INSTITUTO
UNIVERSITÁRIO
DE LISBOA

Avaliador Pedagógico da Qualidade de Código

Francisco Miguel da Anunciação Alfredo

Mestrado em Engenharia de Telecomunicações e Informática

Orientador:

Doutor André Leal Santos, Professor Auxiliar,

ISCTE - IUL

Co-Orientador:

Doutor Nuno Miguel de Figueiredo Garrido, Professor Auxiliar,

ISCTE - IUL

Outubro, 2020

TECNOLOGIAS
E ARQUITETURA

Avaliador Pedagógico da Qualidade de Código

Francisco Miguel da Anunciação Alfredo

Mestrado em Engenharia de Telecomunicações e Informática

Orientador:

Doutor André Leal Santos, Professor Auxiliar,

ISCTE - IUL

Co-Orientador:

Doutor Nuno Miguel de Figueiredo Garrido, Professor Auxiliar,

ISCTE - IUL

Outubro, 2020

Acknowledgments

First, I would like to thank my family, and girlfriend, for all the unconditional support and for always encouraging me to push a little bit further when it came to my academic life, without that, I am not sure if I would ever reach this point, specially while working and studying at the same time.

A big thank you for all the friends and colleagues that offered their time to help me test and improve the tool.

A huge thank you to both my advisers, André Santos and Nuno Garrido. Thank you, Professor André, for all the support regarding all the aspects that were involved in the development of our tool, and thesis. For always being available, and willing, to help me when I found some roadblocks or was facing some issues with the code, I really appreciated it. Thank you, Professor Nuno, for all the advices and really constructive criticism that you gave me during our weekly conversations, it really made and impact on the way that I was thinking and really helped me improve the tool and this thesis.

I would also like to thank Professor Alcides Fonseca, who was a member of the board during the presentation of my thesis and which gave me a very constructive feedback regarding some aspects of my work, that helped me improve it.

To everyone else that was involved in any way and that helped me during this process, thank you.

Abstract

When it comes to learning how to program, during introductory programming courses, students tend to form misconceptions regarding certain code aspects, that they aren't aware of. These misconceptions can result in code quality issues, which don't affect a program execution, but are considered bad practices and should be avoided. Most often, they will stay unaware of such issues until an intervention is made, by a third-party agent, such as a professor. There are already a few tools that focus on this type of issues, by highlighting the code quality issues and supplying a quick fix for them. This approach, while effective, isn't ideal for beginners, since providing a solution without explaining the problem, won't make the student understand why the code that was written is considered a bad practice.

With this being said, a pedagogical tool was developed. This tool uses static analysis in order to apply some code quality issues detectors to the student's codebase, and focus on highlighting the issues, while explaining why they are considered to be wrong and therefore, why they should be avoided.

After the development of the tool was done, a study where a batch of projects from the previous year's course were subjected to the tool, in order to find out how often can these code quality issues be found in the student's codebase, and which code issues are the most popular amongst unexperienced programming beginners.

Keywords: Introductory programming, code quality issues, pedagogical tool.

Resumo

Durante a aprendizagem de programação, os alunos têm tendência a formar ideias erradas sobre os papeis de certas estruturas de código, envolvidas nos programas que os mesmos escrevem. Embora estas estruturas não contribuam para o mau funcionamento dos programas que são escritos, contêm muitas vezes alguns problemas de qualidade de código, considerados más práticas no mundo da programação. Devido à inexperiência por parte dos alunos, estes nem se apercebem que adotaram tais más práticas, tornando-se necessária a intervenção de um terceiro agente, como um professor, para que as mesmas sejam combatidas. Existem algumas ferramentas que abordam este problema, sendo estas focadas na vertente profissional, apenas identificando e oferecendo uma resolução rápida para estes problemas, sem tentar fazer com que o sujeito perceba o porquê de ter cometido tal erro, sendo que esta solução não é ideal para os principiantes da programação.

Posto isto, foi desenvolvida uma ferramenta pedagógica, que usa análise estática de maneira a aplicar alguns detetores de problemas de qualidade de código, no código que é escrito pelos alunos, de maneira a identificar e alertar os principiantes para estes problemas, fornecendo uma explicação pedagógica sobre o porquê de tal má prática ser considerada prejudicial à aprendizagem.

Após a ferramenta ter sido desenvolvida, realizou-se um estudo onde um conjunto de projetos de semestres passados foram recolhidos e processados pela ferramenta, de maneira a descobrir o quão frequente é possível encontrar estes problemas no código escrito pelos alunos, e que tipos de erros são mais comuns entre os alunos.

Palavras-chave: Introdução à programação, problemas de qualidade de código, ferramenta pedagógica.

Contents

List of figures

List of tables

1. Introduction

1.1 Motivation

When it comes to learning how to program, there are a lot of challenges that one can and most certainly will face during the beginner phases. Most of these problems are due to the lack of knowledge about the fundamentals and lack of problem-solving skills, and it can become difficult and frustrating to face them, especially while being on an early stage of the learning process. On the contrary, beginners are also capable of solving problems that they are subjected to but, without proper orientation from a more experienced programmer, such as a teacher, they may use solutions that while allowing them to achieve the correct result, may sometimes not be the best ones, both syntactically and semantically speaking, which may result in poor code quality.

Since worrying about the behaviour of the program and how to implement it can already be a very demanding task for a beginner, subjects like code styling and good code practises regarding code quality, can often be left behind by the student. This can be an issue because a working program does not emit output when it comes to the quality of the source code, which leads to the beginner moving forward, which may lead the beginner to acquire bad coding practices right from a very early stage, practices that tend to get even worse over time and that should be fought right from the beginning.

There have been studies trying to figure out why these quality issues tend to remain unsolved. Most of them came to the conclusion that students are not concerned with topics such as maintainability, performance, and testability, and with this being, they tend to be satisfied once they reach the desired result [1].

A way of facing this situation is to seek counsel from more experienced individuals when it comes to programming, or to do some research on the matter. But most of the time, this can end up being very time consuming, and it's not the most practical thing to do. In a lot of cases, it might not even help the beginner to reach a conclusion and improve its coding/thinking skills.

While there are lots of tools whose purpose is to search for these kind of issues and to aid developers in their work, most of them don't inform the programmer on why the issues that are found are considered to have problems within them. Without the proper feedback about the written code, beginners that may have learned bad practices, most certainly won't avoid using them without the proper guidance, like mentioned before.

1.2 Approach

After researching about what are the causes of the general bad code quality found among programming beginner's projects, the proposed solution to fight this issue is a pedagogical tool that could help beginners improve their programming knowledge and skills, by raising awareness to code that exhibits quality issues after the execution of the program is complete. This tool would start by parsing the source code written by a beginner, analyzing the parsed code and then recognize if there are blocks or expressions, that should concern the beginner about the quality of the code, and if so, warn the beginner about such concerns, the reason why it is considered an issue, and a possible solution/explanation so that the beginner learns from it and avoids doing such mistakes in future developments.

This tool will detect problems such as empty code blocks that are left in the code, problems with the assignment of values to variables, problems with conditions such has if blocks and loop blocks guards. It belongs to the domain of the tools/plugins, that are installed and used in an Integrated Development Environment (IDE), with the purpose of supplying an individual with a set of devices, which have the main objective of promoting good code quality, by discouraging the usage of bad practices and suggesting better ones. This will eventually lead to a better structured and bug-free code base.

1.3 Research Questions

The research questions for this thesis are:

1. Which are the most common mistakes and bad practices performed by programming beginners?
2. What are the subjects that beginners tend to find most difficulties?
3. What is the best way to raise awareness regarding bad code practices, in the first stages of the learning process?

1.4 Objectives

The main objectives of this thesis consist of:

1. Understanding which are the most frequent bad practices and misconceptions regarding the learning process of basic programming structures, that are shared by the greater percentage of beginners that are new to programming.

2. Developing a strategy, by developing a pedagogical tool, in order to address these issues, so that beginners could overtake the obstacles on the way and write better code, in the most autonomous way possible.

1.5 Research Method

The research in this dissertation follows the Design Science Research methodology. This meaning that it will be composed of six different phases. These six phases are:

1. Identification of the problem, where the research problem is defined, and the solution justified. In this case, the problem is the lack of good solutions regarding code quality, in beginner's programming project, and the solution is a pedagogical tool that shall help in such matters.
2. Definition of objectives for the solution. The objective of the proposed tool is to help beginners to be aware of the mistakes and bad practices that are used, regarding code quality.
3. Design and development of the methodology to build such a tool. It shall process the beginner source code and match it with a library that contains the mistakes and bad practices that are being targeted, so that the origins of the issue can be explained to the beginner.
4. Demonstration by giving the tool to a beginner, so that it can be tested.
5. Evaluation by comparing the main objectives to the results obtained in the previous phase.
6. Communication of the problem, the solution, it's utility and effectiveness, via this dissertation, by comparing it to previous tools that target similar aspects in code development.

When the fifth phase is reached, if there is something wrong with the developed solution, steps 2, 3, 4, and 5 and repeat until everything is as expected. To conclude, this method tries to find user insights on the problem first, and then focuses on finding a solution to such problem, by first researching what has been done that can be taken into consideration, and then by following the steps described above in order to achieve the best solution possible [2].

2. Related Work

2.1 Difficulties with learning how to program

Engaging in the learning of a new skill, such as programming, can be a challenging topic to an individual that never had any type of contact with the fundamentals of this subject, and even less with the thinking process that goes behind it. One may face several difficulties and roadblocks when it comes to learning how to program, whether being code quality related issues, like syntactic errors or the bad usage of code structures, semantic issues that can manifest themselves as the absence of being able to think about a problem in an abstract way or overthinking a solution for a problem that required a much simpler one.

Most of the times, beginners can't acknowledge that these issues are present in their minds, resulting in a poor understanding of basic programming concepts and how or when to use them. This will eventually lead to the development of misconceptions that are much more difficult to rectify further down the line of the learning process [3].

Before starting to analyse which of these difficulties are shared among most beginners, a better explanation will be given, regarding the meaning and definition of a misconception. When it comes to learning, a misconception consists on a set of poorly formed ideas that are based on a thinking process that isn't completely right. This means that one may have a formed set of ideas or conclusions regarding a subject, that when combined, do not lead to a well-formed and correct knowledge, or opinion [4]. Regarding the process of learning how to program, a misconception can be better defined as the incapability of fully understanding some of the programming fundamentals, their practical use cases, and the meaning of it. Misconceptions can manifest themselves on every stage of programming learning process, whether it being on the stage where the beginner starts to learn about a language syntactic features, the learning of general concepts within the domain of programming, or the usage of strategic knowledge obtained from previous stages, when it comes to finding a solution to a problem, using programming as the mean to achieve the solution.

The first enumerated stage, learning a language syntax, requires that one must understand and be able to apply the specified set of rules while writing a piece of code, like opening a set of curly brackets when defining a new method. The second stage, regarding the learning of programming concepts, takes place when the beginner has learned the topics within the first stage and knows how to apply them while being aware of what the written code actually stands for. A good example of this, is when a beginner writes a method that includes a loop, while knowing that the loop will execute the code inside of it until the guard condition no longer holds, and this being the reason why the loop was

implemented. Another example is the difficulty on learning the abstract concepts behind an object oriented programming language, such as Java, and the relations that exist among the elements and concepts [5]. The last stage consists on using what was learned from the previous two, to attack or resolve an existing problem [4].

The misconception is formed when, in one of the two first stages, or in both, wrong ideas/conclusions are formed, about what is being learned, whether it being the poor understating of the syntax of a programming language, or of the wrong/disturbed conceptualization of what will happen when a piece of code is running. Having misconceptions regarding these two stages, can then affect how one might think on the solution to a problem. With this being, it is justifiable that beginners should receive help so that they can prevent the acquirement of such misconceptions or fix the existing ones. A more detailed enumeration and analysis of found difficulties and misconceptions, will be done hereinafter.

Mistake	Category	Frequency	Time-to-fix (sec)
Unbalanced parentheses, curly or square brackets	Syntax	1 861 627	17
Non void method without return statement	Semantic	817 140	38
Confusing '=' with '=='	Syntax	405 748	113
Nonvoid method return value is ignored	Semantic	274 963	1000
Invoking nonstatic method as if it were static	Semantic	202 017	48
Class implements interface but not all the required methods	Semantic	186 643	107
Including types of parameters in method invocation	Syntax	117 295	23
Incorrect usage of the semicolon after conditions and loops	Syntax	108 717	387

Table 2.1. Frequency and Median Time-to-Fix for the most common mistakes found in a 2-year dataset [5].

Some of the difficulties/errors that were found the most in previous studies, are displayed in the table above, and syntactically wise, the most frequent ones are the mismatched/missing parentheses when surrounding a condition, the mismatched/missing brackets when declaring a new method , the lack/misposition of quotation marks like semi-colons after a statement or a condition ending, followed by the confusion between the comparator '==', with the assignment symbol '=' , writing quantity comparators in the wrong order, like '=>' or '=<' instead of '>=' and '<=', and so on [3]. The occurrence of this mistakes may be justified by the lack of practice in the early stages of the process of learning a new programming language, such as Java, or to the hard time that some beginners face while learning

the basics of the language syntax. On the other hand, they tend to be the first ones to be identified, due to their relative recognition simplicity, and the fastest ones to be fixed, like seen in the table above, where the syntactic mistakes tend to take much less time to fix, in comparison to the semantic ones.

Regarding the semantics behind a piece of code, it can be said that this is where beginners tend to struggle a lot more. This kind of mistake tends to appear on the second stage of the learning process, where the beginner starts trying to understand/make sense of what the code that is written represents and what it will do while it is running. The lack of experience and knowledge, both syntactic and conceptual, often leads the beginner to start writing code without thinking in a proper and well-formed solution that should combine the thinking process behind solving a problem, with the semantics and syntactic features that a language may provide to write an algorithm that will serve as solution to the referred problem.

There is a very usual situation, where a beginner gets stuck in a position where the lack of coding knowledge, agility or problem-solving skills are the preventing causes behind the inability to obtaining the solution to a certain problem. Most of the times, the beginner seeks help from available literature, or more experienced individuals, and with this kind of help, normally, the beginner can surpass the difficulties that were found, while learning from them, and getting closer to obtain a solution to the problem. There are also times when the beginner is able to think about a solution, implement it, and reach the desired result. This can be a sign that the beginner is starting to learn and evolve when it comes to using programming as a mean to resolve problems, but it is very common to see that, while the correct result has been reached, the way that it was done isn't always as good as it could or should be, syntactic or semantic wise. When this happens, the beginner tends to think that the implementation of the algorithm that was written is correct, and in most cases, lacks the criticism skills to question whether it could have been done better, or not. This type of situation helps feeding the misconceptions that one may possess and will most likely result in the usage of bad thinking and coding practices, in future developments.

2.2 Software quality in the beginner phase of the learning process

Now that some of the primary difficulties have been discussed, the next step is to figure out what is the best way to combat the misconceptions that may be formed in a beginner's mind. There are studies that have tried to answer this very question, but most of them tend to end up in the same conclusion, which is that it is still unclear whether it is a good idea to promote software quality right from the beginning of the learning process, or if it is better to wait until a certain amount of coding knowledge and maturity has been reached by them [6]. The issue with doing it when beginners have

little to no knowledge, is that they already have so much to focus on, such as understanding the basic structures and concepts and learning how to use them, that inflicting that kind of concerns may result in bigger confusions and doubts on a beginner's mind. Contrarily, if beginners are left alone, without addressing their code quality issues, bigger undisciplined coding practices may resolve from such issues, which will most certainly contribute to the formation of syntactic and semantic misconceptions.

A study was conducted where multiple beginners code samples were searched for code smells, which are known coding patterns which are indicative of implementation and code quality problems [6], and analysed in order to find if students would lose bad coding habits acquired from the early stages of learning how to program, as their knowledge and coding experience progressed. Beginners in all levels of programming proficiency tend to retain bad coding quality issues present in their coding developments, independently of the level of coding knowledge and expertise that they were in. It was also showed that being exposed to some kind of software quality concepts, while in context with the target issue that has been found, would result in the acquirement of awareness, regarding the issue, and practical skills which allowed the beginners to proficiently developed better solutions in their future developments while still learning about better software design and implementation practices [6].

On another performed study of commonly occurring quality issues in beginner Java code, there was found that students, even when subjected to code quality analysis tools, tend to hardly, or never, resolve the quality related issues [3]. These support the theory that an increase in programming proficiency and autonomy, does not always lead into proper programming practices, since in most cases, the bad ones tend to persist. These facts support the need to have a proper way of correcting beginners. When these quality issues occur, there should be an educational intervention with is the ability to explain how certain practices are improper, or simply wrong, have the beginner understand why this is and what must be done and learned so that it is fixed and does not happen in future developments.

This can be obtained, as stated before, with the help of a more experienced individual when it comes to programming, like a professor. The issue with this is that it doesn't promote the beginner's autonomy. There are tools that are developed with the intent of giving warnings regarding bad coding practices, and how to fix them, which will be discussed in the next section.

2.3 Code quality inspection

One way to better detect code quality related issues, is by performing an inspection on the source code present in solutions implemented by a beginner. It can be used for improving the software quality as it involves the examination of the code, which can lead to the recognition of problematic blocks that may result from misconceptions and, consequently, be associated with bad code quality practices [7].

The recognition of these patterns is done by comparing the different blocks of code that are present in the parsed source code, with code smells, which are patterns that are generally associated with the implementation of bad practices [7]. These serve mainly as indicators of sections that should be targeted for refactoring, are can be of great use for the development of the proposed pedagogical tool, in the way that it allows the finding of the bad practices done by beginners, which will then be targeted in a future explanation, in order to help the student to recognize and solve them.

The main code smells that will be targeted in order to promote the better quality of code written by students, are the ones that are most commonly found in introductory courses. These are:

1. Empty code structures. These refer to blocks of code such as empty selections (if statement).
2. Code duplication. The same block of code that are found in multiple methods or classes.
3. Long methods. When written methods tend to become too long to understand at first glance.
4. Unnecessary verifications. When beginners implement too much verifications regarding variables values, for example, that could simply be removed or replaced by simpler statements.

It is important to emphasize that the code smells that will be used to detect the bad practices present in beginners code may not only be enumerated ones above [7]. These will serve has a starting point and if the objective is achieved, after the testing phase, other ones will be included.

2.4 Existing pedagogical tools

Many teachers found that teaching programming languages such as object-oriented ones, like Java, is more difficult than teaching a traditional procedural programming language, due to the pedagogy behind object-oriented programs, and the requirement that it has regarding abstract thinking. Beginners must be able to model the physical imagined items, into virtual ones, and define the complex interactions and relationships that are present between each other, and this is not easy to

perform while there is a lack of programming knowledge and experience, that is the case for most of the beginners [8].

There is a lack of appropriate pedagogical tools that can fully, or almost fully, combat the programming paradigms with its own abstract ways. There has still been some development and progress regarding the development of tools that have the purpose of reducing the load of abstraction that one must present in order to better learn and understand the topics enumerated above. Some of these tools will be enumerated hereinafter.

2.4.1 Scratch, Blockly and Greenfoot

There are tools which main focus is to help the beginner better understand the fundamentals of programming, and how to write code with them. These tools are separated in two categories which are the graphical, and the textual ones.

Regarding the graphic domain, many visualization tools, such as Scratch [9], Blockly [10], and Greenfoot [11], have been developed with the purpose of helping the beginner better learn and understand the language in question.

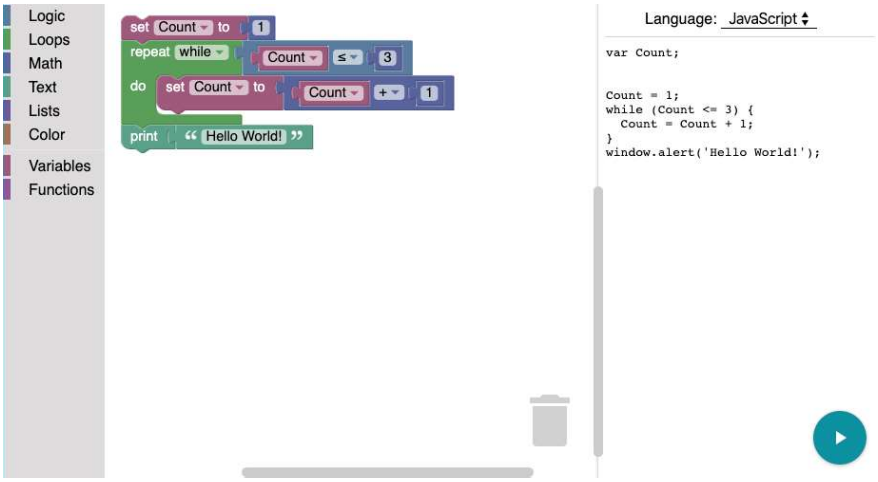


Figure 2.1. Blockly editor view [2].

The first two examples, lay on a block-based solution, that is fully graphical, and enables the beginner to visualize the basics of structured programming in a more and concepts as 2d graphics and create relations between them, without the need to have the required syntactic and semantic knowledge, by simply dragging them next to each other, creating a sequence of events [5]. This approach removes some of the load when it comes to understanding the abstraction behind these concepts, when using a text-based programming environment, and can serve as a good starting point to someone who just started learning how to program, even though it has been developed with a younger audience in mind.

Regarding Greenfoot, it helps teaching object orientation, in java, and offers visualization and interaction tools that are built into the coding environment [11]. It mainly focuses contextualization of object-oriented programming, by allowing the creation of actors that live in worlds, which represent objects that are used to build games, simulations and other graphical programs. These are all programmed using Java textual code, and the platform provides a combination of programming experience, while using the traditional text-based language with visual execution.

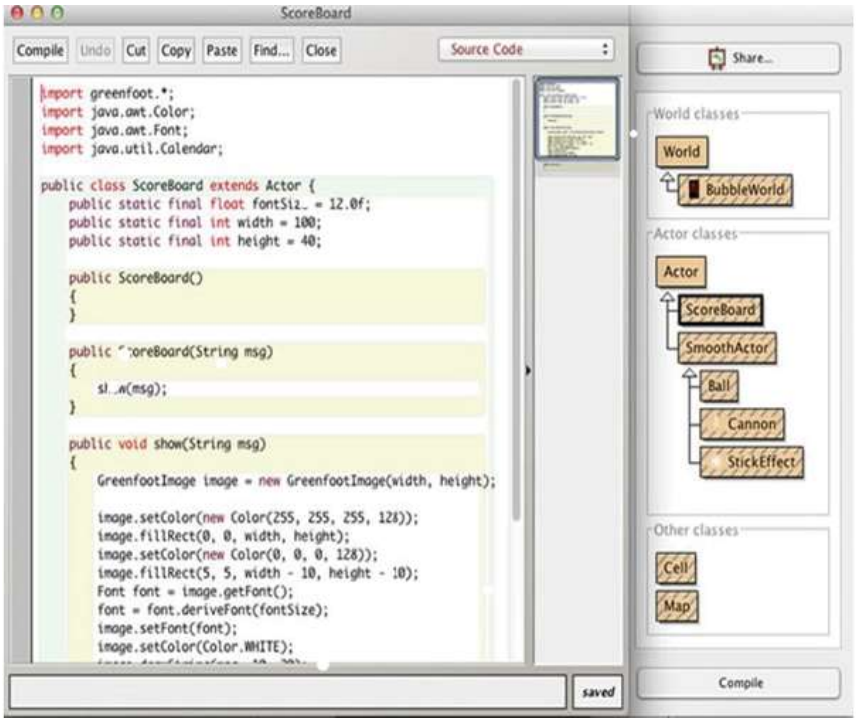


Figure 2.2. Greenfoot development environment view [11].

Greenfoot purpose is to help teachers teaching what most beginners find the hardest, which is the abstraction that is behind the main concepts of object oriented programming, such as the class/object relationship, the methods inside them, the parameters that a method shall receive, and why, and the objects interaction. All of this is better explained with the visualization provided by the Greenfoot's integrated development environment (IDE) [11].

2.4.2 VLT-OOP

Another designed and developed tool that aims to help better understand the main concepts of object-oriented programming is VLT-OOP. It does this by allowing both the development and visualization of the interactions and relations present between structures, to be done through a graphical 2D application [5]. In VLT-OOP, a Class denotes an abstract definition of an Object, so that it can implemented and visualized. Each of these objects possesses its customizable attributes, actions and rules, which can be added and defined using the user interface. They can also be created based on parent objects, and use their parent's attributes, actions and rules, to create more detailed and complex ones. This procedure corresponds to the Inheritance concept. Polymorphism can also be found, by creating rules with associated actions that can be used to specify and define different and more complicated methods. It shares the same goal with the previous tools, while also representing, in a deeper way, what the actual structures and concepts of a programming language are [5].

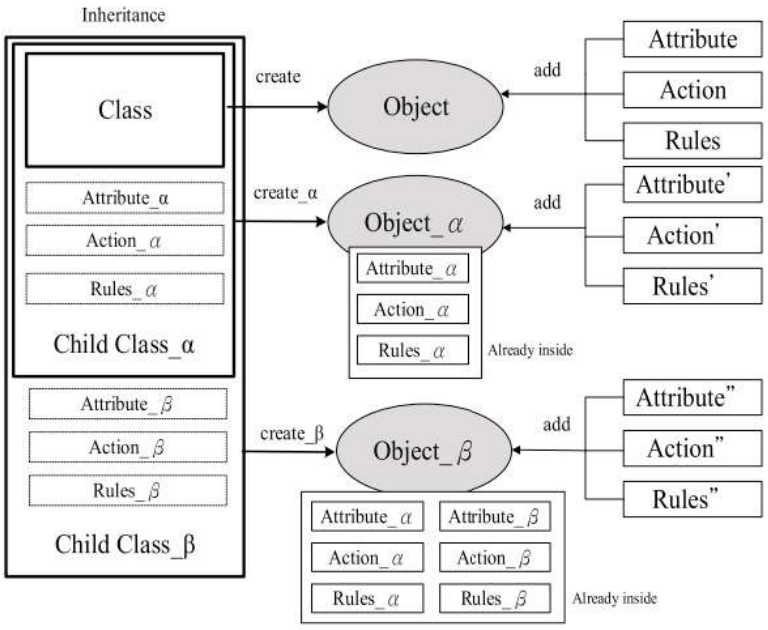


Figure 2.3. The OOP-based Data Model in VLT-OOP tool [5].

2.4.3 BlueJ Environment

BlueJ is a programming environment that is broadly used for teaching object-oriented languages, such as Java, on an introductory level, using an objects-first approach. It deeply relies on a Modelling Unified Language (UML) based graphic interface that supplies beginners with an instant view of a project's class diagrams. It has the purpose of allowing the beginner to analyse and learn about the attributes, methods or API's, that a certain class contains and provides.

The user can interact directly with classes and objects in order to observe the outcome of this interactions, such as method invocation. This can be greatly advantageous due to it allowing the

beginners to define, manipulate and observe the outcome of the cognitive structures or "mental models" that are envisioned by them [12]. This feature serves has a great advantage when it comes to looking to a piece of code in an abstract way, but it has also been found that the transition from manipulating the programming structures via graphic interface, to doing it by writing textual Java code, can represent a challenge in most cases.

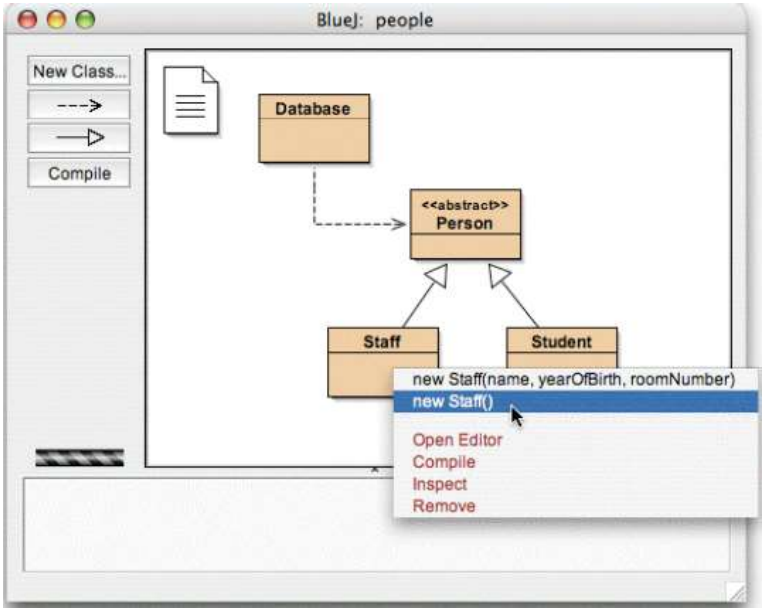


Figure 2.4. BlueJ Interface while manipulation and creating objects [13].

For a beginner programmer, it can be a difficult task to write the first lines of code, which means that the transition should always be followed closely, in order to maximize the changes of the beginner succeeding in writing new code [12].

In general, it is an easy to use and helpful tool that allows beginners to be visualize and better understand how object-oriented programming structures function, but it is also recognized by some educators, that it is not a mandatory tool and that some of its approaches are not the best ones, such as using object-orientation as an topic for programming introduction, or the poor feedback to quality issues [14][15][16].

BlueJ also does not provide any type of recognition regarding code patterns that indicate code quality issues.

2.4.4 BlueJ-UML extension tool

It implements attribute, method and constructor information in a UML styled class diagram. It represents the original BlueJ UML feature, but with a stronger association, aggregation and composition dependency, while strengthening the interactions between beginner and BlueJ in a way that it allows the manipulation on class diagram through the UML notations to the original source code.

Since beginners, when lacking basic programming knowledge, tend to struggle with the notations that are present in UML, this extension aims to provide a highly interactive platform, where learning these annotations through practise is incentivized. It also tries to fight against another difficulty, which is the connection between UML and the source code that forms them. This happens because many tools tend to omit the connection between these two aspects, due the complexity of it. BlueJ does in fact supply a connection between these two aspects, while being weak one, and with this being, it is not considered enough, by some teachers, to teach about system design [8].

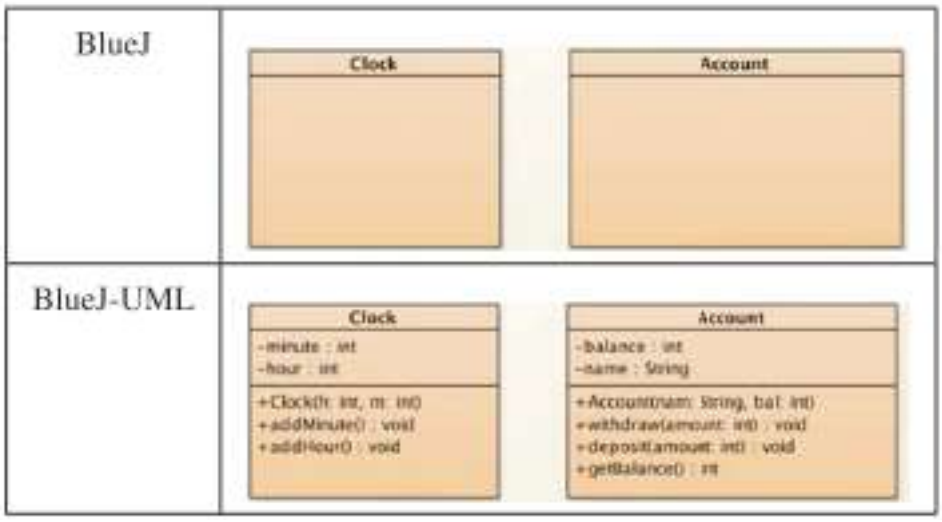


Figure 2.5. Comparison done between BlueJ-UML and BlueJ regarding a class details [8].

BlueJ poor information regarding class correlations, is done by representing a use-dependency, and may represent a difficulty that is too abstract for beginners to understand. Therefore, there was the need for introduction of better association, aggregation and composition dependencies. BlueJ-UML reflects these dependencies from the source code and allows students to interact with the class diagram while updating the source code as well [8].

With this being, this extension tool offers a bi-directional reflection from source code to class diagram. This means that it highlights the connection between what it visible in the graphical section, and the textual source code that was written. It reflects the updates done in any of the sides, on both of the sides, source code and UML visualization.

Figure 2.6. Updating details while on UML viewpoint.

The results obtain with this pedagogical extension of the BlueJ environment showed that it was well introduced in Software Engineering courses and largely used by programming beginners [8].

All the tools above help the beginner to better understand programming basics in order to resolve problems, but they don't help much when it comes to indicating issues related to the quality of the provided solutions. They don't emphasize the importance of having the habit to write better good quality code, which means that beginners, without the help of a teacher, most certainly won't realize the bigger amount of issues present in their code, regarding code quality.

2.5 Professional tools for addressing code quality

Using tools which purpose is to assist facilitate the craft of writing code, has been a must for almost every experienced software developer. They point out bad coding practices regarding mostly syntax issues, suggest better ones, help preventing errors, and by doing this, they allow the improvement of the developer productivity. Since the beginning of software engineering, the number of available development tools has been subjected to extreme growth, being that every developer, independently of the used platform, has at disposal, a wide variety of possible tool combinations to choose from. Each one of these tools can offer a different set of features, such as different UML modelling, code management solutions, database editors, and much more [17]. Therefore, their primary objective is to make the developer life easier, by removing some of the margin for error. A set of these tools will be described hereinafter.

2.5.1 Eclipse IDE

At its essence, Eclipse is a platform where applications can be built, and code can be written, but it was developed with much more in mind. It is an Integrated Development Tool (Eclipse IDE) and represents a platform where developers can build their software projects. It consists on a toolset that provides a wide variety of plug-ins, while being also designed to be extensible using additional plug-ins and is considered a software ecosystem. A common user interface (UI) for working with tools is provided, while the platform handles all the logistics regarding the execution of the right modules of code. One of main purposes behind Eclipse is to allow and facilitate the development of an application, by allowing each team to easily focus on a specific part of the project, without worrying too much whether it will be compatible with the rest of the team's work [18].

As displayed in figure 2.7, eclipse contains multiple plug-ins/extensions, such as the main ones: package explorer, outline, code documentation… that are represented as components in the user interface. Beside these ones, there is a marketplace full of external plug-ins, that can be installed to better suit the development, and its requirements. Every single one of these plug-ins helps the developer keeping the project organized, well-structured, allows the detailed visualization of every programming element involved on the development, and aim to assist the developer writing good quality code. For instance, two major features that eclipse and many other IDE's offer, are syntax and error highlighting. These two often help the developer catching eventual compilation errors that may occur once the code is compiled. These kind of features are also major time-savers, in the meaning that the time saved by warning instantly after the errors are produced, instead of needing to save, compile, and then get the error warnings, the developer can abstract from those concerns and utilize that time to focus on the actual piece of code that is being developed, and whether it makes sense and will work, or not. The pedagogical tool that this theses addresses, was built as a plug-In for the Eclipse IDE.

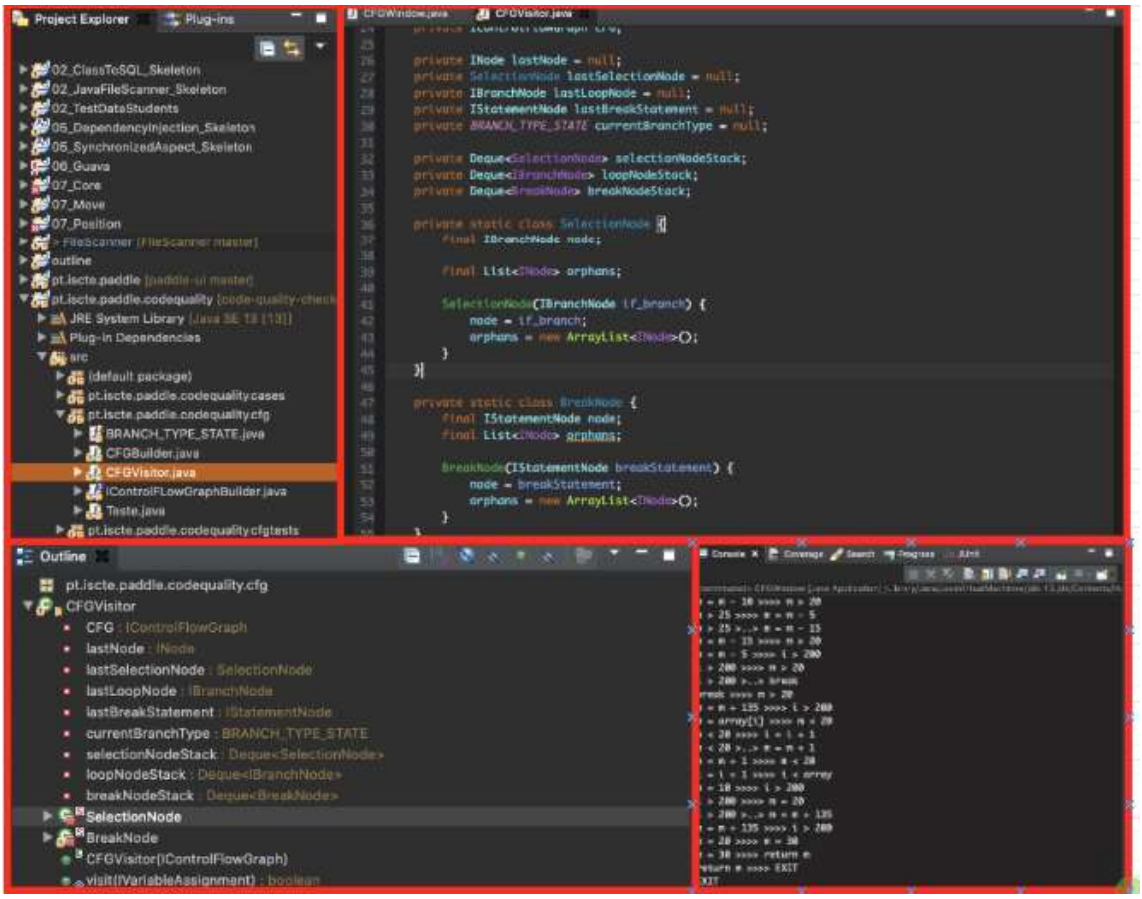


Figure 2.7. Eclipse IDE user interface displaying different plug-ins [16].

2.5.2 IntelliJ IDEA

IntelliJ represents another type of IDE, largely used for Java development. In a lot of ways, it is similar to eclipse, but it offers and more professional experience, and with this being, it is targeted at medium to advanced developers. It has become one of the most, if not the most used IDE, when it comes to developing software. This reason for this is that it offers assistance features that facilitate code navigation and refactor much easier, resulting in the capability of developing programs and applications that are very well designed. There is also autocomplete, that takes what into context, what the developer starts writing, and automatically suggests the full expression that would have to be written, which comes very handy. There are code inspections, that detect and correct anomalous code in the project, before compiling it, which can find various problems, such as code that will never be executed (dead code), bugs, spelling problems, and by doing this, contributes to the overall improvement of the code structure [19]. It also has a lot of features regarding code quality, such as syntax error highlighting, that is very useful for reasons that have already been explained in the Eclipse section above. In general, it is very similar to eclipse in a lot of ways and offers some other features that are appreciated by some developers. Some may say that IntelliJ is better than Eclipse because it lets you write and change code very quickly and easily, and suggest a lot of things based on the context

that the developer is on, but in the end, it comes to personal taste, whether one chooses to use Eclipse, IntelliJ, or some other IDE.

IntelliJ represents another type of IDE, largely used for Java development. In a lot of ways, it is similar to eclipse, but it offers and more professional experience, and with this being, it is targeted at medium to advanced developers.

2.5.3 Symbolic Execution Debugger

The Symbolic Execution Debugger (SED), is an extension that was built for the debugging platform of the Eclipse IDE and represents a graphical visualization of the program's execution flow and state inspections. It is based on the symbolic execution concept, that represents a way of analyzing code by figuring out which input parameters will trigger the execution of each part of a program. The symbolic execution can start at any program point, and the given parameter will restrict the execution of the program, to the path that is of interest to be analysed. It is suited to assist in code reviews, that can the state of the code quality.

The tool is the result of mixing interactive debugging, that allows a running program to be stopped and its current states, and values, checked or altered [20], with the symbolic execution functionalities and capabilities [21]. It goes beyond what traditional debuggers, like the ones present in Eclipse and IntelliJ, can do, and offer developers a better way to debug written code.

The tool starts by creating a symbolic execution tree, that represents the execution flow that is followed, depending of the parameters that are supplied, as shown in the figure below.

This allows the debugging to be made by controlling the execution, this meaning that a program can start at a point of interest, while being visualised in the symbolic execution tree, avoiding the build-up of large data structures, in more complex cases. This way, the execution can be controlled and comprehended in a better and more efficient form. This allows the developer to actually see If a failure is occurring in the chosen execution path, and the debugging process becomes much more intuitive, and eventual bugs or code related issues can be rapidly found, and resolved [21].

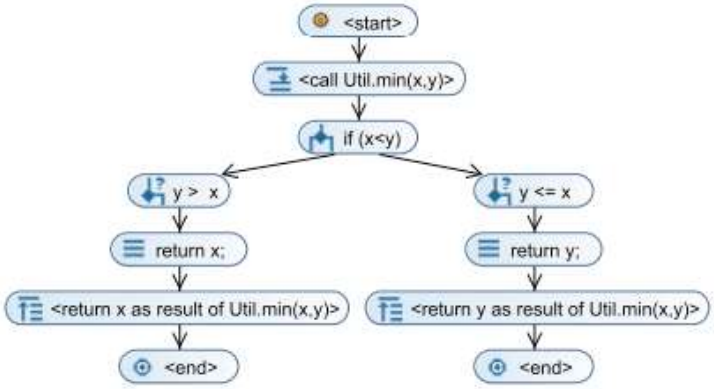


Figure 2.8. Symbolic execution tree example.

2.5.4 Code Linters

There is also a specific type of programming tools, that focus on automatically spot code issues on written programs or applications. They are of extreme use, when it comes to alerting the developer for some errors or mistakes, that otherwise could go unnoticed for a long period of time. It is a type of tool that can be installed in many IDE's, such as Eclipse and IntelliJ, and perform a static analysis of the written code, this meaning that it actually compiles the code, in order to check and search for syntax error, typing mistakes, etc. This analysis is done based on the language that the code is written on. This kind of tool has a great value associated to it and should always be included in the development process, since it can help the developer finding possible mistakes, and acquiring good code quality habits, regarding mostly syntactic aspects. Several studies have shown that developers use linters for several reasons, such as:

1. Error prevention, where linters help catching possible errors in the code base, from a syntactic point of view, such as mistyped variables names, that were missed by the developer, before becoming a runtime bug. The great advantage in this is that it can catch these mistakes from an early stage of the development process, guaranteeing that they do not spread, as the application grows.
2. Ambiguous code, that helps to refactor code that, otherwise, could be seen as hard to read, this meaning that is promotes better coding practices, that allow other developers that look at the written code, to better understand what it represents.
3. Maintaining code consistency**,** where linters help the developer maintaining a consistent code base style. This consistency is beneficial in a lot of reasons, one being that it improves the reading and comprehension of the code across a project.

With this being said, it is possible to say that a good code linter represents a powerful asset while building an application, because it will surely help the developer keeping an eye open for possible

mistakes that may happen, and help on fixing those mistakes, and replacing them with better coding habits, so that they tend to happen less.

2.5.5 SonarLint

SonarLint is one of the most famous linters. It also makes it possible to get instant feedback on code quality issues during development. It does this by offering features such as bug detection, that by statically analysing the written code, tries to match it with thousands of rules, across different programming languages, and by doing this, it is able to recognize when some code breaks one of those specified rules. These rules are extended over three categories [22]:

Bugs – Detecting syntax mistakes that could lead to unwanted behaviour and performance issues.

Vulnerability – A security issue which can mean a possible opening for an attacker.

Code Smell – Clean coding violation which may lead to maintainability issues.

A simple example of what SonarLint can do, is warning the developer for the risk of existing a null value, that could result in an exception thrown, like demonstrated in the figure below.

Figure 2.9. SonarLint warning example [22].

There is a big community that supports and uses SonarLint as tool in their developments.

2.4.6 PMD

PMD is a source code analyser, that focuses on finding common programming flaws in written code, like unused variables, empty catch blocks, unnecessary objects, and many more, with the ultimate goal of assuring good code quality. It runs by scanning the source code for patterns, that are compared to a set of rules and can indicate problematic or flawed coding practices. It can catch solutions that are too complex and can compromise the application performance and maintenance, so it can be seen as a code reviewer.

The process begins by parsing the source code, which consists of two steps: lexing, which produces a stream of tokens, and parsing, which produces an abstract syntax tree (AST), that is a memory representation of the elements present in the source code. After this is done, PMD can concentrate itself on analysing and validating the syntax present in the code, searching for fields, methods and variables and their usage within the scope of the source file. By doing this, it can determine whether these fields, methods and variables are, in a matter of fact, used. Then, the

variables are checked for definitions, assignments and reassignments, in order to allow the detection of issues. After the AST has been successfully initialized, the set of rules can be applied to it, and executed. Each one of these rules has the possibility of reporting violations, resulting in a report that will later be presented to the developer. PMD also allows a developer to elaborate and new rules to it, using a rule designer user interface. Another interesting feature is the illustration of metrics regarding the results obtained from the code analysis [23].

2.6 Summary

During the first part of the literature section, there was an enumeration and analysis regarding which are the most common mistakes made by beginners, and what leads into the formation of misconceptions on their minds. Followed by that, it was found that the best way to avoid and combat the misconceptions, is by supporting beginners with an educational intervention that focuses on how beginners can fix them, or avoid introducing them in the first place [6].

In the second part, after enumerating all of the tools above, it was possible to conclude that there are already some offers when it comes to handling code analysis, with the purpose of warning about code issues, bad coding practices, and advising better ones, but all of this tools are based on an syntactic analysis approach, and they tend to warn about syntax related issues, without approaching in depth, the semantics of the problem and the origins that led the developer to commit such mistakes. This is the part that can be crucial to a beginner.

This kind of guidance that beginners need, regarding the quality aspects of their code, is what is meant to be obtained with the proposed pedagogical tool. It should serve has a backbone in which students can rely to serve has an explainer and a guider that will help them to understand where issues are, the reason behind those issues, and how to solve them. All these combined efforts have the common goal of helping the beginner writing better code.

With this being, the future steps in the building process of the proposed pedagogical tool can be made with the previous conclusions in mind.

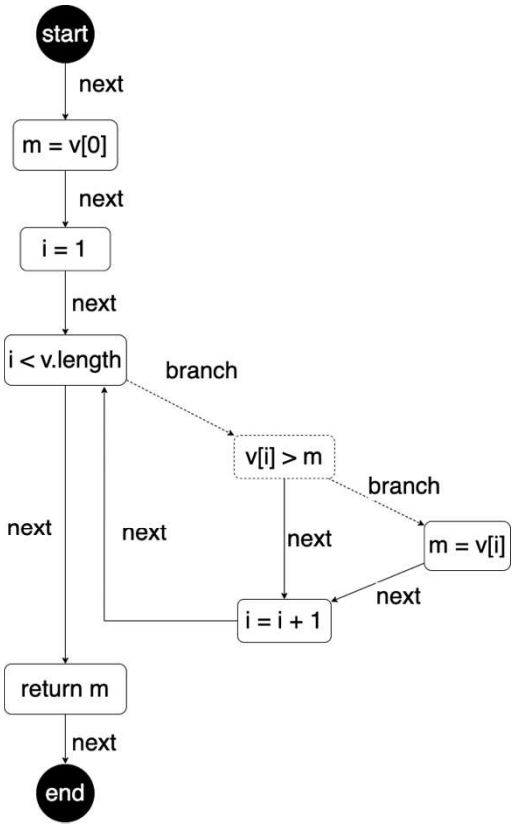
3. Control Flow Graph

The first stage of developing the pedagogical tool was to implement an algorithm that constructs an in-memory structure containing graph-oriented information regarding the execution flow of a program. This structure will later be used to implement most of the pedagogical tool detectors.

3.1 Definition

A Control Flow Graph (CFG) is a representation of the possible execution steps of a method. It consists on a directed graph, whose nodes represent instructions, connected by edges that represent instruction subsequence. An edge from instruction *x* to instruction y means that *y* will execute immediately after x. A CFG is a structure containing information regarding all the possible paths and that may be executed during the runtime of the method, which can be useful in order to better understand the types of events present in the execution of the program. It can help a beginner that may be facing some difficulties on understanding the semantics behind a piece of code, by removing the syntactic complexity of raw code. The CFG structure will be used further down the line, during the development of the pedagogical tool.

```
int max(int[] v) {
    int m = v[0];
    int i = 1;
    while (i < v.length) {
        if (v[i] > m) {
            m = v[i];
        }
        i = i + 1;
    }
    return m;
}
```



Figures 3.1.a & 3.1.b. Code method example.

Figure 3.1.b shows the visualization of a Control Flow Graph, that represents Figure 3.1.a method's execution flow. The different statements and expressions present in the method, are represented by nodes, which are the rectangles in figure 3.1.b. The nodes that have a dashed contour

are called *branch nodes*, which are nodes where a decision is made, such as the node which contains the while loop condition, from the method on Figure 3.1.a. These nodes can either represent the guard of either a selection, or a loop. The other type of nodes represents the basic statements and expressions such as the assignment of a variable's value, such as on the start of Figure's 3.1.a method (ex: int i = 0), function calls, etc, and are called *statement nodes*. The arrows between nodes represent the graph edges, meaning that they indicate the node that shall follow, after the one that is being visited. The dashed arrows are the representation of flow when the expression of a branch node evaluates to true. By establishing this kind of connections between the *statement* and *branch* nodes, it becomes possible to build a graphical visualization, such as the one above, which can be useful while trying to comprehend a method's codebase, since it displays, step by step, the method's execution path.

3.2 Paddle library for code analysis

Building such a structure was made possible by using the Paddle[24] library that creates an Abstract Syntax Tree (AST) of the codebase, which consists on a memory representation of the method codebase. Paddle allows traversing the AST by means of an infrastructure based on the *visitor* pattern, which was applied to the generated AST, and as a result, a variety of visitors, that are triggered by multiple types of events, were made available. These visitors gave, for example, information regarding the start and end of a loop or if selection blocks execution. With them, it was possible to know when any of the multiple types of blocks were executed. They also provide arguments which can be accessed in order to obtain information, such as a variable's value, regarding the operation that was made. With that, it becomes possible to build an algorithm that makes use of all these listeners, and which builds the Control Flow Graph. A few examples of this listeners are:

Visitor API	**Description**
visit(IVariableAssignment)	When any variable is assigned a new value.
visit(ISelection)	When a selection condition, such as an if block, is entered.
visit(ILoop)	When a loop block condition is reached.
visit(IMethodCall)	When a method or procedure invocation is performed.
visit(ISelectionAlternative)	When a selection's elseif/else block, is entered.
visit(IReturn)	When a return statement is performed.
visit(IContinue)	When a continue statement is performed, on a loop.
visit(IBreak)	When a break statement is performed, on a loop.
endVisit(ISelection)	Selection block visit ends
endVisit(IElseBlock)	Selection's else block visit ends.
endVisit(ILoop)	When a loop visit ends.

Table 3.1. Paddle API's

3.3 CFG Building Algorithm

The general idea of the algorithm is that the Control Flow Graph is built on the fly with one depth-first traversal of the syntax tree This means that from the moment that the visitor starts running through the code, every time that a code block visit happens, it is seen as a trigger that shall apply modifications to the list of nodes that form the CFG. For example, when a new value is assigned to a variable, an IVariableAssignment visit will be triggered, which shall add a new node to the list which represents the value's assignment to the variable.

In order to create the flow, the nodes must be connected in a way that each and every node, with exception of the start and end nodes, must have information regarding which nodes point to them (incoming nodes) and to which nodes they should be pointing to, (outgoing nodes) or branch nodes. These connections are symbolized by the arrows in Figure 10.b.

Nodes can be of two types:

- *Branch nodes* – Represent either if/else or loop blocks, where there are two possible outcomes regarding the method's execution path.
- *Statement nodes* – Represent every other type of statements, such as variable assignments, method returns, loop breaks, continue statements, etc.

The common attributes shared between the two types of nodes are:

- *Next node*, that represents the node that will follow in the method execution flow.
- *Incoming nodes*, which is the list of nodes that have the current node set as their *next* node.

Besides this, branch nodes also contain:

1. An *alternative* node attribute, which represents the direction that is pursued when an if or a loop condition is true. In the contrary case, meaning that the condition is evaluated as false, the execution shall follow the *next* node.

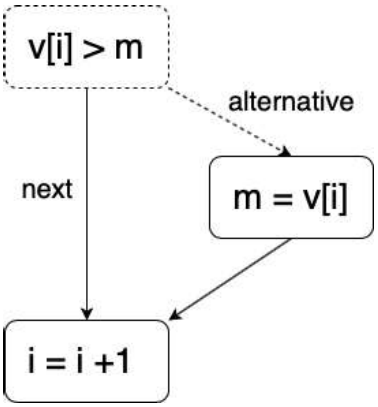


Figure 3.2. Next and Alternative attributes example.

Every time that a method's statement, or block, is reached, its correspondent visitor operation is called and the information regarding that statement becomes available. This information is then used to create a new CFG node, which is then added to the main structure node list. After the node is created, it is necessary to update the preceding nodes, which should set the new node as their next node.

This process is repeated for every statement and every block present in the program, with the exceptions of the first node that is created, that shall be set has the CFG's entry node *next*, and when a return statement is reached, that shall, right after its creation, set its own next as the CFG's exit node. The result will be a Control Flow Graph that has a node list containing all the nodes corresponding each statement present in the visited method, which hold references to the subsequent statements.

3.3.1 Algorithm Explanation

When a statement occurs, such as a variable's assignment, there are a few steps that take place in order to add the statement to the CFG structure. The actions performed in order to do so, are the following:

1. Assignment statement visitor is triggered.
2. Create a new *statement* or *branch* node and add it to CFG nodes list.
3. Check the previous nodes which have pending connections, such as missing *next* or *alternative* attributes, and if the required conditions are met, update these fields with the value of the new statement node that was just created.
4. Set the attribute that indicates the last node that was visited as the value of the new *statement* or *branch* node.

Regarding the third step, where all the logic needs to be applied in order to create the correct connections between the nodes in the CFG node list, a more in-depth explanation needs to be given, in order to better understand the logic behind building a CFG structure.

On the third step, there are a few conditions that must be checked, regarding which nodes shall set the new node that was created, as their *next* node. Depending on the type of visitor that is triggered, the actions performed may vary.

These conditions make use of a series of stacks and variables that are kept in memory, which hold information about the nodes that represent statements and code blocks, that have already been visited, such as if/else, loops, loop breaks, and continue statements. The element that sits on top of each stack, corresponds to the most recently visited statement, or condition block, of that type. For example, every time that an if/visitor is triggered, the stack which contains this kind of block, is updated with the new branch node that was created. Once the block is left, the variable which represents the last visited if/else block node is assigned with this node's value. This way, it becomes possible to access information about statement or branch nodes that have already been left and which correspond to key elements in the execution flow, that need to be accounted in the conditions involved in creating the connections between nodes. These stacks and variables hold a lot of information regarding the type of code structure that they store, such as the condition of an if, the code instructions inside it's block, if it has an else block and if so, the instructions inside such block, a lot more. Same thing for all the other types of statements and blocks that were mentioned.

The main purpose of keeping these structures in memory, is the role that they have when it comes to the conditions that need to exist during the creation of new connections between nodes, otherwise, it wouldn't be possible to know if one node should in fact set another node as its *next* attribute, or if it should be a different node that should take that place.

During the process of establishing a new *next* or *alternative* connection between two or more nodes, there are some steps which represent the conditions that need to be fulfilled.

The following steps represent the actions and conditions that have to be handled upon visiting a code statement, such as a variable assignment, a method invocation, etc:

1. Create a new *statement* node and add it to the CFG structure.
2. Check if the last selection node or last selection loop variables have a valid value, which means that there are nodes that correspond to an if/else or a loop block, which have already been left, and that still have the *next* attribute pending for a value. If there is one of these nodes, or both, that pass that condition, set the new *statement* node as their *next* node.

3. Check if the last break node variable has a valid value and check if the new *statement* node corresponds to a statement that is outside the loop block where the break was called. If both of these conditions are true, set the last break node *next* attribute as the newly created statement node.
4. Check if the last node that was added to the CFG is a branch node and if it is, and if it does not contain a defined alternative node, define the new statement node as it's *alternative* node. If the last node isn't a branch node, do the same condition and check if has a defined *next* node, and if it does not, set the created node as its next node attribute.
5. Set the CFG's last node attribute with the new statement node value.

In the case of the visit being on an if/else or loop block, the necessary steps are very similar to the ones described above, with the differences that:

- In step 1, The created node should be a *branch* node, instead of a *statement* one.
- The new *branch* node needs to be added into the selection or loop node stack.
- Upon finishing the visit to the new *branch* node correspondent code block, the new node needs to be removed from its stack and assigned to the variable that represents the last selection or loop node to be visited.

Also, upon finishing a block visit, of any kind, every node that was created from statements that were inside that block, and that remained with a pending *next* attribute, were called orphans, and were stores in a list so that they could be assigned a *next* value, once the visit to that block was over. The reason behind the existence of these list of orphans is that, for example, a statement node that is inside an if block, can't set its next as the statement node that can be found in that if's else block. It has to wait once the if/else block is left, to set its *next* attribute as some other node that is outside that block. Once the visit to such block is over, the next node, either being a statement or branch node, can check if the list of orphans isn't empty, and if so, set the itself as the next node for all the nodes that are present in that list of orphans.

Besides the statements and blocks mentioned above, there are also the *continue* statement and the *return* statements. In both of these cases, both steps 1, 4 and 5 needs to be performed. The only difference between the visit to a break statement and a continue statement, is that when dealing with the node that corresponds to the continue, the *next* attribute is set to be the loop branch node that is in, which can be consulted by querying the loop branch node stack, which is kept in memory. When it comes to the return statement, the first step, after creating its *statement* node, is to set its *next* as

the CFG exit node, and once that is done, the orphan list is checked and if there are any nodes that are missing a **next** node, the new return statement node is set as that value.

After covering all the conditions that are mentioned above, it becomes possible to create a dynamic algorithm, that should generate a CFG based on any code method that shall be supplied.

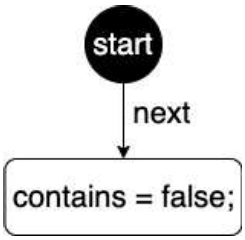
3.3.2 Control Flow Graph Generation Example

```
static boolean containsNumber(int[] vector, int number) {
   boolean contains = false;
   int i = 0;
   while (i < vector.length) {
         if (vector[i] == number)
               contains = true;
         i = i + 1;
   }
   return contains;
}
```

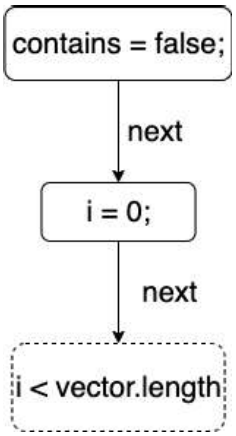
Figure 3.3. Example method that checks for the existence of a given number.

In order to better understand the building process that is involved on the construction of a Control Flow Graph, the method serves as an example from where a CFG can be generated from. The steps involved would be:

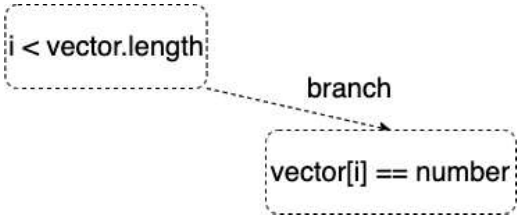
1. The first step would be to add the **start node** into the CFG's nodes list.
2. Upon triggering the variable assignment visitor, create a new statement node that corresponding to the "**contains = false"** assignment.
3. Since the last node in the list is the start node, the first connection between nodes can be created, by setting its next as the "**contains = false"** statement node.



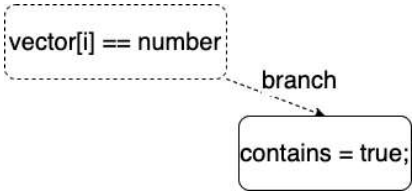
4. Do the same thing as the previous step, with the difference that the created node shall be a branch node that represents the loop condition, instead of a statement one.
5. Set the new branch node as the second line's statement node next.



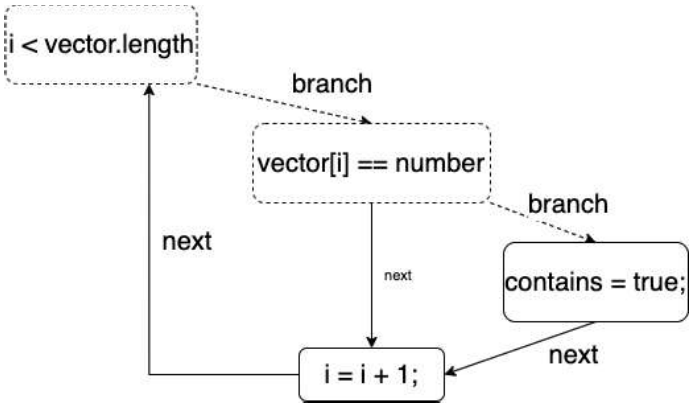
6. Create branch node that represents the if block condition.
7. Check that the last node in the list is a branch one and set the new selection node as its branch.



8. Create the statements **contains=true** node.
9. Repeat step 7.



10. The end of the loop triggers a visitor which means the information regarding the last loop that was visited, is stored.
11. Use the information from step 10 to set the loop's condition node as the next node for the last statement node in the list.



12. Create the statements **return contains** node.
13. Since the loop doesn't have a next node, set the return statement node as its next node.
14. Set the exit node as the return statement node next.

The final result is:

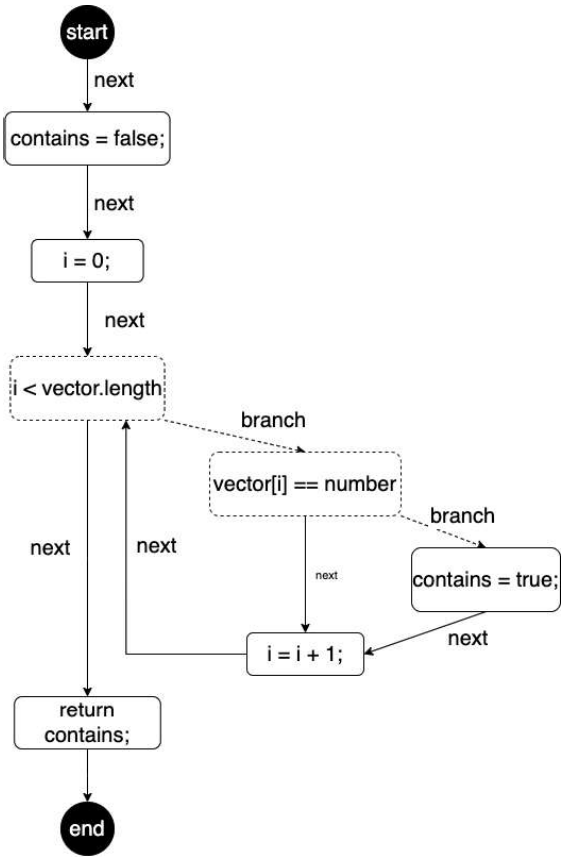


Figure 3.4. Completed Control Flow Graph structure.

3.3 Control Flow Graph APIs

The nodes that are present in the CFG structure, have the following APIs available:

Node API	Description
getElement()	Statement of code that it represents.
getNext()	Next node in the graph, to where the execution flow shall follow, when the condition is true.
getIncoming()	List containing the nodes that have the current node set as their **next** node.
isEntry()	Informs whether the current node is the entry point (first node) in the CFG.
isExit()	Informs whether the current node is the exit point (last node) in the CFG.
isEquivalentTo(Node)	Compares the current node with the supplied one and informs if whether they represent the same node in the CFG, or not.

Table 3.2. Statement Node APIs.

In addition to the above operations, a **branch node** also has the following ones:

Branch Node API	Description
getAlternative()	Next node in the graph, to where the execution flow shall follow, when the condition is true.

Table 3.3. Branch Node APIs.

The complete Control Flow Graph structure provides five API's that can be used by someone who needs access to its nodes, in different ways. These are:

CFG API	Description
getNodes()	Returns a collection that contains the Control Flow Graph nodes.
deadNodes()	Returns a collection of nodes which are that correspond to segments of the method code that are considered unreachable code, which means they will never be executed.
pathsBetweenNodes(start, end)	Returns a list containing all the possible paths that exist between the provided source and destination nodes.
getSubCFG(start, end)	Returns a new Control Flow Graph that is a subset of the original one. This sub-CFG contains the nodes between the provided source and destination nodes.
usedBetweenNodes (start, end, variable)	Returns whether a variable value has been used or changed between a source and a destination node.

Figure 3.5. Control Flow Graph API's.

Overall, a graphical Control Flow Graph like the one displayed above (fig. 10), can be of great use while explaining to the beginner the causes of a code quality related issue like for example, it can visually target and indicate where the issue that was found is, or the path that was travelled by the execution, because of that issue.

4. Code Quality Issues

This section will focus on explaining the process of building the core of the tool. After developing the Control Flow Graph structure, mentioned in the previous chapter, the next step in order to start building the tool was to choose a set of code quality related issues that are considered bad practices regarding the quality aspect, and then working towards a first prototype, that should be able to recognize such issues in the students codebase and warn them when such occurrences happen, while providing an explanation on why they should be avoided, and sometimes, providing a suggestion on how to fix such issues. The initial list of issues which were chosen to be implemented in the tool were:

Issue name	Description
EMPTY SELECTION	When an if block is left empty.
MAGIC NUMBERS	Numeric literal that is used without an explanation.
FAULTY RETURN BOOLEAN CHECK	Unnecessary check of a boolean value before a return statement.
FAULTY BOOLEAN CHECK	Unnecessary condition syntax.
UNREACHABLE CODE	Block of code that can't ever be executed.
USELESS RETURN	Return that serves no use in the execution context.
DUPLICATE SELECTION GUARD	Double-check of a condition.
FAULTY METHOD CALL	Unused method value.
NON-EXPLICIT ELSE	If block left empty while code is written in the else block.
TAUTOLOGY	Condition that is always classified as **true**
CONTRADICTION	Condition that is always classified as **false**
DUPLICATE CODE	Duplicated lines of code between if/else if blocks
USELESS CODE	Instructions that don't change program execution.

Table 4.1. Code quality issues and descriptions.

The tool core consists on the collection of code quality issues that were chosen to be featured. The process of adding a new issue to the list was to first being able to recognize when it occurs in the beginner's code base, followed by the creation of a structure that contains information about that bad practice in particular, and then generating a suited explanation that is displayed to the student. Each of these cases, that tool core consists of, will be explained in four steps:

1. Explaining the probable cause, like a misconception, that originated the issue.
2. Describing the steps involved in recognizing their occurrences.

3. Giving a real code example of the its occurrence and highlight, using the developed tool.
4. Displaying the warning and explanation that the tool generates.

The recognition of an issue occurrence in the codebase, consists on a process that made use of the same tool API's that were used to develop the Control Flow Graph structure, which is based on the **visitor** programming pattern. The Control Flow Graph itself was built in order to help recognizing some of the quality issues, and its usage will be explained in detail in each issue section, further ahead. But independently of the recognition process, after recognizing each issue, a new structure is created, containing information regarding the issue occurrence. A quality issue structure contains the following attributes:

- Category – The type that describes the quality issue.
- Classification – How sever the bad practise is. An issue can either be classified as *LIGHT*, *AVERAGE* or *SERIOUS*.
- Explanation – The reason why it is considered a bad practise and eventual tips on how to fix it.
- Element – The object that represents the place in the code base where the occurrence was found.
- Decorations – Visual effects used to highlight the issue occurrences.

Every time that an occurrence is caught, a new quality issue, containing the structure above, is created and added to the tool's issue collection, in order to be displayed to the beginner.

After the recognition phase is over, the quality issues that were caught are then displayed in an User Interface containing highlights such as code marks that may vary in colour, depending on the classification that was attributed to each issue, where the *serious* issue are marked as red, the *average* ones as magenta, and the *light* ones as blue. There are also textual hints that may warn or provide tips to fix the issues, and more detailed explanations that aim to make the beginner understand why the caught issue is considered a bad practise.

The first set of quality issues that were implemented into the tool, were the ones corresponding to empty code blocks. This means that every time that a beginner left empty blocks, such as if selections or loops, in the codebase, a new empty block issue would be recognized, generated, and added to the tool.

4.1 Control Structures

The first category of issues that were chosen to be detected were the ones involving condition blocks. These issues focus on highlighting some of the misconceptions that a beginner may have formed form

regarding the building process of a condition block, such as an if, or a loop, and how such practises can negatively affect the codebase readability and maintainability.

4.1.1 Empty blocks

The simplest cases of condition related issues are the code blocks such as an empty **if, else** or **loop**, that are left empty, without holding any actions inside of them. This means that every time that an empty block is left on the code base, the beginner would be notified with a highlight on such block and an explanation on why such cases shouldn't happen.

```
boolean greaterThan(int a, int b) {
   if (a > b) {
         return true;
   }
   if (a < b) {
   }
}
```

Figure 4.1. Empty code block example.

An empty condition block is one of the most basic quality issues that can be caught. It holds a condition that is checked, but doesn't add anything relevant to the code execution, since it has no actual actions inside that are executed every time that the condition is verified. A beginner leaving such blocks can be a fruit of simply forgetting to remove such block, because it is not necessary after all, or to add relevant logic to it. With this being, empty blocks are highlighted to the beginner in order to raise awareness about this kind of possibilities.

The process of catching such occurrences made use of one of the **visitor** API's, which corresponds to the one that gets called when a selection, such as an if block, is reached. The API parameter represents the selection block that is being visited and contains multiple API's that supply information regarding its condition, the block itself that contains the executed code when the condition is true, and many others. One of these API's, isEmpty (), returns a Boolean value that informs about the selection block is empty, or not. This last API is the one which was used in order to recognize when an

empty selection is left on the codebase. The next figure shows an example of such occurrence and its highlight and explanation.

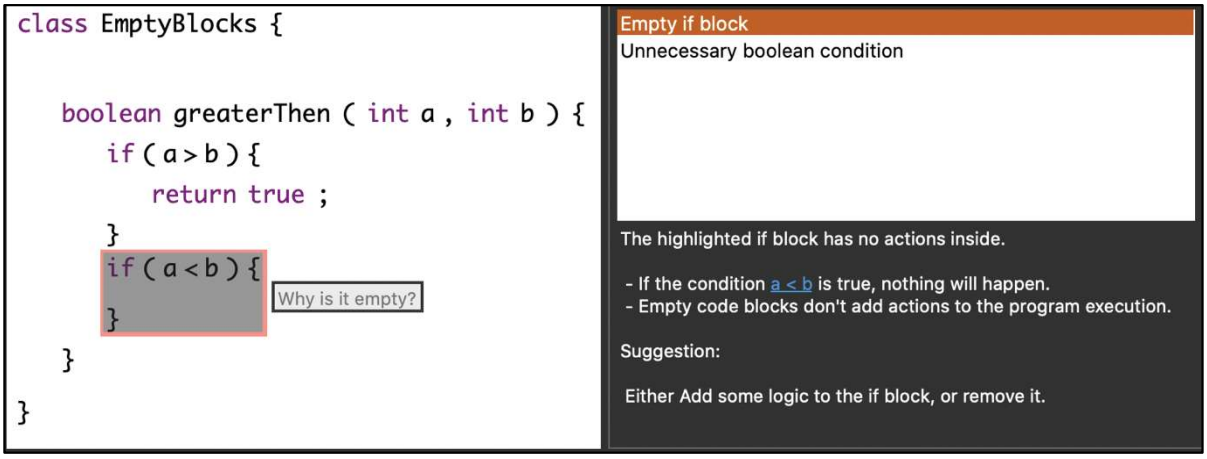


Figure 4.2. Empty if block issue highlight and explanation example.

The highlight has the colour red, due to the issue being classified as serious. The provided explanation aims to warn the user about that an empty if block leads to no actions performed, even if the condition is truthful, and with this being, the block should either be filled with some actions, or removed in order to avoid leaving such empty blocks in the codebase.

4.1.2 Non-explicit else with empty if

It was considered to be a **non-explicit else**, when an if/else block is caught in the codebase, where the if section was left empty, and the code is written in the else block. This type of issue is most likely due to the beginner not knowing that an if condition can consist of a negation, or due to thinking that a condition can only be evaluated as true. This means that when the beginner thinks that the only way to write code that gets executed when the condition is false, is by writing it on the else block. This results in an empty block of code, that could be avoided by applying the negation operation to the condition and writing the code directly in the if block.

```
void addIfDoesNotExist (int[] a, int b, boolean exists){
    if(exists) {
    }
    else {
        a[a.length - 1] = b;
    }
}
```

Figure 4.3. Non-explicit else example.

The recognition was possible by applying the same procedure as the previous cases. By using the API that returns information about the if block's in the code, it is possible to find out the if condition contains an else block. Combining this with the API that informs whether the if block is empty, which can also be applied to the else block, it is possible to recognize the described code quality issue.

The generated result is:

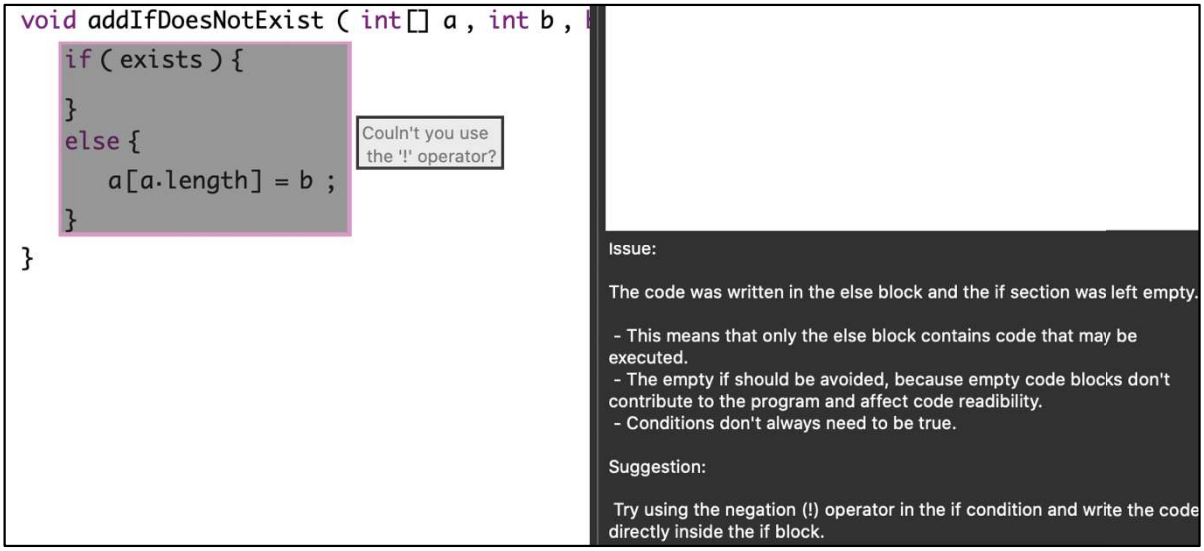


Figure 4.4. If Condition Misconception highlight and explanation example.

Like it can be seen in the figure above, the highlight is marked as magenta, which means that this issue is considered to be average. Both the highlight and explanation aim to make the beginner understand that empty blocks should be avoided because they don't add actions to the program execution, that conditions don't always need to be true, and to raise awareness about the negation operator, that could be used in this case in order to avoid having an empty if block.

4.1.3 Boolean Condition Verbose

When It comes to writing condition, the most common way to form a negated one is by adding the negation operator (!) behind the condition itself. But students very often apply the comparison between the Boolean variable and a literal value (**true** or **false**). This is more of a styling matter, but it can really affect the readability of a condition and for this reason, it was considered to be a quality issue, even though it was classified as a light error.

```
...
if (exists == true) {
   return true;
}
...
```

Figure 4.5. Boolean verbose example.

The reason why a one would practise such issue could be that, like said in the previous issue, the student might not be aware about the existence of the negation operation and readability benefits that it holds and won't use it because of this.

In order to recognize such case, the API used was the one triggered by the occurrence of a binary expression, that consists two parts that are combined by an operator. Once again, the targets were the conditions guards, which operational type would first be checked, by using an API that returned the operation type, in order to figure out if the conditions made use of **relational** operators, such as the equals one (==), or the not equals (!=). If this is the case and both the operation parts types were Boolean variables, then the operations parts values would be matched with the Boolean values **true** or **false**, and if any of the parts matches these values, a new issue match is found.

After generating a new issue structure and adding it to the tool, the result is:

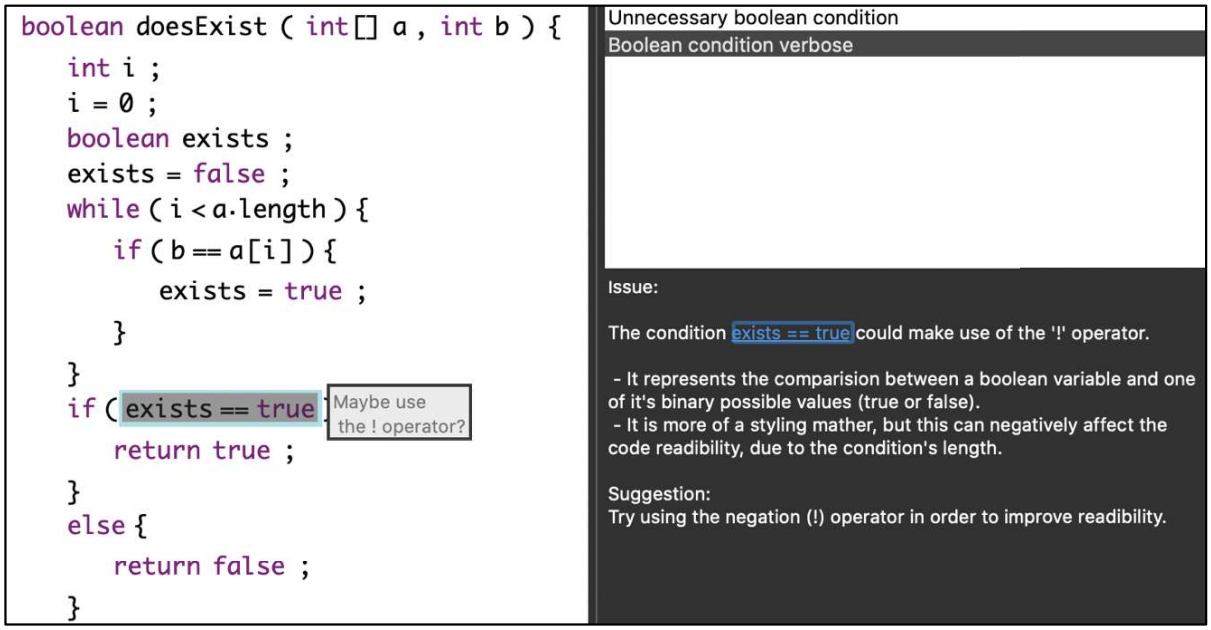


Figure 4.6. Boolean condition verbose highlight and explanation example.

Like seen in the figure above, this time, the highlight colour is blue, due to the issue being classified as light. The explanation warns the user about the points mentioned above, which are that even though this case is more of a code styling matter, it can also affect the readability of the code, and because of that, it was considered a quality issue. It is also suggested to the beginner the usage of the negation operator instead.

4.1.4 Boolean Return Value Check

Continuing in the Boolean type related issues, another example of a code quality issue that involves a Boolean valued variable and that is often seen from beginners code solutions, is when a student writes a method which return type is Boolean, and in order to return the value of a Boolean variable from that method, first checks if the variable's value is true or false, and then uses **returns true** or **return false** statements accordingly.

```
...
if (exists) {
   return true;
} else {
   return false;
}

...
```

Figure 4.7. Boolean return check example.

This unnecessary check may be due to the beginner thinking that the return statements from a Boolean method must always be as the previously mentioned, and maybe not realizing that a Boolean variable can be directly returned instead.

The recognition of this issue was made possible using the CFG's node list. By iterating through the nodes, if one of the node statements represents a *return* statement, then its return type is checked in order to verify if it is Boolean. If the last check is positive, then the block of code which contains the return is obtained via an API from the *visitors* IReturnStatement interface. If the obtained block of code is an if block, then its condition type is checked in order to see if it consists of a boolean variable. If this is verified, and if the if block includes the return *true* or return *false* statements, it is then possible to confirm the presence of the described quality issue. A highlight and explanation example to this issue is:

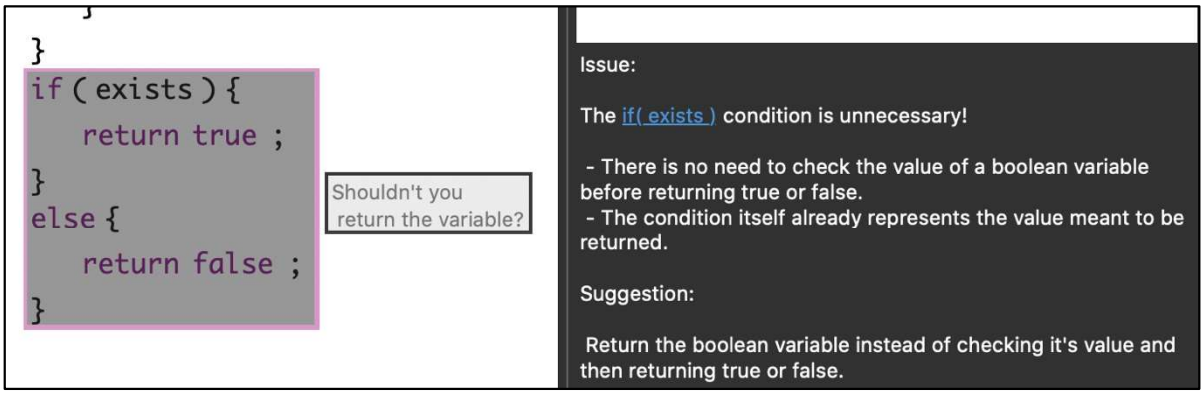


Figure 4.8. Boolean return check example.

This case was classified as *average*, therefore the magenta colour on the highlight and the explanations reinforces the idea that a boolean method does not have to return true or false statements and can instead directly return a Boolean variable. It suggests that the user returns the variable instead of using the unnecessary if else condition.

4.1.5 Tautology

```
boolean isFirstElement (int[] a, int b) {
   boolean isFirst = true;
   if (isFirst) {
         if (a[0] == b) {
               return true;
         }
   }
   return false;
}
```

Figure 4.9. Tautology example.

Some conditions developed by beginners, are written in a way that, independently of the variables values that they may include, they'll always have the same result, which is **true**. This kind of mistake may happen simply because of distraction while writing a condition, or because of the student not being aware of the values of variables that are included in the condition. This means that some combinations of these variables and values used by the student, such as using a simple *true* value**,** may result in conditions that are always evaluated as true, and therefore, form this kind of issue. Since this kind of issue will result in a code block that is always ran, it can generate bugs in the program and because of that, it was classified as being a *serious* one, since it can affect the expected result of a program.

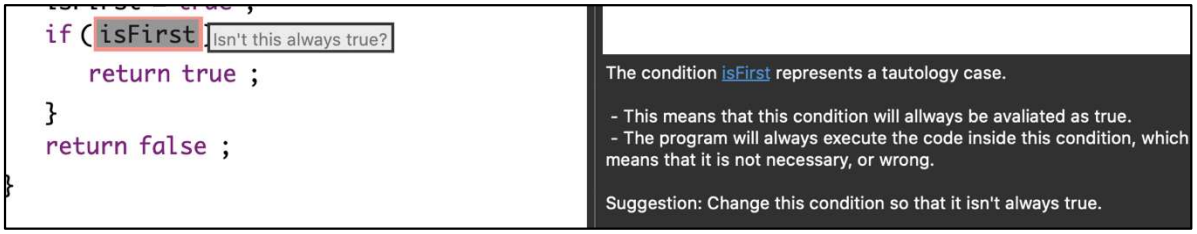


Figure 4.10. Tautology highlight and explanation.

This issue was recognized using the CFG nodes. The nodes were iterated in order to find conditions in a method. When a condition block was found and the literal Boolean value **true** was found by itself, or within a disjunction (||), inside the condition's guard, a tautology cause was found.

When instead of having the literal value *true*, the condition had a Boolean variable, that variable's last modification, or declaration, were analysed in order to check if the assigned value was the Boolean value *true* and if the that variable wasn't changed or used between the last assignment and the condition. This last check made use of the CFG usedOr*ChangedBetweenNodes()* API.

4.1.6 Condition Double-Check

There are some beginners who don't understand that when a condition, or a part of it, is duplicated inside itself, without the variables involved being changed between the conditions, it is considered a case of double-checking. Such thing is totally unnecessary when the entire condition, or a part of a conjunction is duplicated as another condition inside the code block, since in order for the code execution to reach the second condition, it has already been checked and evaluated as true, therefore considering it to be double-checking.

```
...
while (i < array.length) {
   if (i < array.length && array[i] > m) {
         m = array[i];
   }
   i++;
}

...
```

Figure 4.11. Double-check example.

Some students may not be aware that condition will stay evaluated as **true**, as long as the variables involved in it don't change values, and this may cause them to double-check a condition inside itself. It can also be a simple act of distraction. Either way, it is important to point out such issue to the student, since in can make the code confusing by affecting its readability. Because of this, the issue was classified as average.

This issue was recognized using the Control Flow Graph. By iterating its node list, it was possible to recognize when a node corresponded to a condition, and then store it in a list of conditions. Then, every time than a condition was found, it was compared with all the previously stored conditions, in order to confirm two things:

1. If it was placed inside the condition it was being compared with.

2. If the whole condition or any of its parts was matched against any of the parts of the parent condition.
3. If any of the variables included in the matched parts didn't change between the parent condition and its duplication inside itself.

If all three of the points above were checked, then it was possible to confirm that the condition was duplicated inside itself and that it was a case of double checking.

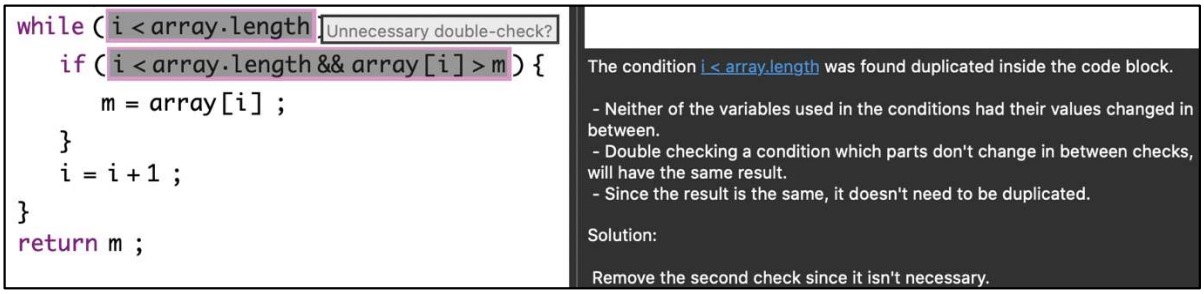


Figure 4.12. Condition Double-check highlight and explanation.

The user is warned that double checking a condition, or part of it, inside itself, while neither of the variables changed value, is not necessary since the code inside the duplicated condition would be executed anyway, and since that is the case, the duplicated condition becomes useless. Therefore, it is suggested to the beginner that the duplicated condition should be removed, in order to avoid unnecessary complexity and damaging the method readability, since the outcome would be the same.

4.2 Assignment & Expressions

4.2.1 Useless Self Assignment

A very common occurrence found in students code is a variable's value assignment, to itself. This may be due to the misconception that assigning a variable's value to itself, will actually change its value, when in reality, it won't. This can also be the result of a typo, where the beginner miss-wrote the name of a variable with a similar name to the one which is being assigned a new value.

```
void max(int[] array) {
    int m = array[0];
    int i = 1;
    i = i;
}
```

Figure 4.13. Useless self-assignment.

The visitor which was used to recognize this issue was the one that is triggered on a variable's assignment happens. When an assignment happens, the string literals of the variable being assigned and the new value, are compared, and if they are the same, means that the variable and the value are equals, which means that the variable is being assigned with the same value that it already had. An example of this is shown below.

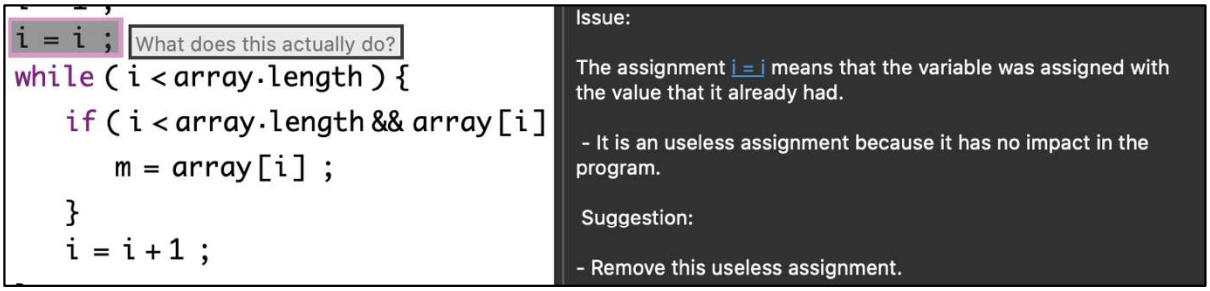


Figure 4.14. Self-assignment example.

4.2.2 Useless Assignment Value

While trying to create a new variable, many beginners have the misconception that in order to assign a value to the variable, it always needs to be initialized. For example, if the result of a method that returns an array of integers, **returnIntArray()**, would to be assigned to a variable, int[] foo, it could simply be represented as int[] foo = returnIntArray(), but many beginners think that the array variable **foo** needs to be initialized like foo = new int[10], for example, before assigning any value to it. This results in a lot of situations where the beginners first initialize a variable with a default value, and only after that, assigns the pretended value to the variable, like for example:

```
void max(int[] array) {
  int m = 0;
  m = array[0];
}
```

Figure 4.15. Useless value assignment.

This leads to the assignment of a value that will never be used. In the case above, the value new int [10] won't be used by the program, ever. With this being, it is possible to argue that the student doesn't realize that a variable doesn't always need to be initialized before being assigned a value Since this kind of issue means that a statement of code will never be used, which can also lead into bugs that are difficult to detect, it was classified as a serious code quality issue.

In order to recognize when this issue occurs, the nodes list from the generated Control Flow Graph were required. By iterating through the nodes, it is possible to recognize when the action that the node represents is a variable's assignment. With this being, it becomes possible to recognize when an assignment is made to a variable that already existed before. After an assignment to an already existing variable is done, the CFG's API that generates all the node paths between two nodes is used between the selected assignments nodes. With these paths, it is possible to check for any statement that makes usage of the analysed variable. If the variable isn't used in any of the paths, then it is possible to conclude that its initial value was never used and with this being, it can be considered useless. The generated explanation was:

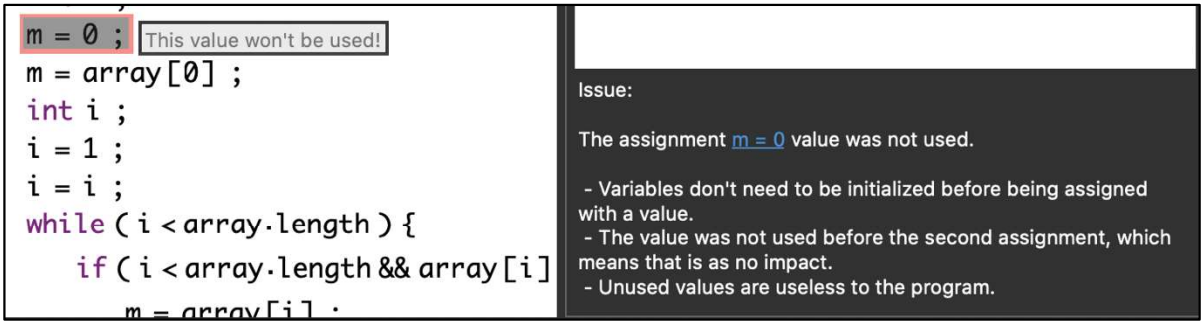


Figure 4.16. Useless Assignment Value highlight and explanation example.

Hopefully, the idea that is transmitted to the student is that the initial assignment value is never used. That is the key to understand why the occurrence was considered a quality issue. The explanation also informs that a variable doesn't need to be initialized, like the one in the picture, before being assigned the desired value. Lastly, it suggested that the highlighted assignment should be removed and the value directly assigned to the variable instead.

4.2.3 Magic Numbers

Using numeric literals in the middle of the code base is a very common practise amongst most programmers, and specially beginners. This is considered a bad practise since assignment a value as a numeric literal to a poorly named variable, for example, provides little to no context regarding what the variable actually represents.

```
double triangleArea(int radius) {
  return 3.14 * radius * radius;
}
```

Figure 4.17. Magic number example.

Since it doesn't provide much context about the code that is being written, it will become more difficult to the beginner or other programmers, to understand the code, once some time has passed since it was written. It also becomes a problem when multiple occurrences of the same numeric value

are present in the codebase, since there will be a lot of places that will need to be refactored, if those values need to be changed.

The beginner does this kind of mistake a lot because, while being on the early stages of the programming learning process, readability and maintainability of the codebase are two concepts that don't receive a lot of attention by the student.

The process of catching such issue on the codebase was made possible by using one of the visitor API's, that is triggered every time that a literal value is found in a method's code. The second step was checking whether the literal value is a numeric value and if it occurs more than once in the method, and if so, it is considered a magic number and a new code quality issue is added to the tool list. This issue was classified as average, since it affects the readability of the code more than its maintainability, which means that its occurrence doesn't always lead into bugs.

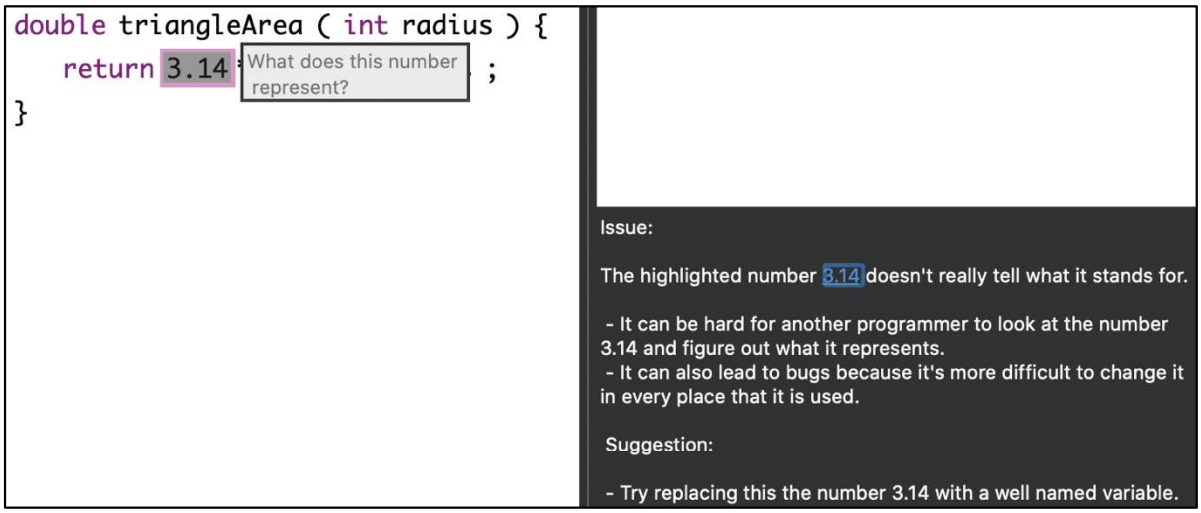


Figure 4.18. Magic number example.

It is explained to the user that this kind of issue affects the two aspects mentioned above, the readability and maintainability of the code, and suggested that such numeric values should be assigned to a constant with a well-formed name that provides the whoever reads the code with some context about what it represents. All of its occurrences should then be switched with the created constant.

4.2.4 Duplicate expressions in if/else

One of the most common bad practises found amongst inexperienced students, is the repetition of the same code statement or expression, in all the code blocks of an if/elseif condition. Most of the times, the beginner can't formulate the conditions in a way that only one condition, with the use of the logical operators, becomes enough, and with this being, ends up repeating one or more code

instructions, across different conditions. This represents a code quality issue, since duplicated code should always be avoided.

The recognition of this issue was made possible by using one of the APIs provided by the if condition visitor, which gives information regarding all the blocks of code present in a if/elseif condition. This way, it became possible to check if there were any expressions or statements that could be found across all the condition blocks, and if so, a new issue had been detected and was added to the code quality issues list.

4.3 Useless Execution

Code that doesn't impact the execution of a program in any way, is considered to be dead code. It is often found in students programs codebases, and this kind of issue may lead into the generation of bugs that can be hard to troubleshoot. It consists of code instructions which have no impact in the program execution, because they're values aren't used or their execution don't change any variable such as class attributes or arguments, which result in a block of code that doesn't change anything in the program.

4.3.1 Unused Function Return Value

Many times, a beginner is seen calling a function which has a return value, without using or assigning such value to a variable, and using that variable. This by itself isn't a problem, since a method can change object attributes, for example, but when the calling of such function doesn't change anything from outside its scope, the program execution won't be affected. It is then possible to argue that the function call can be considered useless to the program, since it doesn't actually alter anything.

```
int[] naturals(int n) {
    int[] array = new int[n];
    int i = 0;
    while (i < n) {
        array[i] = i + 1;
        i++;
    }
    return array;
}
```

```
int[] incrementNaturals(int[] n){
    naturals(n);
    int[] array = naturals(n);
    int i = 0;
    while (i < array.length) {
        array[i] = i + 1;
        i++;
    }
    return array;
}
```

Figure 4.19. Unused value example.

A student may end up with this kind of call due to thinking that the arguments which are provided to the function will get modified by it and have its values changed and forget that those arguments values will actually be copies of the original values. With this being, the arguments aren't the ones being modified by the function, but their generated copies are, and with this being, their values regarding the outside of the function call will remain the same.

Many beginners tend to miss this issue and end up with programs which contain such method calls, that are totally useless to the program and can cause bugs or undesired behaviours, since the logic behind the method should most likely make use of the values returned by such calls. It becomes relevant to warn programmers about this issue, since troubleshooting bugs caused by this kind of issue, can become very time-consuming.

Recognizing this issue involved one of the **visitor** API's, which is triggered every time that a method call is found on a method. When this happens, the method is analysed using the argument that the API provides, which contains information about the method that was called. The first check is to make sure that the method has return type and if so, another API that informs about whether the method causes changes outside of its scope, is used. If the method doesn't change anything outside of its scope, and it's return value is not assigned or used, then it can be concluded that its execution doesn't add anything to the program, and therefore, is considered useless. An example of such occurrence is showed below:

The explanation focuses of telling the beginner that the method's return value was not used, while the execution of such call doesn't alter anything outside of it, not even the arguments. This means that its call doesn't change anything in the execution of the program and therefore, is useless. It is suggested that the beginner decides whether it makes sense that the method doesn't change anything outside of its scope, like object attributes, etc… or to decide whether the method return type shouldn't be used.

4.3.2 Duplicate Function Call

This case is an extended version of the previous one, where the beginner also practises the duplication of the method call, with the same value arguments. This duplicate function call will have the same result value.

```
int[] getIncrementedNaturals(int n) {
    int[] array = naturals(n);
    incrementNaturals(array);
    incrementNaturals(array);
    return array;
}
```

Figure 4.20. Duplicate call example.

The recognition process was also an extended from the previous case, but with an additional check, that was made possible by using the Control Flow Graph. The added verification consists on checking the function call arguments for any kind of changes between the duplicate calls. The issue is only confirmed if the arguments preserve the same values between the function calls, since otherwise the method value would vary.

Such verification was made with the CFG feature that returns the possible paths between the duplicate calls corresponding nodes, with which becomes possible to verify if any of the arguments have been changed in those paths.

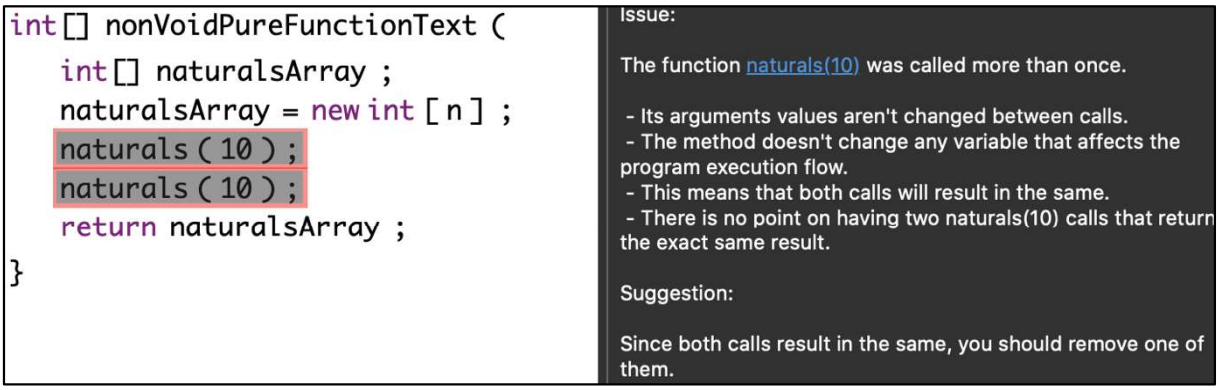


Figure 4.21. Duplicate call highlight and explanation.

4.4 Unreachable Code

A block of code is considered unreachable if it never, under no circumstances, gets executed by the program. There may a statement such as a *return*, or badly formed condition, such as a contradiction, that prevents the program from reaching some code actions.

4.4.1 Faulty Return Statement

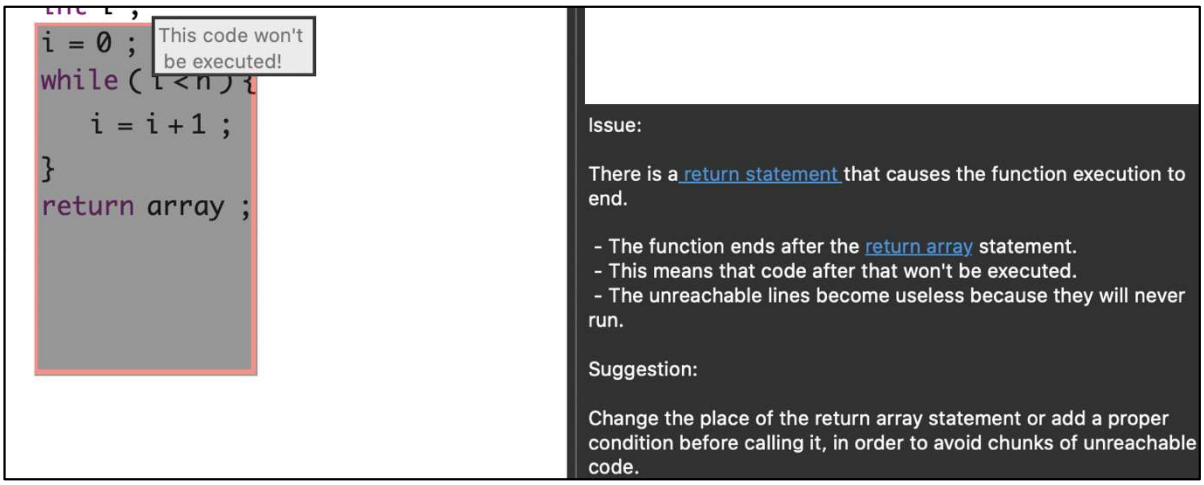
A return statement was considered to be faulty when it caused a method's execution to end, preventing code statements that where further down the flow from being executed.

This issue may be caused by the student not being totally aware about the meaning and functionality of the return statement, or by a distraction where the beginner forgot to remove the code instructions that came after the return statement. Either way, unreachable code lines can damage the program's performance, since it may cause bugs due to the non-execution of the unreachable code instructions. It can also affect the code readability, since the codebase will have code that is never ran, which can be considered useless and therefore, should be fixed or possibly removed.

```
int[] naturals(int n) {
   int[] array = new int[n];
   int i = 0;
   return array;
   while (i < n) {
         array[i] = i + 1;
         i++;
   }
   return array;
}
```

Figure 4.22. Unreachable code example.

This issue was recognized using the *deadNodes()* API from the method's Control Flow Graph. By using this API, it was possible to obtain the list of nodes which were found to isolated from the rest of the CFG nodes, which means that no node points to them. This means that the list contains the nodes



that represent the method's code instructions that are never executed. The first node of the list contains the *return* statement that caused the unreachable code issue, and with this being, it was possible to use this list to highlight the cause and the effect of this issue, which corresponds to the return statement and the following unreachable code instructions.

Figure 4.23. Unreachable code highlight.

4.4.2 Contradiction

Representing the exact opposite of the tautology issue (4.1.5), there are the conditions which are always evaluated as *false*. This kind of issue is the exact opposite of the contradiction one, and can be

```
boolean isFirstElement (int[] a, int b) {
   boolean isFirst = false;
   if (!isFirst) {
         return false;
   } else if (a[0] == b) {
         return true;
   }
}
```

Figure 4.24. Contradiction example.

be executed. The way to recognize this issue follows the same steps as the tautology one, but instead of comparing the variable values with the literal value **true**, it they are compared with the literal value **false**.

Since the condition, in the example above, will always be evaluated as *false*, the assignment will always be performed. This represents a tautology case. The explanation is the exact opposite of the tautology issue. Both these issue suggestions tell the student to evaluate if the condition is necessary at all, or to change so that it won't be always evaluated as *true*, or *false*.

5. Experiments & Results

In general, some of the issue types that are recognized by the tool, are also recognized by professional tools, such as SonarLint. But it is important to mention that there are also issues which were implemented, such as the duplicated condition guard, which are not included in the list of issues that are recognized by such professional tools.

Like it was mentioned, the bigger difference from the professional tools, and the focus of the pedagogical tool, is warn and supply the students with an explanation that helps them figure out why they committed the issue that they are being warned about, instead of giving them the quick fix, without focusing on the core of the problem, like most of the professional tools tend to do.

Upon the development of the pedagogical tool described in the previous chapter, we carried an evaluation under two perspectives. Mining existing code of students to investigate the occurrence of code quality issues, and usability experiments with the tool with end users (students).

5.1 Evaluation preparation steps

The first step was to gather some of the introductory programming projects from the last edition at our institution. All the data regarding the student identification was anonymized. After doing this, the next step was to associate the projects to their grades.

The projects were gathered with the intent of being submitted to the code quality checker tool, which contained the code quality issues detectors, used to find out whether the tool was able to to check if the projects contained such issues, or not.

But before submitting the projects to the code quality checker, they had to be submitted to another tool, that is part of the Paddle library used to build the code checker, which transformed them into the input object that the checker requires. The output of the parsing would be a batch of modules. Each of these represent the parsed value of a student project which was used has an input by the code quality checker. Then, the code issues are loaded and searched in the provided projects, by specifying the visitors described in the previous section.

5.3 Applying the tool to the parsed student projects

The linter object has an *analyse* function, which accepts a single project represented by a module, or a batch/list of modules. This will make use of the specified visitors and apply them to the modules that are supplied. The expected output is a list containing all the code quality issue that were found in the modules that were supplied. With this list of quality issues, it was possible to find out which issues

are present in the projects that are being examined. By having these results across all the projects, it becomes possible to find out the amount of times that each issue occurs per project, which are the most recurring issues and their location in the student's code.

The first test that was performed by following the previous described steps included projects of 153 students. After running the analysis on the batch of projects, we found **2077** code quality issues across all the 153 modules. Table X presents the distribution of issues distribution regarding the number of projects that contained at least an occurrence of an issue, and the total number of occurrences per issue.

Issues	# of projects	# of Issues
EMPTY SELECTION	24	39
MAGIC NUMBERS	151	1368
FAULTY RETURN BOOLEAN CHECK	50	68
FAULTY BOOLEAN CHECK	58	101
UNREACHABLE CODE	0	0
USELESS RETURN	64	169
DUPLICATE SELECTION GUARD	7	9
FAULTY METHOD CALL	76	266
NON-EXPLICIT ELSE	9	18
TAUTOLOGY	7	8
CONTRADICTION	4	5
DUPLICATE CODE	12	15
USELESS CODE	9	11
TOTAL	-	2077

Table 5.1. Presence of quality issues in student projects.

After gathering the results, the next step was to separate the projects by grade, since this kind of information was available, in order to try to understand if there would be any noticeable changes regarding the number of occurrences of certain types of issues, between the groups of projects, separated by the grade that they were assigned with. A project can be graded with an evaluation that varies between A and C (considering positive grades), with the grade A being the best and C the worst. As it can be seen in table 5.1, the number of issues tends to be larger for projects with better grades, since the average number of issues per case is bigger. This can be due to the students that had a better grade writing a more substantial amount of code, than the students with a lower grade, since the evaluation was based on the number of features that were included in the projects, which can be translated as more or less code. The distribution of the percentages of found issues regarding the grade of the project was:

Issue Type	A	B	C
EMPTY SELECTION	2%	1%	3%
MAGIC NUMBERS	65%	66%	70%
FAULTY RETURN BOOLEAN CHECK	4%	2%	3%
FAULTY BOOLEAN CHECK	5%	3%	6%
UNREACHABLE CODE	0%	0%	0%
USELESS RETURN	7%	12%	9%
DUPLICATE SELECTION GUARD	0%	1%	0%
FAULTY METHOD CALL	13%	13%	9%
NON-EXPLICIT ELSE	1%	0%	0%
TAUTOLOGY	1%	0%	0%
CONTRADICTION	0%	0%	0%
DUPLICATE CODE	1%	0%	0%
USELESS CODE	1%	0%	0%

Table 5.2. Percentage of found issues, per project grade.

The results shown above represent the percentages, regarding each of the issues types, of the total amount of issues that were found across all the projects with a certain grade. Like it can be see seen in table 5.2, across all the three possible grades, the issue type *magic number* was the one that was found the most, with a percentage ranging between 65 and 70 percent, across all grades of the projects that were analysed. Followed by the *faulty method call* issue, ranging between 9 and 13 percent, and the *useless return* issue, which percentage ranged between 7 and 12 percent.

Issue Type	10 to 13	14 to 16	17 to 20
EMPTY SELECTION	1%	2%	2%
MAGIC NUMBERS	69%	61%	69%
FAULTY RETURN BOOLEAN CHECK	2%	5%	4%
FAULTY BOOLEAN CHECK	5%	6%	3%
UNREACHABLE CODE	0%	0%	0%
USELESS RETURN	11%	10%	5%
DUPLICATE SELECTION GUARD	0%	1%	0%
FAULTY METHOD CALL	10%	11%	14%
NON-EXPLICIT ELSE	0%	1%	1%
TAUTOLOGY	1%	0%	0%
CONTRADICTION	0%	1%	0%
DUPLICATE CODE	0%	1%	1%
USELESS CODE	1%	0%	0%

Table 5.3. Percentage of found issues, per semester final grade.

This time, the percentage of found issues was calculated by separating the students not by project grades, but by final semester grades, since it was found to be a better indicative of how well a student knowledge and skills really progressed during the semester, since students could possibly help each

other during the projects development, which should have affected the quantity of issues that were found. The grades can spread from 0 to 20, with grades lower that 10 being meaning that the student failed the course, but for the sake of the experiment, these ones were not included. The grades above 10 were separated into 3 groups, 10 to 13, 14 to 16 and 17 to 20.

What was found was that the results don't vary that much from the previous distribution. Projects from students with lower and higher final grades seem to have pretty much the same issues occurrence percentage in almost all of the issue's types, with the exception of the *useless return* issue, where students with higher grades seem to be better aware of this bad practise and don't tend to perform it as much as the students with lower grades.

Another way to interpret these results, is by analysing the percentages regarding the total number of each issue, like it was done above, but also by the percentages of each issue found across all the projects. With this being said, the percentages are:

Issue Type	# of projects (%)
EMPTY SELECTION	16%
MAGIC NUMBERS	99%
FAULTY RETURN BOOLEAN CHECK	33%
FAULTY BOOLEAN CHECK	38%
UNREACHABLE CODE	0%
USELESS RETURN	42%
DUPLICATE SELECTION GUARD	5%
FAULTY METHOD CALL	50%
NON-EXPLICIT ELSE	6%
TAUTOLOGY	5%
CONTRADICTION	3%
DUPLICATE CODE	8%
USELESS CODE	6%

Table 5.4. Percentages of each issue found across all the projects.

The results from the table above offer a much more promising view, when it comes to the number of projects in which a certain issue type was found. By looking at the table, it is possible to confirm that almost all the issue types that were being caught by the tool, were found across various projects, from the all of projects that were analysed. This means that practically every project contained issues regarding code quality, which means that the students can be warned and advised, in order to fix those issues.

Regarding the spread of both the number of issues and the number of projects in which some issues were found, the results were expected to vary according to grade, for instance, the A-graded projects were expected to contain a smaller percentage of issues classified as *light*, and a bigger

percentage of issues classified as *serious*. But the contrary was found, that the percentages of issues, regarding all the issues types, are very similar across all the three grades, and are independent from the issue classification. One possible reason for this can be that the projects tend to be highly revised and advised (by instructors), before being submitted for evaluation. This means that the students may had already been advised by a professor, for example, on topics to improve in their codebases, and with that being, considerably less issues were expected to be found, in comparison with projects that did not go through such process, or with codebases from the introductory programming practical classes, where the kind of issues that are targeted by the tool have a much bigger chance of being found.

It is also important to mention the outliers were found, regarding both the total number of issues, found across all projects, and the number of projects that contain a certain issue type. In both cases, the outliers are the *magic numbers* and the *unreachable code* issues. The first one was found to be present in 99% of the projects, with this most likely being because of subject that the projects were focused on, which resolved around manipulating the colour of images. Since the codebase had a lot of methods that did this in various ways, there were found a lot of loose integers values, which represented the values of red, green and blue and with this being, the tool recognized a lot of magic numbers. On the other hand, the unreachable code issue wasn't found in any project at all. A possible reason for this is that since this issue, besides being considered a code quality issue, can also insert some bugs on the program and even originate compilation errors, which will be noticed by the students, since most IDE's won't let the program execute when this kind of errors are present in the codebase.

5.5 Usability tests with students

After finishing the experiment that was described above, another small experiment was performed. This second experiment involved some students that had to take a survey of 10 questions that contained code method examples, with the objective of pointing out if there were any issues related with the quality of the code that was displayed to them. The difficulty of finding these issues varied from question to question, depending on the type of the issue that was being targeted. Some questions didn't have issues at all, which were meant to make the experiment a little bit harder. The students had between 1 and 2 minutes to answer each question, and in case of need, were offered with help from either the person that was conducting the experiment or from interface with the tool that was developed.

This experiment only had 2 volunteers, which had already done the introductory programming course, and which were more experiment when it came to programming knowledge, which was a

downside, since the tool is meant to be used by students that are in the early stages of learning how to program.

As it was expected, the 2 students that answered to the survey, didn't face much difficulties when it came to figuring out which issues were present in each question, and only in very few cases, the tool had to be used in order to help the student figure out what was wrong. Regardless, the students still offered some feedback which helped improving the tool, such as suggestions that helped improve some of the explanations that were given for some of the issues that were highlighted during the tests.

6. Conclusions and Future work

Based on the research that was carried out, the development of the pedagogical tool and the results that were obtained, we conclude that:

- Professional quality checkers, in general, do not focus on pedagogical aspects, when it comes to the explanations for the code quality issues that they detect and highlight, focusing mainly in offering a quick fix.
- Using Control Flow Graph (CFG)s was the main structure what allowed the development of most of the code quality issues detectors, and that can be used to implement additional detectors.
- By using the Paddle library and the developed CFG, it became possible to detect some code quality issues which even professional tools do not detect.
- Students tend to include several code qualities issues in their code bases, during the entire course of introductory programming, not only the first few weeks.

The main focus of this thesis, was to build a tool that could successfully detect and highlight a list of chosen code quality issues, while providing a pedagogical explanation, so that it becomes possible for the student to realise and even understand why the detected issue is considered a bad practice. Hopefully, in this way, the student will learn to avoid such bad practices, resulting in better code bases, right from the beginning phase of learning how to program.

With this being said, after finishing the related work section of the thesis, it became possible to realize that current available tools for IDEs, such as Eclipse, which is the one that is used at our university, are not properly focused on the pedagogical side, instead, they provide little or no explanation on why the detected issue is a problem, and often only provide a quick-fix for the problem that was detected, resulting on the student not understanding why those issues represent problems for the codebase.

The CFG was a core artifact in the development of the pedagogical tool. Without it, most of the cases detection implementations would not be possible or would have become a lot harder to implement. Its development required a considerable implementation effort, but the final result served the exact purpose that it was meant to. Since the CFG structure provides a clear API, it can be easily having other use cases. With this being, we decided to add it to the Paddle framework. It will most likely become useful to future thesis, or projects, that shall be done by other colleagues, since it can be very useful with respect to static analysis.

The majority of code quality issues detectors that were implemented, are also detected by the professional tool SonarLint[22] that was mentioned in the related work section. We found that it was important to include those issues in the developed tool, with the company of a well put explanation, resulting in a highlight that was more pedagogical. On the other hand, with the developed CFG structure and the Paddle tool visitors, it became possible to implement a series of detectors, such as the *duplicated guard condition* one, which are not addressed by professional tools in their list of detectable issues. This means that the developed tool features some detections that are exclusive to it, as far as we investigated, while there are also several other of code quality issues which can be implemented. We have focused on the issues that may relate to common programming misconceptions that beginners have, rather than concerning with industry practices (e.g., code conventions).

After analyzing the gathered projects with our detectors, the results showed that there are still a lot of code quality issues that are found amongst a significant percentage of the projects. This means that some issues tend to stick with the student long after they start to learn how to program, since the projects are developed by the students in the final phase of the course. Therefore, the purpose of our pedagogical tool is justified, since it can help the students to become aware of such quality issues occurrences and help them to understand and eliminate those bad practises, right from the early stages.

Moving forward, the next steps would be to arrange a lot more interviews with students, that the ones that were performed on the end of the development stage of the pedagogical tool. With these interviews, it could be possible to understand if there are any flaws with the explanations that the pedagogical tool currently supplies.

After collecting all the feedback from the students, the next step would be to research and create a new list of quality issues detectors, and continue developing the tool, so that it could detect a wider range of code quality issues. After that, a second prototype would be developed, with the intent of being used by the students, during the practical classes, where most of the code quality issues occurrences are expected to be found.

References

[1] H. Keuning, B. Heeren, and J. Jeuring, 'How teachers would help students to improve their code', *Annu. Conf. Innov. Technol. Comput. Sci. Educ. ITiCSE*, no. 1, pp. 119–125, 2019.

[2] R. Pello, 'Design science research - a short summary', *Medium*, 2018.

[3] H. Keuning, B. Heeren, and J. Jeuring, 'Code quality issues in student programs', *Annu. Conf. Innov. Technol. Comput. Sci. Educ. ITiCSE*, vol. Part F1286, pp. 110–115, 2017.

[4] Y. Qian and J. Lehman, 'Students' misconceptions and other difficulties in introductory programming: A literature review', *ACM Trans. Comput. Educ.*, vol. 18, no. 1, pp. 1–24, 2017.

[5] J. M. Su and F. Y. Hsu, 'Building a Visualized Learning Tool to Facilitate the Concept Learning of Object-Oriented Programming', *Proc. - 2017 6th IIAI Int. Congr. Adv. Appl. Informatics, IIAI-AAI 2017*, pp. 516–520, 2017.

[6] P. Techapalokul and E. Tilevich, 'Novice Programmers and Software Quality: Trends and Implications', in *Proceedings - 30th IEEE Conference on Software Engineering Education and Training, CSEE and T 2017*, 2017, vol. January, 2017.

[7] E. Van Emden and L. Moonen, 'Java quality assurance by detecting code smells', *Proc. - Work. Conf. Reverse Eng. WCRE*, vol. January, 2002, pp. 97–106, 2002.

[8] J. Keung, Y. Xiao, Q. Mi, and V. C. S. Lee, 'BlueJ-UML: Learning Object-Oriented Programming Paradigm Using Interactive Programming Environment', *Proc. - 2018 Int. Symp. Educ. Technol. ISET 2018*, pp. 47–51, 2018.

[9] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, 'The Scratch Programming Language and Environment', *ACM Trans. Comput. Educ.*, vol. 10, no. 4, Nov. 2010.

[10] J. Trower and J. Gray, 'Blockly Language Creation and Applications: Visual Programming for Media Computation and Bluetooth Robotics Control', in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, 2015, p. 5.

[11] M. Kölling, 'The Greenfoot Programming Environment', *ACM Trans. Comput. Educ.*, vol. 10, no. 4, Nov. 2010.

[12] M. J. W. Lee, S. Pradhan, and B. Dalgarno, 'The Effectiveness of Screencasts and Cognitive Tools as Scaffolding for Novice Object-Oriented Programmers', *J. Inf. Technol. Educ. Res.*, vol. 7, pp. 061–080, 2008.

[13] M. Kölling, B. Quig, A. Patterson, and J. Rosenberg, 'The BlueJ System and its Pedagogy', *Comput. Sci. Educ.*, vol. 13, no. 4, pp. 249–268, 2003.

[14] SOUSA, Hugo Silva. 'Illustrating Debugger Execution Leveraging on Variable Roles'. Lisbon: ISCTE-IUL, 2016. Masters dissertation. Available: https://hdl.handle.net/10071/14632.

[15] A. L. Santos and H. S. Sousa, 'PandionJ: A Pedagogical Debugger Featuring Illustrations of Variable Tracing and Look-Ahead', in *Proceedings of the 17th Koli Calling International Conference on Computing Education Research*, 2017, pp. 195–196.

[16] A. L. Santos, 'Enhancing Visualizations in Pedagogical Debuggers by Leveraging on Code Analysis', in *Proceedings of the 18th Koli Calling International Conference on Computing Education Research*, 2018.

[17] S. Draxler, G. Stevens, and A. Boden, ‘Keeping the development environment up to date - A study of the situated practices of appropriating the eclipse IDE’, *IEEE Trans. Softw. Eng.*, vol. 40, no. 11, pp. 1061–1074, 2014.

[18] E. Foundation, ‘Eclipse Documentation’, 2019. .

[19] JetBrains, ‘Code Inspections’, 2019. [Online]. Available: https://www.jetbrains.com/help/idea/code-inspection.html. [Accessed: 04-Jan-2020].

[20] IBM, ‘Interactive Debugging with Node.js’, 2018. .

[21] M. Hentschel, R. Bubel, and R. Hähnle, ‘The Symbolic Execution Debugger (SED): a platform for interactive symbolic execution, debugging, verification and more’, *Int. J. Softw. Tools Technol. Transf.*, vol. 21, no. 5, pp. 485–513, 2019.

[22] A. Boroumand, ‘Java Code Linting with SonarLint’, 2018. [Online]. Available: https://www.codebyamir.com/blog/java-code-linting-with-sonarlint.

[23] A. Dangel, ‘Code quality assurance with PMD – An extensible static code analyser for Java and other languages.’, 2018. [Online]. Available: https://www.datarespons.com/code-quality-assurance-with-pmd/.

[24] A. Santos, ‘Paddle’. Available: https://github.com/andre-santos-pt/paddle.