 2 but one volunteer exceed that limit. For this exercise, using a higher depth meant that the code was not optimized.
- Overcomplicated Boolean expression: One of the students used a Boolean expression with more than 3 operators. Since only one of them got that error, it was possible to implement the class with less than 3 operators.
- Too many methods: One participant has more than 20 methods in his class.
- Using stars in imports: One volunteer use stars in his imports. While it is not really a big mistake, we believe it is better to be as precise as possible when we import libraries so that we do not import more classes than needed.

### 6.3.5 Conclusion

Those results were interesting because we chose detectors that should help novices but more experienced students still made the corresponding errors. It proves the fact that we tend to keep bad habits when we write code. One of the reason that was already noticed is that the learning process of programming language is often project-based and for all the projects we have to do, there were only functional checks. It is then normal that we do not pay any attention to the style since there is no check.

After the exercises were done, we contacted the participants to get their opinions about that experiment. They all agree to say that using a tool such a Checkstyle to look at the style is really interesting in general and improves the readability. However, some of them also think that it is important to choose the right checks as they felt that it is case-dependant but also that teachers should not promote a style that is too restrictive. It is clear that the goal of using such checks is to help students write clean code but not to make them all have the same implementation.

We also learn from the teacher methodology that as it might seems easy to make exercise, it requires some time to make the good choice and to design the exercise correctly.

### 6.3.6 Threats to validity

We wanted to make an exercise that would be resolved by beginners because they are our target audience but since none were available or hard to reach in the holiday period, we used volunteer that are more experienced. Even if we asked them to not optimize and simulate novices, we cannot say if they managed to do so. Furthermore, those volunteers were friends that agreed to help us on so even if their feedback was positive, it might be biased.

## 6.4 Conclusion of the experiments

In the end we learned a lot from the three experiments. Integrating Checkstyle to Pythia did not seem difficult thanks to the architecture of the platform and the fact that we can create distinct environments on it. Write custom check is not hard if we understand how Checkstyle AST are built.

Using Checkstyle correctly was more complex than we expected because we needed to chose the detectors that suited our needs. If we have too many detectors, the tool might seem too restrictive as we force a unique style and the output can become hard to read if too long. If not enough detectors are used, we are not restrictive enough and can not check a lot of patterns.

After describing the teacher's point of view, we learn that the process of making exercise is not that easy and so if teachers get the opportunity to use a tool such as Checkstyle, it would require some time to know what can be check using that tool. However, having coding style checks would provide teachers more possibility for exercises and so more diversity.

For our work, we would make other experiments if we got the opportunity to do so. It would have been really interesting to use Checkstyle on the same set of students more than one time so that we could have seen some improvements. With that being said, we believe that there would be improvements in students code if they use the tool as proved by the experiment made in [2]. They used Checkstyle for two projects, taught coding style conventions in between and noticed that the number of errors in the second project decreased.

# Chapter 7

# Conclusion

The main goal of this master thesis was to provide an introduction to structural coding checks for Java and to see how we could use them in order to extend the Pythia platform. The utility of such checks in our case is to help students to adopt a good coding style. It means that those checks are used to promote clean code. Why is it important? Because coding properly brings several advantages. For example, a well-written program is more readable as appropriate names for variables and methods are used or the indentation is respected. A code that respects some coding conventions is also more practical to use when several people are working on the same project. From a teaching point of view, it is of course more convenient to correct and to understand a student's program that is clean. It can save a lot of time in some situations. Promoting a good coding style during a programming language learning phase is really important because it could prevent students from developing bad habits. For that particular reason, it could be interesting to do so through the Pythia platform that is now used by our students and teachers.

## 7.1 Summary of this work

In this work, we first take a look at three Java tools to understand what exactly is a coding style check and how it is implemented by those tools. We learned that they use detectors that look for particular patterns in the code. Tools manage to do so by checking either the bytecode or by transforming the code into an AST. The bytecode approach used by Findbugs turned out to be not convenient to work with at all, as we tried to do so. PMD was a good alternative to Checkstyle but a bit less documented and therefore less practical. That is why we prefered to use Checkstyle to implement some custom detectors. We introduced Pythia by going through all of its features and using the web interface. In section 2.2, we also described briefly its architecture.

We looked at other online learning platforms so that we can compare Pythia to them and try to identify some similarities or differences. We found that all of those platforms are pretty similar in terms of what they provide to their users. It was interesting to notice that none of them seems to provide coding style checks which makes it a good addition since Pythia would be more complete compared to them.

We briefly defined functional checks and coding checks and we also tried to compare the different approaches used by the tools we looked at. After that, we discussed the implementation we made for this thesis which groups both the implementation of custom detectors using Checkstyle but also the integration process to Pythia.

As a validation part, we decided to create a first exercise to simulate how it would be to have coding checks in Pythia. We described our methodology then we explained more concretely what we did. We asked students to help us by trying to resolve the exercise and interact with the tool that was appropriately customised for our needs. To collect students' opinions on our work, we gave them a quick survey. Both the exercise and the answers we received were very interesting since there are a lot of things we took and learned from. We also decided to try Checkstyle ourselves by resolving an exercise and using the tool with the default configuration file. The most important thing we took from it is the importance of writing a good configuration file and therefore using a set of detectors that fits the exercise. We forgot to ask for teachers' opinions because we were too focused on students and failed to realise that in time. As a third experiment, we tried to describe the teacher's methodology for making exercises and were able to try Checkstyle on a bigger exercise. The volunteers were more experienced developers than the novices from the first experiment but we saw that they still made basic mistakes.

In the end, we are convinced that adding coding checks on a learning platform such as Pythia would definitely be useful. However, to take the maximum of such check, it will require to identify beforehand what we need to look for and implement corresponding detectors if they do not already exist.

## 7.2 Limitations and potential solutions

Through all this work, there were limits that arose and that should be considered. As we just mentioned, using coding checks requires to know what patterns to search for in students' code. Another limit is that there are some instructions that we might want to look for but that are really complicated to detect. The reason is that there is always more than one way to write a set of instructions that could perform the same action. For example, in the exercise we wanted to make sure that students would check the input board size. At first we were thinking of writing a detector that would look for an "*if*" instruction that had "*board.length*" in its condition. However if students would decide to store the size in a variable and then use that variable in the "*if*", the checks would not work. It is a simple example but we can easily imagine that there are lots of other cases where it would be difficult to implement a good detector. The fact that coding checks are not very relevant when used alone but should instead be used in addition of functional check should also be kept in mind. We saw in our exercise that coding checks alone could give students the feeling that they are doing it right if no error is detected while in fact their code does not perform as expected. A last problem which is linked to the first one is the importance of writing a good configuration file. Writing a configuration file is not hard but writing a good one that is appropriate and fits our needs is more complicated.

Fortunately, there are solutions to all those limits. The need of knowing beforehand what to look for could be satisfied through iteration. Pythia has been developed to stay more than

one year. That means that for an exercise in particular, we can have some feeling about what to look for and write corresponding detectors. Once the exercise is done, if there are still errors that we do not think of, we can improve the coding check by writing more detectors so that the more the exercise is resolved, the more we will meet new errors and the more our check will be improved and complete. The limitation that comes from the fact that there are many ways to implement a program for a specific task could be resolved by reducing the freedom of the students (if it is possible). As shown in our exercise, the less freedom students have in their code, the more it is easier to predict what they would write as instructions. For the configuration file, there are two ways to deal with it. The first one is using iteration. We start with a configuration file that is not that precise and the more it is used, the more we see what is good or bad so that we can improve it. At the beginning, it requires some time and some feeling about what to check. The second way to handle the configuration file problem is to proceed as we did in our additional experiment. It consists of using all the detectors on the first check and then see which ones we can remove from the set because they are not appropriate.

## 7.3 Future work

As we can see in the result we received for the validation part, there is still work to be done before we could extend Pythia with coding checks.

Even if we received a good percentage of positive student opinions about our work, we cannot affirm that it is representative enough because only ten students have participated. As a next step in the validation part, we wanted to make an experiment similar to the one described in [2]. It would have definitely be interesting to ask to the same set of students to resolve multiple exercises with coding checks and see if they improve themselves from the feedback provided by Checkstyle. For example, we could have split students in two groups and let them do two exercises but not in the same order and with coding checks for one of those exercises. That way, we would have seen if students directly apply what they would learn by comparing the code from the two groups.

With more exercises, we also could have simulated a small course with a plan for several weeks. Each week, we would provide an exercise that focuses on one basic error. For some weeks, we could also checked the error from a previous week as well to see if the students learn from their mistakes. At the end, we would have covered a large set of basic errors and could see the evolution through the weeks.

Another point to work on before going further is to evaluate if teachers would be interested to use coding style check. Because we think it is interesting, we made the assumption that it is also the case for teachers but we cannot be sure until we ask them. Are they convinced that a tool such as Checkstyle could be beneficial to the students? We can think that it also depends on courses.

Besides the utility of that check, there is a need of a Pythia developer to be able to work with the tool to customise it accordingly with the needs but also to implement detectors. The Pythia crew takes care of the platform, they are those that implement all the checks and the web interface. They are the ones that would need to handle the coding style check. It might seems hard but Checkstyle already includes a lot of detectors that would be sufficient

in most of the cases. Developing a custom detector mainly requires that we understand how Checkstyle AST are built but as we already mentioned, the tool is well documented in its website. In general, if a teacher wants to prevent a pattern in students' code, he would have a clear idea of what should be checked and creating the corresponding detector should not be that hard.

If coding checks were used in Pythia, one of the main problems is that the Pythia developers act as a middleman. If a teacher wants to make use of such checks, he needs to be aware of what Checkstyle can do but since he doesn't have to understand how exactly the tool work, it is a bit tricky. However, we think that with the experience that teachers have and all the flaws they have already seen while correcting students' code, they should know what to focus on and then discuss with Pythia developers if what they want is feasible.

Our thesis focused on coding checks for Java. However, as Pythia is a platform that handles other programming languages, similar work could be done for them.

In this work, we have considered coding style checks but there exists other kind of checks. Also, we focused on individual checks but we could imagine group checks that compares students' codes. For example, when we want to check the performance by using execution time, we usually decide a time limit that tells whether or not a code is optimized enough. With a group check, we could compares execution time of students' implementation between them. Students could feel motivated to optimize their code and try to have the best time. In general, checks are made to notify students when they do something wrong but we could implement checks to reward good code. One way to do so is to compute a score from students code; good code would receive higher score. If we made those scores visible to all students, there could be a kind of competition that again would motivate students to produce good code.

## 7.4 Final words

In the end, we learned a lot from the introduction of the tools we chose to the experiment we made. We are sure that coding style checks could be useful to Pythia but as we have just said, there is still work to do. We do not think that this thesis is sufficient to say if such check have to be added to the platform but we can see our work as an opening to it. We gave a kind of introduction and try to stimulate the interest on it. Besides that, we became more familiar with Pythia since we worked for it and even if it is still considered in a beta phase, we can feel the potential it has and we believe it will be more and more important in the future. For that reason, working on a possible addition to it is really pleasant. To conclude, we hope this work will promote the idea of coding properly and its importance. Writing code is something that all students can do but doing it cleanly is sometimes harder during the learning phase. That is why we should try to educate them that way while teaching them how to program. Students are the ones who will implement the programs of the future. That is the reason why it is so important to teach them the good habits as soon as we can. We need to assist them as much as we could because later, it is their programs that will assist us in our daily tasks.

# Chapter 8

# Bibliography


[1] S. M. Hiroaki Hashiura and S. Komiya, "A tool for diagnosing the quality of java program and a method for its effective utilization in education,"

[2] Y. H. Lim, "Tool support for learning programming style," 2009.

[3] "Findbugs." url = http://findbugs.sourceforge.net/, Jan. 2014.

[4] D. Hovemeyer and W. Pugh, "Finding bugs is easy," in *Companion to the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '04, (New York, NY, USA), pp. 132–136, ACM, 2004.

[5] "Findbugs example of detector looking for unguarded logging." url = https://www.ibm.com/developerworks/library/j-findbug2/, Jan. 2014.

[6] "Checkstyle." url = http://checkstyle.sourceforge.net/, Jan. 2014.

[7] M. P. Robillard, R. J. Walker, and T. Zimmermann, "Recommendation systems for software engineering," *IEEE Software*, vol. 27, pp. 80–86, July 2010.

[8] A. Lozano, G. Arévalo, and K. Mens, "Co-Occurring Code Critics," Dipartimento di Informatica Università degli Studi dell'Aquila, L'Aquila, Italy, July 2014. Extended Abstract.

[9] "Pmd." url = http://pmd.sourceforge.net/pmd-5.1.0/, Jan. 2014.

[10] Y. Arakawa and M. Arai, "Eclipse plugin tool for learning programming style of java," vol. 2, 2013.

[11] N. Truong, P. Roe, and P. Bancroft, "Static analysis of students' java programs," in *Proceedings of the Sixth Australasian Conference on Computing Education - Volume 30*, ACE '04, (Darlinghurst, Australia, Australia), pp. 317–325, Australian Computer Society, Inc., 2004.

[12] S. Combéfis and V. le Clément de Saint-Marcq, "Teaching programming and algorithm design with pythia, a web-based learning platform," *Olympiads in Informatics*, vol. 6, pp. 31–43, 2012.

[13] "Project euler." url = http://projecteuler.net/about, Apr. 2014.

[14] "Rubymonk, interactive ruby tutorials." url = https://rubymonk.com/, Apr. 2014.

[15] "Try python, interactive python tutorials." url = https://rubymonk.com/, Apr. 2014.

[16] "Code school, learn by doing." url = https://www.codeschool.com/, Apr. 2014.

[17] K. Ala-mutka, T. Uimonen, and H. matti Järvinen, "Supporting students in c++ programming courses with automatic program style assessment," *Journal of Information Technology Education*, vol. 3, pp. 245–262, 2004.

[18] "Home and learn." url = http://www.homeandlearn.co.uk, June 2014.