POLITECNICO DI TORINO

Master’s Degree in Computer Engineering

Master’s Degree Thesis

# Programming assignments evaluation using technical debt

Supervisors

Prof. Marco TORCHIANO

Prof. Antonio VETRÒ

Candidate

Alessio MASON

December 2023

# Abstract

In recent years, the idea of technical debt is gaining adoption to describe the problem introduced - more or less consciously - by imperfections in code in an attempt to reduce time and costs of software development.

The goal of this thesis is to investigate whether technical debt, a concept usually applied to large industrial software, can be applied to the software developed by students in their programming assignments.

The idea could be useful both for students and for teachers.
Students could find the aid of an automated analysis of their programs helpful to understand which topics they still have not understood completely and what mistakes or oversights they incurred during their preparation.
For teachers, on the other hand, analysing the code produced by students might be useful at two different times: throughout the course, to understand whether some topics are still a bit obscure to a number of students and might require a revision. Then, following the final evaluation, technical debt analysis may be useful to understand the overall comprehension of the most difficult concepts or even to award the students who wrote not only working but also clean and maintainable code, which is a skill that is not usually considered in early-on courses but that most definitely has to be picked up before entering the labour market.

Specifically, this paper examines a known context, the *Object-oriented programming* course for the Computer Engineering Bachelor's degree of Politecnico di Torino: real past projects have been analysed verifying which problems arise as the commonest amongst the students, whether they make sense in the context of students' initial approaches to programming and their possible correlation with the final evaluation received.

This study did prove useful to understand the repeating issues amongst the students and to work out a way to automate said analysis, so that, in the future, it could be carried out even during the course to better comprehend the students' preparation.

# Acknowledgements

As I stand here writing these lines that mark the completion not only of this thesis work but also of five years of intense studies, I thank all the people that helped me and encouraged me along this journey.

To my parents, who supported me all the way along and that always push me to strive for more.

To my brother, who inspires me every day with the passion he puts into everything he does.

To all of my friends, for being there for me and for making me understand what the next chapters of this journey might look like.

To all of you I extend my heartfelt gratitude.

# Table of Contents

# List of Tables

# List of Figures

# Acronyms

| | |
|---|---|
| **CFG** | Control-flow graph |
| **CI/CD** | Continuous Integration/Continuous Development |
| **CoC** | Cost of Change |
| **E2E** | End-to-end (testing) |
| **POM** | Project Object Model |
| **TDD** | Test-driven development |
| **UUID** | Universally Unique Identifier |

# Chapter 1

# Introduction

In recent years, technical debt has emerged as a new concept in software engineering. The idea, first formulated by Ward Cunningham in 1992, is that every bug or not properly written section of code represents a debt. "Every minute spent on not-quite-right code counts as interest on that debt", Cunningham postulates.

In fact, every new feature or even modification operated while not addressing the most problematic parts of the code will eventually have to be repaid: fixing the same bug, as trivial as it may be, might take much more time if, with the passing of time, new features, dependencies and, in general, assumptions have been made upon that piece of software.

These imperfections in code are not always intentional: they can be caused by lack of time during development (including the testing phase) or by intentional choices, either lacking proper consideration or even after well-rounded ones but based on a context that changed throughout time.

A few examples of these imperfections that constitute technical debt are bad organizations of classes, known defects of the code never fixed, requirements implemented only partially, code not dutifully tested or whose coverage is insufficient.

Technical debt is not at all an infrequent matter: what happens even in the most efficient of work environments with the best possible code produced is that, periodically, some amount of time has to be allotted to "repay the debt", refactoring already-existing code, avoiding even more cumbersome modifications down the line. The idea of technical debt has proven effective in reducing unforeseen changes and modifications during development and has been growingly adopted in recent years.

Several softwares and solutions have then emerged to automate and facilitate the identification of technical debt: these softwares perform different kinds of analyses that highlight the presence of pieces of code that might constitute a problem (the debt) and even provide and estimation of the time needed to solve the issues (the "sum" needed to repay the debt).

These kinds of softwares, though, are usually applied in medium-to-large companies to manage the development process: the goal of this thesis is to understand whether they can be applied to analyse code produced by students in their assignments or final evaluations.

Typically, throughout their career, students produce small projects aimed at solving only a handful of requirements, usually avoiding testing and other phases of development. Furthermore, focusing on the first few years of learning, as this paper does, means that a lot of errors that might be found in the code might come just out of inexperience of the students, not even knowing that some practices might represent a bad behaviour in software development.

The aim of this research, in any case, is to explore whether applying the concepts of technical debt and automated code analysis to these projects could reveal macroscopic errors not highlighted by the assignments evaluations, but nevertheless important to be understood and, in future, possibly avoided by the students.

This could represent an advantage both for students and for teachers. Students, in the first place, by having their assignments analysed might recognise that they lack preparation or studying on some specific topics, that some concepts still have to be ironed out. Automated analysis could very well represent a valid aid in studying and more importantly understanding the mistakes or the oversights made during the preparation.
For teachers, on the other hand, analysing the code produced by students might be useful at two different times: initially, throughout the course, analysing the assignments as they are being handed in, to understand whether some topics are still a bit obscure to a number of students and might require a revision. Then, following the final evaluation, technical debt analysis may be useful both to understand the overall comprehension of the most difficult concepts and even to award the students who wrote not only working but also clean and maintainable code, which is often a skill that might not be considered in early-on courses but that most definitely has to be picked up before entering the labour market.

Specifically, this paper will examine a known context, the *Object-oriented programming* course for the Computer Engineering Bachelor's degree of Politecnico di Torino: this course represents the first approach for students to object-oriented programming and, also, to automated testing and evaluation. All programming assignments throughout the semester and, afterwards, the final exam are evaluated against a batch of tests, written by the evaluators.

Given that all the students work on the same assignments composed of the same requirements, applying an automated analysis to the code might underline some common critical issues not fully understood by the students or, in the case of the final evaluation, might incentivize the production of better-quality code.

The thesis will first explore the concepts of software quality, technical debt and automated software analysis, portraying the existing context in which these technologies are usually spent.
Subsequently, the whole research process will be described, including all the tentative solutions approached, the softwares considered for the purpose and the ones eventually chosen.
Finally, the results of the analysis conducted on the projects produced by the students for a past exam will be laid out, verifying, for this specific case, which problems arise as the commonest amongst the students, whether they make sense in the context of students' initial approaches to programming and their possible correlation with the final evaluation received.
In the end, a general picture will be composed, examining which further developments might be suitable for this thesis and whether the application of technical debt to students' assignments is a viable idea, in which way and with what implications.

# Chapter 2

# Context and background

This chapter is intended at understanding the general context that serves as a starting point for this thesis and how the goal of this research came to be.

## 2.1 Software quality

Back when software development was in its infancy, software quality was not even conceived as a concept. Software was developed and bugs were squashed out as they were identified.

With the passing of time, this approach was no longer sufficient. In the '70s, software could no longer be considered "good quality" if it "just worked": all requirements had to be met. This meant developing tests to ensure that such requirements were fulfilled. However, this created long delays, aversion to change and a strong dependency between software development and testing [1].

In recent decades, software quality has again evolved as a concept: passing an exhaustive series of tests is still a necessary but no longer a sufficient condition. Nowadays, "quality is a development issue, not a testing issue" [2]. This means that beyond ensuring that the produced software adheres to all functional requirements, offering all the functions and performing all the tasks it has been designed for, said software has to be certified to conform to non-functional requirements too.

These non-functional requirements, as a general rule, stipulate *how* the system should behave or even *how* the code should be written and organized. Although not exhaustively, ten common non-functional requirements can be identified [3]:

- Scalability: it refers to the systems' ability to perform and operate as the number of users or requests increases;

- Availability: it is measured as a percentage of uptime and defines the proportion of time that a system is functional and working;
- Extensibility: it measures the ability and feasibility to extend a system, either by adding new functionalities or modifying existing ones, and the effort required to implement the extension;
- Consistency: it guarantees that every read returns the most recent write. This means that after an operation executes, the data is consistent across all the nodes, and thus all clients see the same data at the same time;
- Resiliency: it measures the ability to resist, handle and recover from accidental and malicious failures;
- Usability: it gauges the system's capacity to enable users to perform tasks safely, effectively and efficiently, offering a comprehensible and accessible experience;
- Observability: it is the ability to collect data about the program execution, the modules' internal states and the communication between components, for example by logging, monitoring and raising alerts;
- Security: it highlights the degree to which the software protects information and data so that people, other products or systems have access to data appropriate to their types and levels of authorization;
- Durability: it relates to the software's ability to meet users' needs for a relatively long time;
- Agility: it enables developers to respond to changes quickly; it is characterized by the maintainability (testability and ease of development), the deployability (the time it takes to get code into production), the configurability and the compatibility with other products or designs of the software.

In the latest years this change in perspective has caused a paradigm shift in how the development process is organized. Firstly, Test-Driven Development (TDD) has been introduced as a concept, including unit testing (testing the single feature), integration testing (testing the integration of different software layers) and end-to-end (E2E) testing (testing the whole process). Its principle is to "write the test first" [4]: a single unit test is written, which fails because the functionality is not implemented yet. Then "just enough" code is written to make the test pass, and then the process is repeated for every functionality needed, accumulating unit tests along the way [5].
But that is not all: versions checking is fostered, to make sure that the latest

versions of runtime environments, frameworks, and libraries are used, and Continuous Integration/Continuous Development (CI/CD) means that new features are integrated in the main production codebase as they are finished, avoiding discovering conflicts and incompatibilities only at the end of development [6].

Aggregating these new concepts in an organic form, Agile software development has been proposed as a new way of organizing work. First proposed in the 2001 *Manifesto for Agile Software Development* [7], these new Agile practices promote a new way of working, different from the traditional approach.

The traditional approach, usually called *waterfall*, consists in dividing software development into separate subsequent phases: design, development, testing, deployment and maintenance. This usually leads to several problems, as this model does not cope well with change, leads to unpredictable software quality, as testing is only performed at the end, and if any problems arise a lot of restructuring work is often needed [8].

The Agile Manifesto proposes a leaner solution that aims at delivering working software frequently, favours the collaboration between developers and stakeholders and welcomes change, even late in the process. This agility is also researched through continuous attention to good design, avoiding slack solutions that might be faster initially but that might cost time and resources down the line: this is where the concept of technical debt is introduced, which will be described later in this chapter.

Specifically, the Agile Manifesto values [7]:

- individuals and interactions over processes and tools;
- working software over comprehensive documentation;
- customer collaboration over contract negotiation;
- responding to change over following a plan.

Adopting the Agile approach has led companies to achieve up to 30 percent increases in productivity and implementation speed while simultaneously reducing residual defects at time of release by over 70 percent [9].

The Agile methodology is not the only possible solution for a process that fosters software quality: one of its main applications, for example, is the Extreme Programming methodology, whose main principles are reviewing the code intensively (mainly through pair programming), performing continuous integration and testing, iterating quickly and considering only the minimum workable requirements, to limit the scope of the work. In fact, the name comes from the adage "what is good, do to the extreme" [10].

Albeit gaining popularity in recent years, Agile, Extreme Programming or similar methodologies do not represent the only solution to write good-quality software. On the contrary, companies and developers usually mix the approaches, accommodating their own needs [11].

The common line, though, is the search for greater software quality. What is nowadays important is not only *what* the software does, but also *how* it does it and *how* it is organized. As previously mentioned, this required several changes throughout the years in how work is organized, but these changes were quickened by a couple of reasons.

### 2.1.1 Possible effects of poor software quality

Two main motivations propelled the search for better-quality software: risk and cost management.

#### Risk management

Software failures have caused many different problems over the decades, ranging from human fatalities to simple annoyances that nonetheless have to be taken care of. Many examples can be cited, from crashes of perfectly-crafted planes caused by mere software bugs [12] [13], to simply customers of smart thermostats left in the cold [14] or to hazardous rockets explosions [15] [16]. These of course are negative events for a company that in many cases can be prevented by pursuing high software quality.

#### Cost management

In addition and often aside the concrete dangers, costs risks take a prominent role in underlying poor software quality. Good software costs less to maintain or to extend in functionality, whereas bad software inevitably leads to rewrites and reworks, paralyzing the development process, as programmers have to work on past functionalities to be fixed and cannot continue with the planned schedule, accumulating backlog. Even worse, though rarer, are business-disrupting events, such as trading firms almost going bankrupt due to a computer glitch [17] or car makers having to issue recalls due to malfunctioning software in their vehicles [18].

### 2.1.2 Causes of poor software quality

As previously discussed, many tools and mechanisms exist today to improve software quality. Nevertheless, the results are, often, still far from being satisfactory, as mentioned. Why is that?

Many reasons coexist: one possible answer, tough, is that, despite vast improvements in recent years, the spotlight is nowadays still on *coding* and not on the entire software engineering process. This focus, though, concentrates on the wrong aspects, because programmers spend more time designing, testing, maintaining code or generally managing the software operations than they spend actually coding new functionalities in [19] [20].

The consequences of this general thinking are rather practical: sometimes quick prototypes are made, writing code that will either have to be thrown away, being too expensive to fix, or at least heavily refactored. The common misconception is that producing working code quickly speeds up the development process, whereas it will actually slow it down in the future [21].
Often not enough time is dedicated to testing, again prioritizing writing new code, accumulating the debt of testing towards the later stages of the software flow [22]. On the other hand, another common misconception is that a piece of code that passes all tests is definitely correct: testing usually takes out the most evident bugs, but not always subtle timing bugs and edge cases are found.
Finally, lack of time diminishes software quality, or better lack of planning and prioritization of the wrong aspects: peer review is often disregarded and fixed dates and requirements do not concede the needed flexibility for software to be improved [21].

All these issues contribute to worsen software quality and can all be summarized by a common metaphor that can be used to describe them: the technical debt.

## 2.2 Technical debt

As previously discussed, often in software development quite some time is spent on fixing bugs or problems, time that could have been better spent had the design of the code been done properly in the first place and software quality prioritized. A new concept has been formulated in recent years that perfectly sums up this pattern and habit: the technical debt.

This metaphor was first proposed by Ward Cunningham in 1992, describing the work he had conducted with his team on the WyCash project, the first commercial software developed and maintained using all the Extreme Programming principles [23], trying to organize their work according to this philosophy [24].

By this metaphor, every time some code is produced, some form of debt is signed. This may be caused by bugs or imperfections introduced in the code and still not recognized, meaning that they will have to be fixed at a later time, but more generally the debt represents the developers' incomplete understanding of the features being developed.

This means that time spent on not-quite-right code counts as interest towards that debt: by continuing to build new functionalities upon the existing code those previously introduced imperfections will become harder to address and improve, and continuing to code without refactoring, i.e. without reorganizing the program to reflect the newly acquired understanding of the features (acquired by developing them) only means that the program will eventually contain no understanding of the matter and improvements will take longer and longer, or, worse, will be practically impossible [25].

The way to repay even just part of the debt is to refactor the code, again not only to address the imperfections but mainly to reflect one's current understanding of the problem, "putting the learning back into the program" [25].
Cunningham suggests that the best time to refactor a program is right before you extend it, "making a place for a feature before I make the feature" [26].

Jim Highsmith even defines the Cost of Change (CoC), which of course increases with the passing of time and with more technical debt present. He suggests that even doing nothing makes matters worse, only incremental refactoring can eventually reduce the Cost of Change [27].

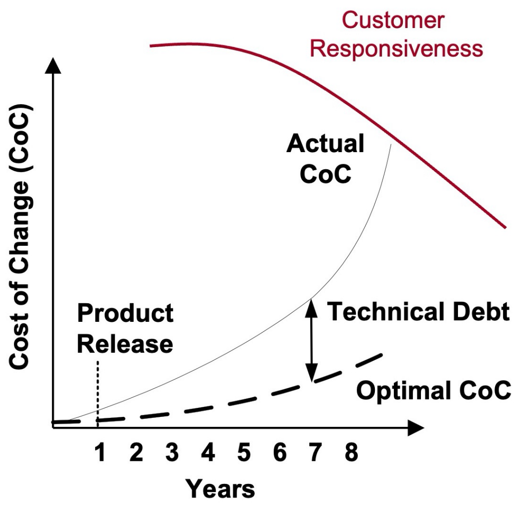


**Figure 2.1:** Cost of Change [27]

Cunningham further elaborates that the whole debt metaphor and the ability to pay back debt depend "upon your writing code that is clean enough to be able to refactor as you come to understand your problem" [25]. That is, the programmer cannot be asked to foresee the future, as it usually even "does the future a disservice by trying" [26] (making assumptions that, in any case, will have to be changed down the road), but it is paramount to write organized code from the outset, in order to be able to modify it at a later time, and to welcome change from the beginning, even making it a part of the schedule and accounting for it.

In 2001, Cunningham reinstated the metaphor [28] by postulating that:

- Skipping design is like borrowing money;
- Refactoring is like repaying principal (in financial terms, the principal is the initial amount of money that is borrowed in a loan);
- Slower development due to complexity is like paying interest.

Of course creating technical debt during development presents both advantages and disadvantages.
On the one hand, it allows to release on time and within budget, to produce more code and to reach a working state in a quicker way. It is beneficial for stakeholders too, as they have the opportunity to get back to developers in less time with feedback and opinions.
On the other hand, technical debt implies increased maintenance costs as the project grows, inferior flexibility of the code and, overall, worse productivity.

### 2.2.1 Types of technical debt

Technical debt can be divided into different types, depending on its origin [29]:

- Code debt: imperfections or bugs in code, inconsistent coding style;
- Design debt: violations of design rules, e.g. bad organization of classes;
- Test debt: lack of tests, inadequate test coverage, and improper test design;
- Documentation debt: incomplete or outdated documentation.

They all concur to increase the general debt but, depending on the context and the specific project, one might be worse than the other. In general, technical debt is usually composed of all of them.

### 2.2.2 Causes of technical debt

Many different reasons determine the growth of technical debt for a specific project. With the main one remaining the lack of understanding of the problem, as per the original metaphor formulated by Ward Cunningham, literature has in time identified other possible causes of technical debt [29]:

- Schedule pressure: working under pressure inevitably leads to hasty decisions; programmers tend to adopt shortcuts or to make code "that just works", creating problems such as code duplication (it is easier to copy and paste than to refactor to reuse what could be common code). This approach means that the product will be released on time but will be riddled with bugs, imperfections or will be plagued by technical debt;

- Lack of skilled designers: designers that do not understand the fundamental design principles or the problem at hand will definitely take bad decisions, either planning, designing or reviewing decisions that will lead to poor software quality;
- Lack of application of design principles: as per Ward Cunningham, failing to apply sound design principles tends to create code that is then quite difficult to extend or modify. Sometimes this is not even the designers fault: it often happens when relieving an old codebase that has outdated and legacy code;
- Lack of awareness of design imperfections and refactoring: programmers are often unaware of the imperfections introduced in the code; by not addressing them, deferring refactoring, the technical debt accumulates over time.

As previously mentioned, it is often acceptable to create a certain amount of technical debt, either to prototype a certain functionality or to gain a better understanding of the requirements, but always with the thought in mind of addressing it as soon as possible.

## 2.3 Identifying the technical debt

Given that sometimes it is somewhat difficult to even recognize the presence of technical debt in a codebase, different kinds of methods and analysis have been formulated to identify debt and to estimate the size and time needed to address it.

This kind of analyses are broadly divided into two main categories: dynamic analyses and static analyses, the main difference being whether the actual code is run or not in order to be analysed. These are not alternatives but tend to be complementary approaches.

### 2.3.1 Dynamic program analysis

The commonest and most typical kind of analysis, dynamic analysis necessitates for the code to run in order to analyse it.
This type of analysis is more input-centric [30], is best suited to handle runtime programming language features like polymorphism, dynamic binding, threads, etc. and usually incurs larger run-time overheads than static analysis [31].

The main modality of analysing the software is through the execution of a series of test cases. Tests can be unit tests (testing the single feature), integration tests (testing the integration of different software layers) and end-to-end (E2E) and acceptance tests (testing the whole process) [32].
Subsequently, the code coverage is usually computed, to understand whether the tests are in fact exhaustive and examine all parts of the program.

Another modality is monitoring the program during its normal execution. This can be done in various ways, either by debugging, logging or monitoring resource usage, events, and interactions.

Other types of dynamic analysis are program slicing (reducing the program to the minimum form that still produces the selected behaviour, to better identify what causes a problem) [33], security analyses and fuzzing (executing a program on a wide variety of inputs, often randomly generated) [34].

### 2.3.2 Static program analysis

Static analysis differs from the dynamic one, as the code does not have to be run in order to be analysed.

Static analysis works by creating a model of the state of the program and then determining how the program reacts to that state. Given that a program execution can carry out in many different ways, the analysis has to keep track of all the possible states. This is of course not feasible, as even every possible user input or interaction would need to be considered, so a simplification has to be performed in order to obtain a workable model: this means that the analysis will be somewhat less precise than a dynamic one, but will probably be able to provide different and complementary results [31] [35].

Different kinds of static program analyses can be performed.

#### Data-flow analysis

This kind of analysis focuses on the values data assume during the execution of the program and how they evolve: specifically, the goal is to verify whether "tainted" data (that is, user-supplied data which is still unchecked) is sanitized before being used [36] [37].
This technique is often used by compilers to optimize the code.

#### Control-flow analysis

The purpose of the Control-flow analysis is to determine what "elementary blocks" (functions or pieces of code in general) may lead to what other "elementary blocks".

The information is collected and represented in a control-flow graph (CFG). This is yet another technique used by compilers to optimize the code [36].

#### Abstract interpretation

The theory of Abstract interpretation is a general methodology for calculating analyses: it allows the extraction of information about a possible execution of a program without actually executing the program. It can be used by compilers to look for possible optimizations or for certifying a program against certain classes of bugs [36] [38].

#### Types systems

A type system is a logical system that defines a set of rules that can be applied as a property to all elements of a language, such as variables, constants, functions or expressions. This allows to perform a static analysis that ensures that such requirements, defined by the programmer, are met. This practice is usually called *type checking* and is useful to prevent errors, to avoid unwanted behaviours, to improve the efficiency (thanks to optimizations performed by the compiler) or to even just better explicit the designer's intentions [39].

#### Model checking

Software model checking is the algorithmic analysis of programs to prove properties of their executions. It traces its roots back to logic and theorem proving, providing ways to check if a model (which in this context means a formal model of a piece of code, though in other contexts it could be a model of a piece of hardware) complies with a given specification [40].

### 2.3.3 Dynamic and static program analysis: the synergy

As mentioned, dynamic and static analyses are not alternatives, but they are often complementary. The two approaches can be applied to a single problem, producing results that are useful in different contexts. For instance, both are used for program verification. Static analysis is typically used for proofs of correctness, type safety, or other properties. Dynamic analysis demonstrates the presence (not the absence) of errors and increases confidence in a system.
Quite often performing one analysis first, then the other (and perhaps iterating) is more powerful than performing either one in isolation [35].

## 2.4 The results of the analyses: bugs and code smells

These types of analyses, as discussed, provide varied and different kinds of results. Static program analysis, specifically, provides one kind of result that is worth exploring, as it will be the center of this thesis: the detection of bugs and code smells.

### 2.4.1 Bugs

A bug, famously, is an anomaly in the code that causes a malfunction of the program. A typical example is a mathematical operation where the divisor is zero (typically, this happens when it is the user to provide, even indirectly, the divisor and the input is not thoroughly checked): as the operation is clearly impossible, the program will crash.

More in general, though, only the worst bugs actually cause the program to crash: more often the effects will be more subtle, harder to recognize and perhaps even manifesting themselves in seemingly unrelated parts of the program.

A bug might not even cause any visible effect most of times: this is why static program analysis is useful to recognize in advance such bugs, vastly reducing time spent debugging or investigating the bug.

### 2.4.2 Code smells

A different kind of imperfection that can be found in code is a code smell. It should be noted, initially, that a code smell is not a bug: in fact, code smells do not cause anomalies during the execution of the program and, even if not addressed, they are for the most part harmless.

They are, although, quintessential in the improvement of software quality, as they are strictly related to the concept of technical debt. Even the name, in fact, suggests it: first coined by Kent Beck, it evokes the idea that "intricate code doesn't smell right" [41] [42].

A code smell, ultimately, represents an imperfection in the code, indicates poor software quality (in that specific part of the codebase, at least) and suggests the possibility of refactoring [43].

Criteria for recognizing a code smell are not set in stone and human intuition is always greatly valued, but a few broad examples can be provided [43]:

- Duplicated code: code that has been copied and pasted; it could be extracted, generalized and unified to improve readability and flexibility (in case of future modifications);
- Long method: "the longer a procedure is, the more difficult it is to understand"; dividing long methods in smaller ones improves encapsulation (each piece of code does one specific thing) and readability (the method name itself acts a a comment and provides an explanation of what the code does);
- Large class: usually a combination of long methods and duplicated code; can be refactored for better clarity;
- Long parameter list: long lists of parameters in methods signatures that are difficult to handle; they can be replaced with method calls, substituted with a parameter object or the whole method could be refactored to reference the attributes of its own object (in the case of object-oriented programming);
- Speculative generality: first suggested by Brian Foote, this happens when the code is generalized beyond the current needs, covering edge cases and special needs that are useless at the moment and that only concur at rendering the code more intricate and harder to maintain.

These are only a few examples of many imperfections that can be classified as code smells: it is evident that none of these anomalies cause any misbehaviour in the program, but they increase the technical debt, worsening the quality of the codebase and causing it to become more and more difficult to manage.

## 2.5 Putting it all together: applying technical debt to programming assignments evaluation

As already cited, program analysis is a valid tool for recognizing the presence of technical debt in a codebase. This kind of analysis is usually employed in software development companies [44]; the idea of this thesis, though, is to apply these same methods to the evaluation of students' programming assignments.

The idea, of course, is not to identify technical debt in itself, but to investigate whether this concept and program analysis can be helpful for both students and teachers to understand common mistakes and imperfections.

Evidently, bugs need to be addressed in order to obtain a fully-working program; it could be argued, however, that it could be premature to highlight code smells in students' assignments. The goal of the thesis, nonetheless, is to understand whether even code smells can be useful in recognizing where the students could improve: subsequently, perhaps, the list of rules to check the code against could be reduced, if certain ones were to be judged too advanced or irrelevant.

The following chapters will describe how this idea was put to the test and, eventually, what findings were gathered in the process.

# Chapter 3

# Analysis methodology

After reviewing, in the previous chapter, the context from which this thesis originated, this chapter will proceed to describe the methodologies adopted to conduct the work, which was the starting point, what choices were made and what softwares were selected to be used.
The results of the work will be reviewed in the following chapters.

## 3.1 The starting point

The goal of this thesis was to apply the concept of technical debt to students' programming assignments: specifically, the *Object-oriented programming* course for the Computer Engineering Bachelor's degree of Politecnico di Torino was chosen as the starting point.

This course is for many reasons peculiar in the career of this degree [45] [46].
Firstly, this is the first course that introduces not only object-oriented programming, but also the first concepts of software engineering to the students.
It is also the first course to provide automated evaluations, both for assignments throughout the course and for the final exam.

Specifically, the course presents several programming assignments during the course, characterized by the concepts of the Java language introduced in the course up to that point. The assignment is composed of a few different requirements and is accompanied by a small example class that tests the main methods or functionalities required and that is useful for clarifying doubts regarding requirements.
After the deadline a test suite is added to the Continuous Integration/Continuous Development pipeline, ensuring that all tests are run in the same environment.

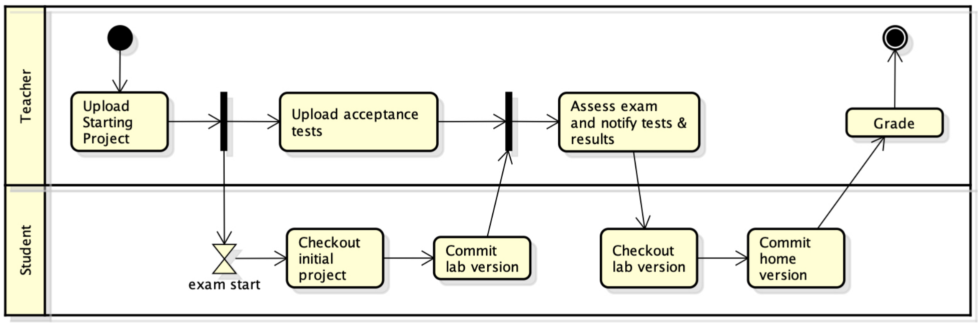


**Figure 3.1:** Exam procedure [45]

The final exam works in a similar fashion: the teachers prepare an initial project and upload it to the student's repository. During the exam, then, the student is again provided with a requirement document and a small example class, as the acceptance tests are made available only after the deadline. In the following days the students have to fix or complete the program in order to make it pass all the tests in the suite: the exam is then graded considering the amount of tests passed by the *lab* version and the amount of modifications needed to complete the fully-working *home* version (and the answers to a few theory questions).

Both the assignments and the final exam are tested through a CI/CD pipeline. All students' repositories are hosted on a private GitLab instance, thus the pipeline is configured in a *gitlab-ci.yml* file [appendix A.1]. This file describes the steps to be taken when executing the pipeline: in this initial version, only one stage of the process is present, in which the program is compiled and tests are run.

All students' projects are written in Java and are created as Maven projects. Apache Maven is a build automation tool that simplifies the building process of (primarily) Java projects [47]. All projects are described by a Maven's Project Object Model (POM) [appendix A.1], an XML representation of the project which describes the project's configuration and plugins and that manages the project build phase [48].

## 3.2 The initial goal

Dynamic program analysis is what is currently used by both students and teachers in the context of this exam: students likely use debugging to complete their assignments, whereas teachers use automated testing to assess the work after students hand it in.

The initial goal was to match a static analysis phase to the dynamic one, both for teachers and for students.

Specifically, by applying the concept of technical debt students would be able to get an evaluation of their work as they progress, listing peculiar bugs or code smells highlighting where they need to improve their assignment or even their understanding of the issue altogether.
Teachers, on the other hand, could use a static analysis report to comprehend where their students' knowledge is weaker and what could be useful to revise in class.

The initial goal was to understand how to perform a static program analysis in the already-defined context of the *Object-oriented programming* course, to assess which softwares would be better suited for the task and to evaluate whether a full analysis report would be appropriate for a student.

This was to be done with two modalities: an individual analysis for the student, which eventually might be integrated in the GitLab pipeline after the testing phase, and a massive analysis for teachers, to analyse an assignment as a whole.

## 3.3 Initial orientation analysis

The first step was to understand which software to use to perform the static program analysis: many alternatives were considered, but both due to previous personal experience with it and for the valid documentation available for the integration with Maven SonarQube was chosen.

SonarQube is a software developed by SonarSource that performs code inspections to highlight bugs and code smells in code [49].

Working on the individual analysis necessity, after setting up a local instance of SonarQube as a Docker container, the GitLab CI/CD pipeline [appendix A.2] was modified to include a second stage: the first stage remained the testing one, where the code is compiled and tests are run. If this stage is successful, the *codequality* stage is started, where a SonarQube analysis is initiated and results are saved in the (currently local) instance of SonarQube. The POM file was also modified [appendix A.2] to include the SonarScanner for Maven plugin, necessary to run the analysis.

To test the new stage the *gitlab-ci-local* tool [50] was used: this allowed to avoid pushing a new commit for every test by running them locally.

### 3.3.1 Initial analysis on personal projects

An initial analysis on past personal projects was conducted. Specifically, having attended the *Object-oriented programming* in 2020, all programming assignments published during that course, an exam example and two exams were analyzed.

| | # of lines | Bugs | Code smells | | | | Duplication | Passed tests |
|---|---|---|---|---|---|---|---|---|
| | | | Total | Critical | Major | Minor | | |
| **LAB01 University** | 87 | 3 (D) | 42 | 0 | 22 | 20 | 0% | 13/17 |
| **LAB02 Hydraulics** | 87 | 0 (A) | 18 | 1 | 7 | 10 | 2.3% | 38/38 |
| **LAB03 Diet** | 76 | 4 (C) | 10 | 0 | 4 | 6 | 5.7% | 54/54 |
| **LAB04 Mountain Huts** | 343 | 0 (A) | 3 | 1 | 0 | 2 | 0% | 1/1 |
| **LAB05 Clinic** | 398 | 0 (A) | 0 | 0 | 0 | 0 | 0% | 1/1 |
| **EXAMDEMO Milliways** | 437 | 0 (A) | 1 | 0 | 0 | 1 | 0% | 1/1 |
| **EXAM_2020-06-23 Sports** | 78 | 0 (A) | 2 | 0 | 1 | 1 | 0% | 44/44 |
| **EXAM_2020-07-07_Delivero** | 79 | 0 (A) | 0 | 0 | 0 | 0 | 0% | 50/50 |

**Table 3.1:** Results for initial analysis on personal projects

It can be noted that bugs or code smells can only be found in the first assignments: this is probably due to the structure of the course. In fact, these assignments were modified after the publication of the acceptance tests, therefore some bugs or code smells might have been fixed in the process.

This is not what would normally happen if this process was to be applied to a student's assignment: the student might find this kind of analysis useful even during the coding process. It was impossible, however, to recover prior versions of these assignments resolutions, so this issue has to be taken into account.

The counterexample is the first assignment, "LAB01 University", which is the only one that was not fully fixed after the publication of the acceptance tests (hence not fully passing the tests in this analysis) and is also the one presenting the severest bugs (a few divisions whose divisor is not checked to be different from zero) and receiving the worst reliability rating (D).

The code smells found in the assignments are mostly related to styling inconsistencies, coding or naming conventions not followed or duplicated code.

## 3.4 Batch analysis of an assignment

The following step was to understand how to run a batch analysis on a large group of projects; that is, how to analyse all the projects belonging to an assignment at the same time, grouping the results in order to obtain a general picture.

Having worked out a way to automatically run the static analysis on every student's commit, the initial idea was to rework the data obtained from said analyses to avoid running them again.

Different options offered by SonarQube were considered, such as applications or portfolios, that are functionalities that allow to group different SonarQube projects into a single one to aggregate the data. Though probably viable solutions, these functionalities are only available in the paid editions of SonarQube, so other options were then explored.

The other problem with this initial idea was that it was nearly impossible to run an analysis on past assignments or, in general, on projects that had not been already analysed in the context of the GitLab pipeline. This was a severe limitation for the initial idea, which was worth considering.

Two more "manual" alternatives were then discussed, both needing some kind of grouping of the projects before running the analysis. Choosing one of these options also meant decoupling the individual modality of the analysis from the batch one: each one could be run independently, and the latter would no longer need for every project to be run through the CI/CD pipeline before aggregating the data.

### 3.4.1 Multi-project batch analysis

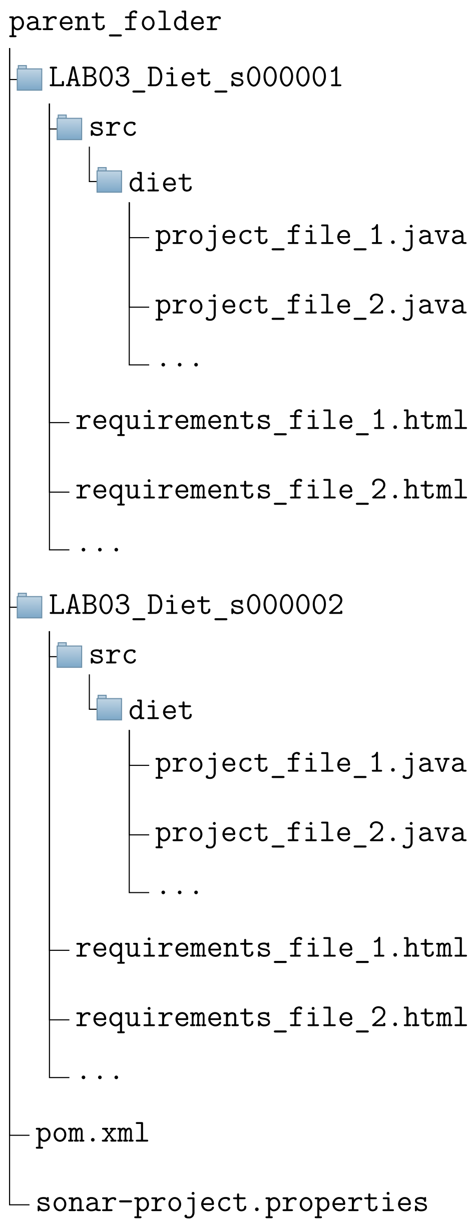
The first option considered for a batch analysis consisted in creating a large project, containing different sources: specifically, every student's project would have to be indicated as a different source.

The setup of the analysis would be quite simple: all the students' projects would have to be located inside a parent folder, which would also contain the POM file (not mutated from the previously slightly modified one, designed for the "individual" analysis [appendix A.2]) and a *sonar-project.properties* file [appendix A.3.1]. This file serves as a configuration file for the SonarScanner plugin that performs the SonarQube analysis and would be necessary, in this case, to indicate each subfolder as a different source, to be treated as such in the final analysis. SonarQube, in fact, divides the analysis by the different sources, but given that the project is only one (described by the POM file in the parent folder), this specific directive would have to be provided.

After manually testing this solution (before developing an automation to apply it to a massive analysis), it was evident that this methodology would work, as a test analysis was successfully completed, but a few problems would have to be resolved.

Firstly, all the extra files (such as HTML - the requirements documents - or XML files - the POM files) and the test files would have to be ignored. While the extra files did not represent a problem at compile time, test files, to be obviously ignored in the analysis as files not developed by the students, would have to excluded from the compilation too to avoid conflicting namespaces and packages names.

Generally, given that each subfolder would contain a project based on the same initial template provided by the teachers, all packages names (and therefore all import commands, if present) would have to be renamed uniquely: grouping



```
parent_folder
├ LAB03_Diet_s000001
│ ├ src
│ │ └ diet
│ │   ├ project_file_1.java
│ │   ├ project_file_2.java
│ │   └ ...
│ ├ requirements_file_1.html
│ ├ requirements_file_2.html
│ └ ...
├ LAB03_Diet_s000002
│ ├ src
│ │ └ diet
│ │   ├ project_file_1.java
│ │   ├ project_file_2.java
│ │   └ ...
│ ├ requirements_file_1.html
│ ├ requirements_file_2.html
│ └ ...
├ pom.xml
└ sonar-project.properties
```

**Figure 3.2:** Multi-project batch analysis folder schema (based on projects and folder names for the third assignment)

projects based on a common template under the same project means that a package designed by a student conflicts with all the others developed by their colleagues.

While probably all solvable issues, a second option for performing a batch analysis was explored.

### 3.4.2 Multi-module batch analysis

The other option, while by many aspects similar, presented some key differences with the previous one.

The idea was to exploit Maven's multi-module functionality: in Maven a project can be composed of different modules, each being a different project, that are then grouped in a parent one. While quite similar to the previous solution, this idea is different: previously, Maven was unaware of the different submodules and would just compile every Java file contained in the parent folder. It was then up to the SonarScanner (via the configuration provided in the *sonar-project.properties* file) to distinguish the different subprojects and treat them as such.
In this case, on the other hand, Maven is fully aware of the modularity of the parent project, and then SonarQube merely distinguishes the folders from which the different analysed files were taken.

This also solves the problem of conflicting namespaces: even if different packages share a common name, this does not represent a problem because Maven compiles each submodule separately and assembles them in the larger project only afterwards.

For the same reason tests would not have to be excluded from the compilation, but nonetheless they would need to be subsequently avoided by the analysis, again given that they are not written by the students.

In detail, to carry out this approach, each submodule would need to contain a POM file [appendix A.3.2] that describes it and that references the parent project where the submodule resides. Then, the parent folder would need to contain another POM file [appendix A.3.2], describing the larger project and referencing all the submodules which compose it.

Another manual test was conducted following this alternative approach, verifying the validity of this solution too.

In the process, the parent POM file [appendix A.3.2] was also designed to contain all sorts of directives for the analysis: after specifying the language of the analysis (Java, in this case), all tests and example classes files were excluded (namely excluding all files containing the `@Test` directive, though some further exclusions might be needed) and all additional files (HTML, XML) were ignored. Code duplication and coverage were also disabled, as duplication between different projects might be picked up (though only due to the shared template) and code coverage was null, given that all tests files were ignored. Finally, SonarQube's Source Control Manager was disabled too: this is a functionality that allows to analyse the author of each line of code, amongst other things, but such a feature was not necessary for purpose of the work.

### 3.4.3 Choosing an approach for the analysis

After testing both options manually (by manually typing in the necessary files and locating them were needed) both approach resulted valid and viable.

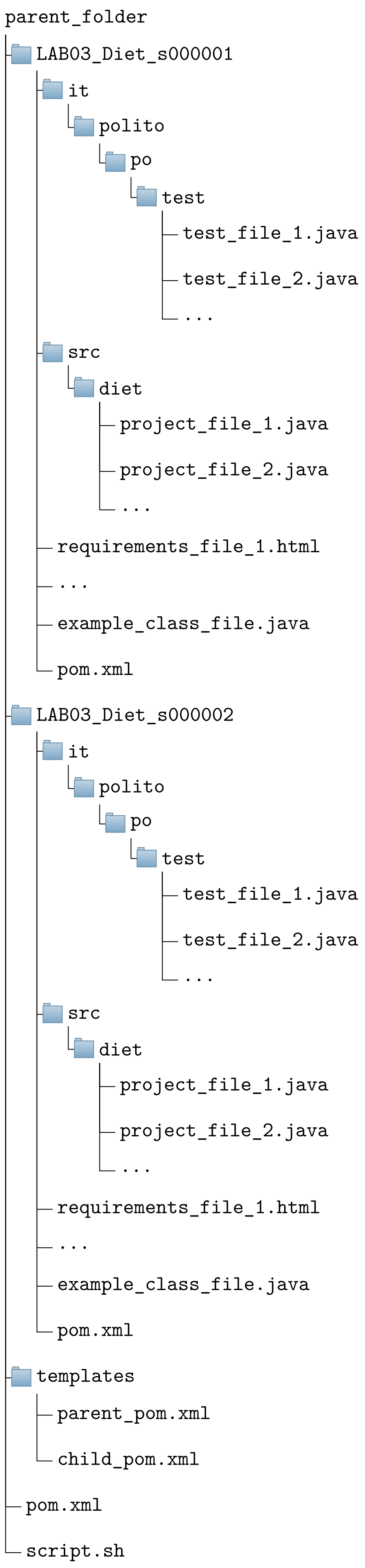
The choice made, though obviously arguable, was to follow the latter multi-module approach. The choice was made considering that it would have been probably slightly easier to automate the process of writing and placing the POM files for the multi-module approach, rather than automating the necessary renaming of packages and imports for the multi-project solution.
Furthermore, the multi-module alternative allowed to avoid interfering with the files written by the students: while renaming the packages would likely cause no harm, especially in small projects like programming assignments, philosophically it would mean meddling with the students' work, perhaps slightly contradicting the goal of the analysis. The chosen method, on the other hand, only needed an "external approach", modifying files unrelated to the students' code.

After making this choice it was clearly necessary to automate the process, as it would not be feasible, for a massive analysis, to manually place all POM files and compile them correctly following the multi-module solution.

### 3.4.4 Automating the analysis

To create an automation, templates for the parent POM [appendix A.3.2] and the child POM [appendix A.3.2] were created.

Afterwards, a Bash script was created [appendix A.3.2]. To work, this script has to be placed inside the parent folder, alongside a folder named "templates", containing the *parent_pom.xml* and *child_pom.xml* files.
This script considers all the subfolders of the folder in which it is placed as submodules (excluding the "templates" and "target" subfolders): it visits all subfolders and copies there the child POM, substituting the `parentArtifactId` (that is, the id of the parent POM) with the value passed as parameter to the script (this allows to customize the name that will eventually be displayed in SonarQube) and the `childArtifactId` with the name of the subfolder (which is by definition unique, so no conflicts can arise).
After finalizing the child POM, the parent POM is modified to add the visited subfolder to the list of modules. This allows to create the cross-referencing needed by Maven's multi-modular solution.
Finally, after iterating through all subfolders, the script launches a Maven command that builds the project and then launches the analysis, whose results are then saved to the SonarQube instance specified in the parent POM.



```
parent_folder
├─ LAB03_Diet_s000001
│  ├─ it
│  │  └─ polito
│  │     └─ po
│  │        └─ test
│  │           ├─ test_file_1.java
│  │           ├─ test_file_2.java
│  │           └─ ...
│  ├─ src
│  │  └─ diet
│  │     ├─ project_file_1.java
│  │     ├─ project_file_2.java
│  │     └─ ...
│  ├─ requirements_file_1.html
│  ├─ ...
│  ├─ example_class_file.java
│  └─ pom.xml
├─ LAB03_Diet_s000002
│  ├─ it
│  │  └─ polito
│  │     └─ po
│  │        └─ test
│  │           ├─ test_file_1.java
│  │           ├─ test_file_2.java
│  │           └─ ...
│  ├─ src
│  │  └─ diet
│  │     ├─ project_file_1.java
│  │     ├─ project_file_2.java
│  │     └─ ...
│  ├─ requirements_file_1.html
│  ├─ ...
│  ├─ example_class_file.java
│  └─ pom.xml
├─ templates
│  ├─ parent_pom.xml
│  └─ child_pom.xml
├─ pom.xml
└─ script.sh
```

**Figure 3.3:** Multi-module batch analysis folder schema (based on projects and folder names for the third assignment)

To launch the script [appendix A.3.2], as mentioned, it is necessary to pass the parent's artifact id and the SonarQube token for authentication. This choice was made to make the script secure and portable, given that the token is specific to the SonarQube instance.

This script greatly automates the setup of the analysis: it is only necessary to place all the projects to be analyzed in a common folder, to place the templates and the script in this folder and then launch the latter with the appropriate parameters.

## 3.5 Analysing a past exam

After completing the automation of the analysis, a past assignment was analysed in its entirety.

An exam of the aforementioned *Object-oriented programming* course was analysed: specifically, the first call for the exams of the 2022/2023 edition of the course, held on June 26th, 2023.

The exam consisted of 428 projects to be analysed: as described, the projects were placed in a common folder, alongside the templates and the script, and then the script was launched, setting up the files as needed and then performing the analysis.

A slight modification was made to the script in the process: the compilation would previously halt if a single module's compilation would fail. This had to be avoided, because it is common in this kind of assignments to find projects that do not compile, either for a student's mistake or because the student decided not to finish the exam or assignment. A `--fail-never` option was added to the Maven compilation command, ensuring to simply skip and ignore all modules that do not compile.

To extrapolate all the data, i.e. how many occurrences of every bug or code smell were found for every student, a SQL script was written [appendix A.3.2]. The script queries the database which SonarQube uses to save the analysis data.
Assuming that every folder containing a student's project is named according to their student number, such as *s123456*, the script selects all students numbers from the components names (the components are all the files SonarQube analyses, which, as mentioned, contain the student number in their path), then performs a *left join* with another table, created selecting all the students numbers for the students whose projects contained at least one occurrence of the rule being considered and the number of such occurrences.
Therefore, this script outputs two columns: a column containing all students numbers and another column listing the occurrences of the specific issue for each student.

This query has to be performed for every rule of interest (only the commonest rules were selected for the analysis, as will be mentioned in the following chapter) by specifying the rule's id: these ids were retrieved manually, from the `rules` table of the database.

The following chapter will describe in detail the findings of this analysis: all the data retrieved, the commonest bugs and code smells registered and the amount in which they were found. Subsequently, a statistical analysis will be performed, in order to understand the correlation between said bugs and code smells and the final mark the students received.

# Chapter 4

# Analysis results

This chapter will lay out the results of the analysis conducted on the first call for the exams of the 2022/2023 edition of the *Object-oriented programming* course, held on June 26th, 2023, as mentioned in the previous chapter.

All the data were gathered in an Excel spreadsheet [appendix B]: marks and occurrences of every bug or code smell for every student were registered.

Specifically, the spreadsheet contains multiple sheets:

- A "Data" sheet, which contains a row for every student: for every student, a few columns summarize general data (such as the final mark received, the number of lines of code of their project and the number of bugs and code smells totalled) and then each issue is reported in a different column, marking, for every student, how many occurrences of a specific bug or code smell were found;
- An "Analysis" sheet reports the results of the general analysis described in section 4.1;
- A "Rules" sheet reports all the rules that raise bugs or code smells that were considered: their unique id (UUID) is reported alongside their name, their shortened name (used for brevity in the other sheets) and part of the data regarding the single issues analysis described in section 4.2;
- For every bug or code smell a separate sheet is then presented, reporting in two different columns the marks of the students who registered said issue and the ones of those who did not; the results of the t-test analysis described in section 4.2 are then laid out.

Only the rules originating issues which were the commonest amongst all projects were selected: this was done to eliminate outliers, but it was a purely subjective evaluation. All the ignored rules were anyway very scarcely registered and deemed not important for the goal of the analysis: generally speaking, rules totalling less then 20 unique occurrences (thus appearing in a very limited number of projects, as multiple occurrences can be found in the same project) were not considered, especially if regarding issues too specific for students at their first approach with object-oriented programming. This means that out of the 497 registered bugs 438 were analysed, and out of the 3952 registered code smells 2791 were studied.

After selecting the rules, the first step in the analysis of the results was to set aside all the projects that did not receive a final evaluation, either because the student was absent (but having booked the exam the project was created) or decided to withdraw from the exam (which is possible by not submitting a fully-working *home* version that passes all requirements). This meant reducing the total number of projects from 428 to 216.

Then, two kinds of analyses were performed: a general analysis, trying to understand the correlation between the number of bugs or code smells and the evaluation, and one specific for every issue, trying to study its significance for the final mark.

## 4.1 General analysis

### 4.1.1 Marks - issues correlation

Firstly, it was interesting to understand whether some sort of correlation between the number of bugs or code smells found in every project and the final mark existed.

To study it, the linear correlation coefficient (also known as Pearson's correlation coefficient) was computed. This is a coefficient that ranges from -1 to 1 and is a measure of dependence between two variables. Specifically, if the coefficient is positive a positive correlation exists, meaning that the two variables change in the same direction (if one increases, the other increases too and vice versa). On the other hand, if two variables are negatively correlated they change in opposite directions: if one increases, the other decreases [51].

The strength of a correlation is measured by it "effect size", defined by two thresholds: if a correlation coefficient is greater than 0.3, then the correlation has medium strength. If it is greater than 0.5, the two variables are strongly correlated [52].

The correlations were computed using the CORREL function in Excel [53], both considering all bugs and code smells found and only the ones relative to the rules chosen for the analysis.

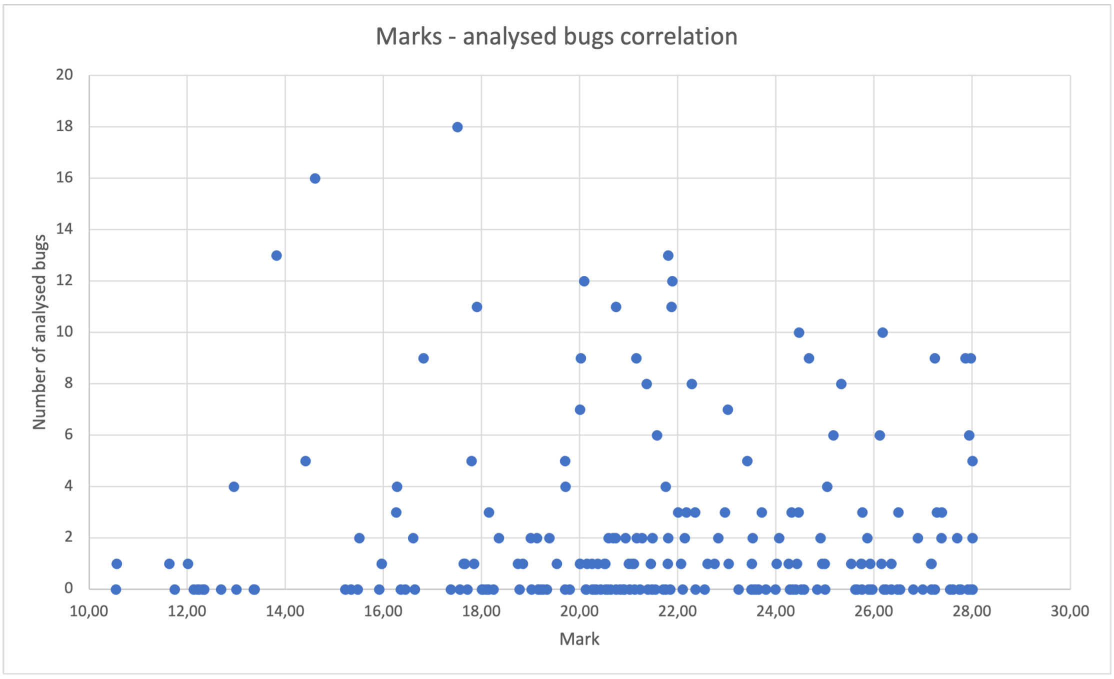


**Figure 4.1:** Marks - analysed bugs correlation

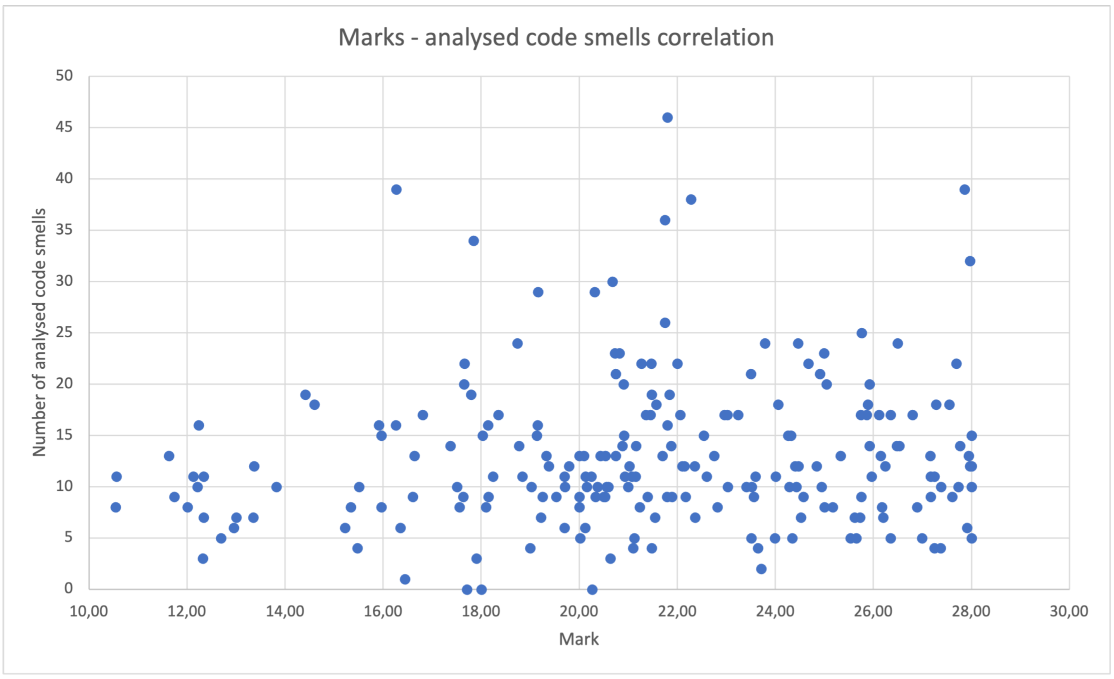


**Figure 4.2:** Marks - analysed code smells correlation

| Correlation | Coefficient |
|---|---:|
| Marks - bugs | 0,02405504 |
| Marks - code smells | 0,09117172 |
| Marks - analysed bugs | 0,02266843 |
| Marks - analysed code smells | 0,11144053 |

**Table 4.1:** Correlation between final marks and bugs or code smells found

The results quite evidently highlight almost no correlation between the variables. A first result to be noted, then, is that generally speaking the presence of code smells or bugs does not affect the evaluation: this is likely because the final mark strongly depends on the number of tests the program passes, tests that do not consider nor verify the quality of the software.

### 4.1.2 Lines of code - issues correlation

The correlation between the total number of lines of code of the projects and the number of bugs or code smells was also researched. This was done to verify whether a student producing more lines of code, thus likely solving more requirements, also registers more issues.

| Correlation | Coefficient |
|---|---:|
| Lines - bugs | 0,17453429 |
| Lines - code smells | 0,25949454 |
| Lines - analysed bugs | 0,15144641 |
| Lines - analysed code smells | 0,22289385 |

**Table 4.2:** Correlation between number of lines of code and issues found

In this case the correlation is weak, though present: this means that the more lines of code written, the more likely the student is to produce bugs or code smells. This argument is true but weak, so it cannot be taken as a general assumption.

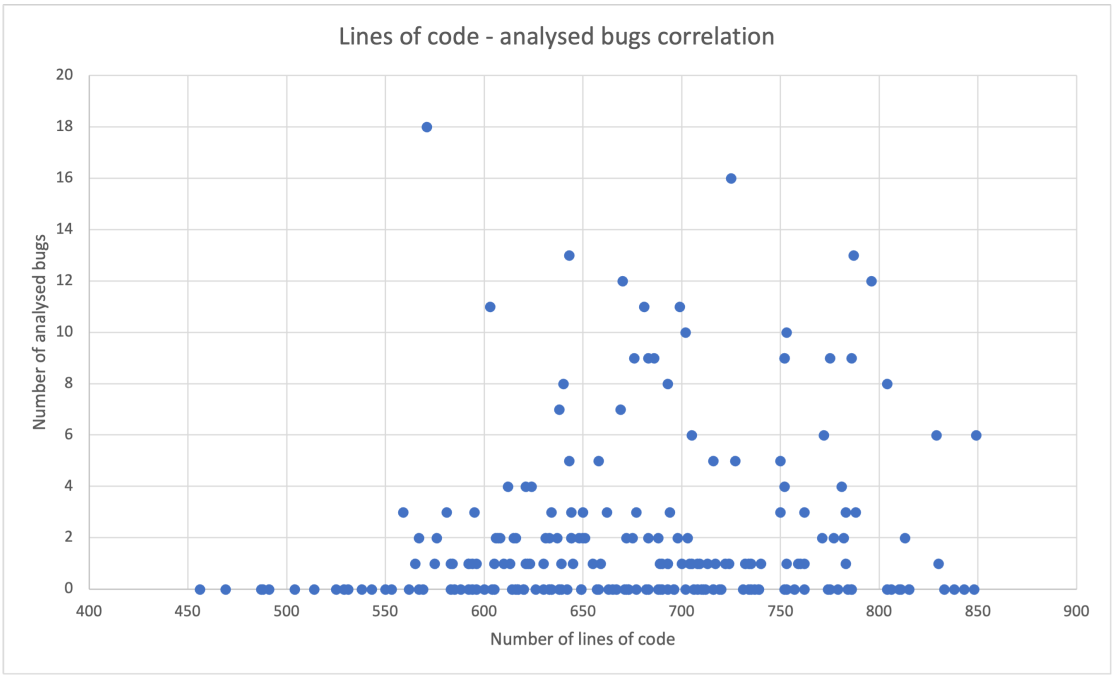


**Figure 4.3:** Lines of code - analysed bugs correlation

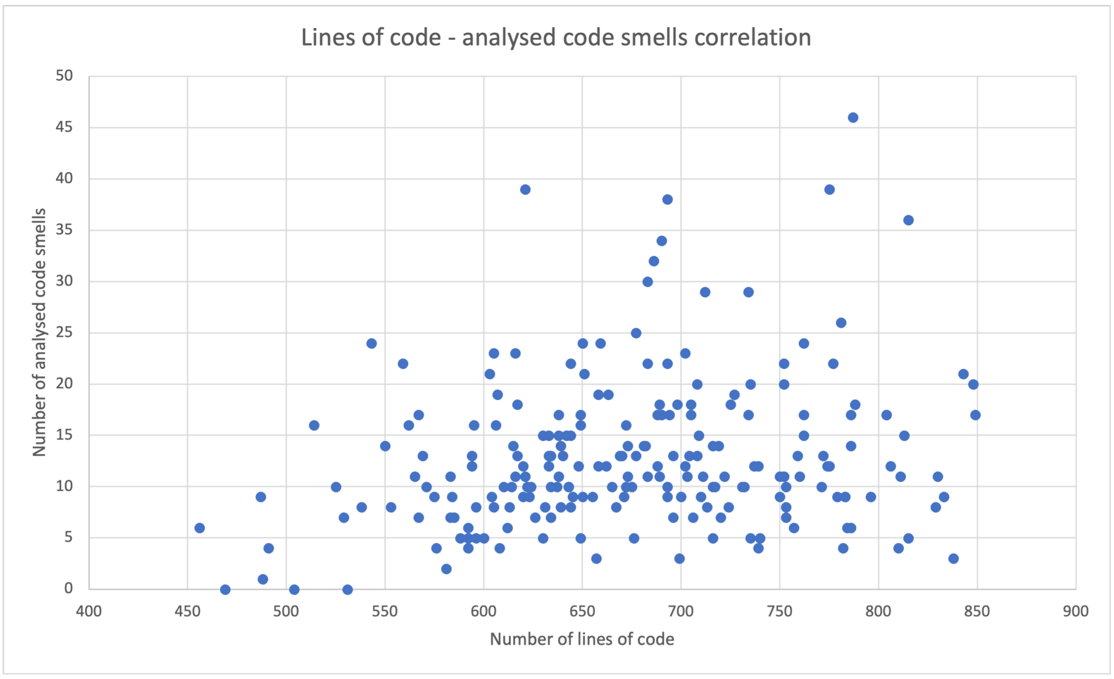


**Figure 4.4:** Lines of code - analysed code smells correlation

## 4.2 Single issues analysis

After exploring some general findings, all the registered bugs and code smells were analysed. As mentioned, only the commonest ones were considered.

Generally speaking, the major issues found are:

- Types management (comparing different types or using nullable types without considering the `null` case);
- Not checking hazardous cases such as possible divisions by zero;
- Comparisons of strings and Boxed types by reference (with `==`) and not by value (using `.equals()`);
- Code formatting conventions not observed;
- Methods, fields or pieces of code left unused or commented out.

All the bugs and code smells relative to the rules selected for the analysis will be listed and explained in the following pages. For every issue, a series of considerations was carried out.

Firstly, the correlation between the occurrences of the single issue and the final mark was computed, using the aforementioned Pearson's linear correlation coefficient. This was meant to understand whether the presence of a single bug or code smell is significant for the final evaluation.

Secondly, a more specific test was carried out to understand the actual influence a bug or code smell had on the final mark: the average mark of the students who incurred at least one occurrence of the issue at hand was computed, and then the average mark of the students whose projects did not contain any occurrence of such issue was calculated too. To verify whether the difference between the two means is significant, a t-test was performed.

A t-test is a statistical test that is used to compare the means of two groups and to understand whether the difference is significant. In this case a two-sample t-test assuming different variances was performed, given that the two groups are independent and that no hypotheses can be formulated on their actual variance.

The two-tailed t-test result will be considered: this is the result to evaluate when one mean can be either higher or lower than the other, and both directions are significant [54]. In fact, this will be useful to determine whether the ones who registered a certain issue got in average a different mark than the ones who did not register it; that is, if said issue had any influence on the final mark or, even, if the mark had any influence in finding more of those issues (for example, it could be plausible to find more bugs with a lower mark).

The results of these tests will be laid out in the following pages for every bug or code smell: every analysed issue will be explained, a concrete code example will be presented and then the numbers of the analysis will follow. All code examples are taken from the analysis itself, so they are unmodified first-hand examples of real code written by students during the exam and of the consequent issues.

At the end, all the most important findings will be summarized (in section 4.3).

### 4.2.1 Bug: Division by zero

#### Bug explanation

SonarQube's description of the bug is "Zero should not be a possible denominator". This bug is registered when a divisor is not checked to be different from zero (if a divisor is zero, a fatal error occurs). It usually happens when the divisor is derived, even indirectly, from a user input.

#### Bug example

This bug was found, for example, in this method:

```java
    public double getCompleteness() {
        double totSlot = 0;
        double sa = 0;
        for (Schedule s : schedules.values()) {
            totSlot += s.getSlots().size();
            sa += s.getAppointments().size();
        }

    return sa/totSlot;
```

Here, `totSlot` is zero if the `schedules` array is empty, but no check is performed: if the array is in fact empty, the program crashes.

#### Analysis

The occurrence of this bug has no significant correlation neither with the final mark nor with the number of lines of code, as shown in table 4.3.

The important result, described in table 4.4, is that P(T<=t) two-tail is less than alpha (the chosen standard significance level): this means that the difference in the average mark between the group of students who incurred this bug and the

| **Description** | **Value** |
|---|---|
| Total occurrences | 24 |
| Number of projects involved | 20 (9.26 %) |
| Correlation with final mark | 0.21981811 |
| Correlation with number of lines of code | 0.153771416 |

**Table 4.3:** Division by zero bug correlation analysis

| | **Bug found** | **Bug not found** |
|---|---|---|
| Mean | 24.1335 | 21.10857143 |
| Variance | 13.2058766 | 18.13863077 |
| Observations | 20 | 196 |
| Hypothesized mean difference | 0 | |
| Alpha | 0.05 | |
| Degrees of freedom | 25 | |
| t Stat | 3.48629712 | |
| P(T<=t) two-tail | 0.00182719 | |
| t Critical two-tail | 2.05953855 | |

**Table 4.4:** Division by zero bug t-test analysis

group of the ones who did not is significant. Therefore, this bug had an influence on the final mark, or vice versa (perhaps a variation in the final mark means more or less bugs found, given that the final mark is determined by tests that do not consider software quality).

### 4.2.2 Bug: String ==

#### Bug explanation

SonarQube states that "Strings and Boxed types should be compared using "equals()"". The reason is simple: in Java, `==` compares two objects by reference. That is, `==` tests whether two references specifically refer to the same object, whereas the `.equals()` method verifies whether two objects, even if different by reference, are logically the same.

Although this bug relates to all boxed types, such as `Int`, `Long` or `Char`, it is more frequently found when comparing strings.

### Bug example

An example from the analysis in which the bug was detected:

```
1      public void addDoctor(String id, String name, String surname,
      String speciality) throws MedException {
2          if (!this.specialities.contains(speciality)) {
3              throw new MedException();
4          }
5          for (int i=0; i<doctors.size(); i++) {
6              if (doctors.get(i).id == id) {
7                  throw new MedException();
8              }
9          }
10
11         // ...
12     }
```

The id (line 6) is compared by reference, which will likely return different results than expected.

### Analysis

| Description | Value |
|---|---|
| Total occurrences | 268 |
| Number of projects involved | 62 (28.7 %) |
| Correlation with final mark | 0.003264319 |
| Correlation with number of lines of code | 0.073938472 |

**Table 4.5:** String == bug correlation analysis

This bug is quite common, as it has a lot of occurrences and is found in almost one third of projects, but it has no correlation with the final mark and the difference between those who registered the bug and those who did not is not significant.

|  | **Bug found** | **Bug not found** |
|---|---|---|
| Mean | 22.1480645 | 21.08292208 |
| Variance | 15.4432782 | 19.35825742 |
| Observations | 62 | 154 |
| Hypothesized mean difference | 0 |  |
| Alpha | 0.05 |  |
| Degrees of freedom | 125 |  |
| t Stat | 1.73986195 |  |
| P(T<=t) two-tail | 0.08434456 |  |
| t Critical two-tail | 1.97912411 |  |

**Table 4.6:** String == bug t-test analysis

### 4.2.3 Bug: Collection.get()

#### Bug explanation

The SonarQube description of the bug is "Inappropriate "Collection" calls should not be made". This bug is reported when methods like `remove()` or `contains()` are called with a parameter of a different type than the one of the collection the method operates on.
The signature of these methods, such as `Collection.contains(Object o)` or `Collection.remove(Object o)`, presents a parameter of type `Object`, so the call itself is allowed, but this usually leads to unexpected results, as the method will always return `false` or `null`.

#### Bug example

The bug is found in pieces of code such as:

```java
 public String setAppointment(String ssn, String name, String surname, String code, String date, String slot) throws MedException {
     if (!doctors.containsKey(code) || !doctors.get(code).getSchedules().contains(date)) {
         throw new MedException();
     }

     // ...
 }
```

In this case the `getSchedules()` method retrieves a `List<Schedule>`, but the `contains()` method checks for a `String` type, so it will always return `false`.

**Analysis**

| **Description** | **Value** |
|---|---|
| Total occurrences | 43 |
| Number of projects involved | 14 (6.48 %) |
| Correlation with final mark | -0.072426947 |
| Correlation with number of lines of code | 0.048735667 |

**Table 4.7:** Collection.get() bug correlation analysis

| | **Bug found** | **Bug not found** |
|---|---|---|
| Mean | 19.8707143 | 21.49386139 |
| Variance | 14.0542995 | 18.59106462 |
| Observations | 14 | 202 |
| Hypothesized mean difference | 0 | |
| Alpha | 0.05 | |
| Degrees of freedom | 15 | |
| t Stat | -1.5504928 | |
| P(T<=t) two-tail | 0.14186221 | |
| t Critical two-tail | 2.13144955 | |

**Table 4.8:** Collection.get() bug t-test analysis

Although being quite a severe bug, a correlation with the final mark is not found. The t-test does not signal any significance.

### 4.2.4 Bug: Optional.get()

**Bug explanation**

SonarQube explains this bug stating that an "Optional value should only be accessed after calling isPresent()". This means that, when dealing with an Optional value, it should be first verified whether said Optional actually holds a value: otherwise, a `NoSuchElementException` will be thrown.
To avoid the exception, the `isPresent()` or `!isEmpty()` methods should be called. Alternatively, other methods such as `orElse()`, `orElseGet()` or `orElseThrow()` can be used to specifically deal with an Optional value.

### Bug example

```
    public String getMeetingTitle(String meetingId) {
        return meetings.stream().filter(m->m.getId().equals(meetingId
)).map(Meeting::getTitle).findFirst().get();
    }
```

In this case the `findFirst()` method returns an Optional value that will be empty if the Stream is empty too: if no meeting is found by the filter method or if there are no meetings at all, an exception will be thrown.

### Analysis

| **Description** | **Value** |
|---|---|
| Total occurrences | 28 |
| Number of projects involved | 17 (7.87 %) |
| Correlation with final mark | 0.03400739 |
| Correlation with number of lines of code | 0.061490846 |

**Table 4.9:** Optional.get() bug correlation analysis

| | **Bug found** | **Bug not found** |
|---|---|---|
| Mean | 21.7858824 | 21.35472362 |
| Variance | 14.3255007 | 18.79739475 |
| Observations | 17 | 199 |
| Hypothesized mean difference | 0 | |
| Alpha | 0.05 | |
| Degrees of freedom | 20 | |
| t Stat | 0.44538535 | |
| P(T<=t) two-tail | 0.66082476 | |
| t Critical two-tail | 2.08596345 | |

**Table 4.10:** Optional.get() bug t-test analysis

The bug is not correlated with the final mark and the t-test highlights no significant difference.

### 4.2.5 Bug: Null deref

#### Bug explanation

SonarQube specifies that "Null pointers should not be dereferenced": doing so will throw a `NullPointerException`.

#### Bug example

```
    public boolean equals(Object o) {
        Seat s = (Seat)o;
        return (this.getLetter() == s.getLetter() && this.getRow() ==
    s.getRow());
    }
```

In this example the parameter is of type `Object`, which means that it can be `null` too. The first line simply casts the parameter, but if `o` is in fact `null`, the `getLetter()` method will throw an exception.

#### Analysis

| Description | Value |
|---|---|
| Total occurrences | 20 |
| Number of projects involved | 10 (4.63%) |
| Correlation with final mark | 0.135923946 |
| Correlation with number of lines of code | 0.129891303 |

**Table 4.11:** Null deref bug correlation analysis

| | Bug found | Bug not found |
|---|---|---|
| Mean | 22.433 | 21.33796117 |
| Variance | 33.2396678 | 17.77273534 |
| Observations | 10 | 206 |
| Hypothesized mean difference | 0 | |
| Alpha | 0.05 | |
| Degrees of freedom | 9 | |
| t Stat | 0.59297572 | |
| P(T<=t) two-tail | 0.56779674 | |
| t Critical two-tail | 2.26215716 | |

**Table 4.12:** Null deref bug t-test analysis

The bug is found in a small number of projects; the correlation and t-test analyses do not highlight peculiarities.

### 4.2.6 Bug: Silly ==

#### Bug explanation

The bug is explained by SonarQube stating that "Silly equality checks should not be made". Quite simply, comparisons of dissimilar types will always return false: this means that the equality check in question can be removed or, more likely, that the comparison has to be fixed.

#### Bug example

```java
public class TrainManager {
    private List<String> stopsList;
    private Map<Integer, Stop> stopsMap;
    private Stop lastStop;

    // ...

    public int setLastStop(String stop) {
        this.lastStop = this.stopsMap.get(stop);
        int tot = 0;
        boolean found = false;
        for (String s:this.stopsList) {
            if (s.equals(lastStop)) {
                found = true;
            }
            if (found == true) {
                tot += this.getNumberOfBookings(lastStop.getName
(), s);
            }
        }
        return tot;
    }

    // ...
}
```

In this example `lastStop` is of type `Stop`, being extracted from the `stopsMap` map. However, it is then checked against `s`, which is a value from the `stopsList` list, which holds elements of type `String`. This means that the comparison will always return `false`, and the function will always return 0.

**Analysis**

| **Description** | **Value** |
|---|---|
| Total occurrences | 13 |
| Number of projects involved | 9 (4.17%) |
| Correlation with final mark | 0.02613918 |
| Correlation with number of lines of code | 0.107718868 |

**Table 4.13:** Silly == bug correlation analysis

| | **Bug found** | **Bug not found** |
|---|---|---|
| Mean | 22.0888889 | 21.35821256 |
| Variance | 11.2689611 | 18.7342099 |
| Observations | 9 | 207 |
| Hypothesized mean difference | 0 | |
| Alpha | 0.05 | |
| Degrees of freedom | 9 | |
| t Stat | 0.63059433 | |
| P(T<=t) two-tail | 0.54398385 | |
| t Critical two-tail | 2.26215716 | |

**Table 4.14:** Silly == bug t-test analysis

The bug appears in a limited number of projects and no correlation is found. The t-test analysis does not report any significant difference.

### 4.2.7 Bug: Intermediate Stream

#### Bug explanation

SonarQube explains the bug by saying that "Intermediate Stream methods should not be left unused". This happens when a Stream is not concluded by a terminal operation: as intermediate operations on a Stream are lazy, if no terminal operation requests for any data no work will ever be executed. This is symptomatic of unfinished pieces of code.

### Bug example

```
    public Map<String, List<String>> findSeats(String begin, String
    end, String klass) {
        cars.values().stream().filter(c -> c.hasAvailableSeatStops(
    begin, end));

        return null;
    }
```

The bug is found in pieces of code that have not been finished by the students: the `return null` statement is provided by the initial project template and is meant to be substituted; not doing so means that the student decided not to complete the method.

### Analysis

| Description | Value |
|---|---|
| Total occurrences | 10 |
| Number of projects involved | 8 (3.7%) |
| Correlation with final mark | -0.090408697 |
| Correlation with number of lines of code | -0.003773042 |

**Table 4.15:** Intermediate Stream bug correlation analysis

| | Bug found | Bug not found |
|---|---|---|
| Mean | 18.97125 | 21.48163462 |
| Variance | 11.0397268 | 18.49360698 |
| Observations | 8 | 208 |
| Hypothesized mean difference | 0 | |
| Alpha | 0.05 | |
| Degrees of freedom | 8 | |
| t Stat | -2.0713214 | |
| P(T<=t) two-tail | 0.07208092 | |
| t Critical two-tail | 2.30600414 | |

**Table 4.16:** Intermediate Stream bug t-test analysis

The bug, appearing in a very limited number of projects, has no correlation with the final mark; the t-test analysis shows no significant difference.

### 4.2.8 Bug: Math operands cast

#### Bug explanation

Regarding this bug, SonarQube states that "Math operands should be cast before assignment": the problem lies in the fact that, for example, dividing two integers and then assigning to a double is possible (due to automatic type conversion), but upon assigning precision will already be lost, as operating on integers always generates an integer. While technically possible, this will probably yield unexpected results.

#### Bug example

```
    Double compare1 = (double) (starHour + starMin/100);
```

In this case either `starHour` or `starMin` should be cast to `double` in order to preserve floating-point precision before the assignment to `compare1`.

#### Analysis

| Description | Value |
|---|---|
| Total occurrences | 32 |
| Number of projects involved | 26 (12.04%) |
| Correlation with final mark | -0.058853922 |
| Correlation with number of lines of code | 0.068137292 |

**Table 4.17:** Math operands cast bug correlation analysis

| | Bug found | Bug not found |
|---|---|---|
| Mean | 21.205 | 21.41378947 |
| Variance | 18.757074 | 18.43428503 |
| Observations | 26 | 190 |
| Hypothesized mean difference | 0 | |
| Alpha | 0.05 | |
| Degrees of freedom | 32 | |
| t Stat | -0.2307878 | |
| P(T<=t) two-tail | 0.81894883 | |
| t Critical two-tail | 2.03693334 | |

**Table 4.18:** Math operands cast bug t-test analysis

No significant evidence is highlighted by the analysis for this bug.

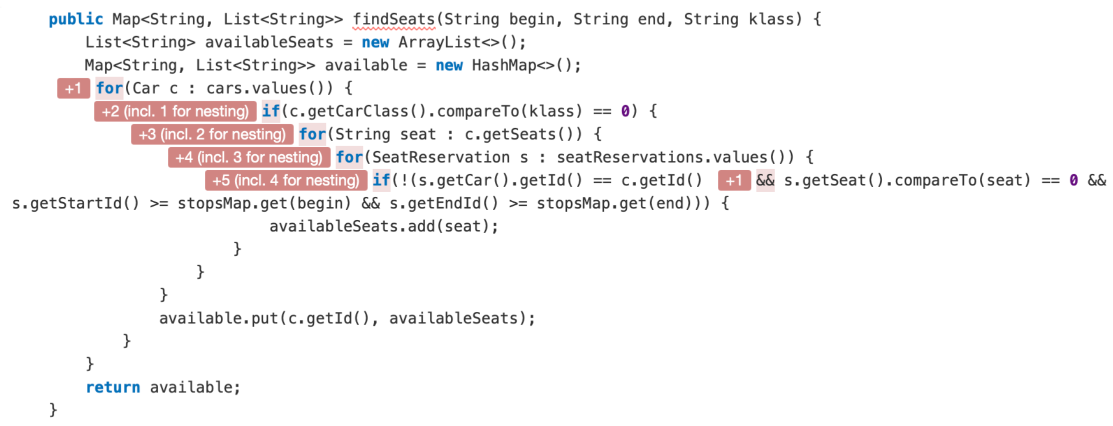
### 4.2.9 Code smell: Cognitive Complexity

#### Code smell explanation

SonarQube states that "Cognitive Complexity of methods should not be too high": the Cognitive Complexity is a measure of how hard the control flow of a method is to understand. Methods with high Cognitive Complexity will be difficult to maintain.

Specifically, the Cognitive Complexity is increased by 1 for every complexity-incrementing statement, such as conditions (`if`, `else`, `case - for the switch statement`), loops (`for`, `while`), logical operators (`&&`, `||`) or exception handlers (`catch`, `throw`). Nested statements count more.

#### Code smell example



```
    public Map<String, List<String>> findSeats(String begin, String end, String klass) {
        List<String> availableSeats = new ArrayList<>();
        Map<String, List<String>> available = new HashMap<>();
    +1 for(Car c : cars.values()) {
        +2 (incl. 1 for nesting) if(c.getCarClass().compareTo(klass) == 0) {
            +3 (incl. 2 for nesting) for(String seat : c.getSeats()) {
                +4 (incl. 3 for nesting) for(SeatReservation s : seatReservations.values()) {
                    +5 (incl. 4 for nesting) if(!(s.getCar().getId() == c.getId()  +1 && s.getSeat().compareTo(seat) == 0 &&
 s.getStartId() >= stopsMap.get(begin) && s.getEndId() >= stopsMap.get(end))) {
                        availableSeats.add(seat);
                    }
                }
            }
            available.put(c.getId(), availableSeats);
        }
    }
    return available;
}
```

In this example the Cognitive Complexity of the method is 16, which is above the standard threshold of 15. SonarQube therefore suggests to improve the understandability of the method by reducing its complexity.

#### Analysis

| Description | Value |
|---|---|
| Total occurrences | 37 |
| Number of projects involved | 31 (14.35%) |
| Correlation with final mark | 0.142434151 |
| Correlation with number of lines of code | 0.221994825 |

**Table 4.19:** Cognitive Complexity code smell correlation analysis

| | **Smell found** | **Smell not found** |
|---|---|---|
| Mean | 22.79709677 | 21.15264865 |
| Variance | 14.50948796 | 18.73325979 |
| Observations | 31 | 185 |
| Hypothesized mean difference | 0 | |
| Alpha | 0.05 | |
| Degrees of freedom | 44 | |
| t Stat | 2.179446466 | |
| P(T<=t) two-tail | 0.034694501 | |
| t Critical two-tail | 2.015367574 | |

**Table 4.20:** Cognitive Complexity code smell t-test analysis

The code smell has no significant correlation neither with the final mark nor with the number of lines of code of the project. However, the t-test analysis shows a significant difference between the average mark of the students who registered this code smell and the one of those who did not.

### 4.2.10 Code smell: Methods set static fields

#### Code smell explanation

SonarQube's description of the code smell is "Instance methods should not write to "static" fields": this means that a static field should not be updated by non-static methods, which could lead to errors in case of multiple instances of the class or multiple threads. Static fields should only by updated by `synchronized static` methods.

#### Code smell example

```java
    private static int stopCode = 0;

    public int defineStops(String... stops) {
        for(var s : stops) {
            stopCode++;
            this.stops.put(stopCode, s);
        }
        return this.stops.keySet().size() - 1;
    }
```

### Analysis

| Description | Value |
|---|---|
| Total occurrences | 11 |
| Number of projects involved | 9 (4.17%) |
| Correlation with final mark | 0.016048698 |
| Correlation with number of lines of code | 0.072142935 |

**Table 4.21:** Methods set static fields code smell correlation analysis

| | Smell found | Smell not found |
|---|---|---|
| Mean | 21.70333333 | 21.37497585 |
| Variance | 13.982775 | 18.64665813 |
| Observations | 9 | 207 |
| Hypothesized mean difference | 0 | |
| Alpha | 0.05 | |
| Degrees of freedom | 9 | |
| t Stat | 0.256113648 | |
| P(T<=t) two-tail | 0.803621012 | |
| t Critical two-tail | 2.262157163 | |

**Table 4.22:** Methods set static fields code smell t-test analysis

No significant findings emerge from the statistical analysis for this bug.

## 4.2.11 Code smell: Stream.toList()

### Code smell explanation

SonarQube describes the code smell stating that a ""Stream.toList()" method should be used instead of "collectors" when unmodifiable list needed": the code smell refers to the fact that, when terminating a Stream, the `.collect(Collectors.toList())` method is usually used, but this yields a mutable list. It is then preferable to use `Collectors.toUnmodifiableList()`, introduced in Java 10, although being quite verbose, or better the specific terminator `Stream.toList()`, introduced in Java 16.

### Code smell example

```java
    public Collection<String> getCarsByClass(String klass) {
        return cars.values().stream()
                .filter(c -> c.getClass().equals(klass))
                .map(Car::getId)
                .collect(Collectors.toList());

    }
```

### Analysis

| Description | Value |
|---|---|
| Total occurrences | 479 |
| Number of projects involved | 150 (69.44%) |
| Correlation with final mark | 0.446797723 |
| Correlation with number of lines of code | 0.325362376 |

**Table 4.23:** Stream.toList() code smell correlation analysis

| | Smell found | Smell not found |
|---|---|---|
| Mean | 22.39086667 | 19.11090909 |
| Variance | 15.56285629 | 17.57011608 |
| Observations | 150 | 66 |
| Hypothesized mean difference | 0 | |
| Alpha | 0.05 | |
| Degrees of freedom | 118 | |
| t Stat | 5.392462945 | |
| P(T<=t) two-tail | 3.62281E-07 | |
| t Critical two-tail | 1.980272249 | |

**Table 4.24:** Stream.toList() code smell t-test analysis

The issue is found in the vast majority of projects: it has a positive medium correlation both with the final mark and with the number of lines of code. This means that the more lines of code written or the higher the mark, the more likely it is to register this issue. This probably underlines a general lack of comprehension of this problem, though it being quite specific: it is possible that students are not fully aware of the difference between modifiable and unmodifiable lists and that they are more accustomed to using the `Collectors` methods for terminating every Stream, not knowing the rather new method specific for the purpose. This could mean that the students who write more code and who receive a better evaluation simply have "more opportunities" to incur this code smell.

According to the t-test analysis, the difference between the average mark of the students who incurred this specific code smell and the one of those who did not is significant.

### 4.2.12 Code smell: Standard output to log

#### Code smell explanation

SonarQube states that "Standard outputs should not be used directly to log anything". It is pretty normal for beginners to use the `System.out.println()` method to quickly assess the value of a variable, but a logger should be preferred to foster uniformity and clarity. This is of course not a critical issue, especially for students.

#### Code smell example

```java
    public Collection<String> getSpecialists(String speciality) {

        ArrayList<String> retVal = new ArrayList<>();
        for(Doctor d : doctors.values()) {
            if(d.spec == speciality) {

                retVal.add(d.id);
                System.out.println(d.id);

            }
        }
        return retVal;
    }
```

#### Analysis

| Description | Value |
|---|---|
| Total occurrences | 56 |
| Number of projects involved | 28 (12.96%) |
| Correlation with final mark | 0.086480853 |
| Correlation with number of lines of code | 0.123474702 |

**Table 4.25:** Standard output to log code smell correlation analysis

| | **Smell found** | **Smell not found** |
|---|---|---|
| Mean | 22.34 | 21.24696809 |
| Variance | 7.583007407 | 19.893833 |
| Observations | 28 | 188 |
| Hypothesized mean difference | 0 | |
| Alpha | 0.05 | |
| Degrees of freedom | 51 | |
| t Stat | 1.781023496 | |
| P(T<=t) two-tail | 0.080864575 | |
| t Critical two-tail | 2.00758377 | |

**Table 4.26:** Standard output to log code smell t-test analysis

The analyses reveal no peculiarities.

### 4.2.13 Code smell: entrySet iteration

#### Code smell explanation

The description of this code smell provided by SonarQube is ""entrySet()" should be iterated when both the key and value are needed". Students sometimes iterate over a `keySet` (that is, the set of all the keys of a specific map), and then use each key to retrieve the value corresponding to it stored in the map. This is highly inefficient, as it accesses the map multiple times, and renders the code harder to read. A more efficient way would be to iterate over the `entrySet` itself: that is, the pairs of keys and values.

### Code smell example

```java
public class MedManager {
    SortedMap<String,Doctor> doctors = new TreeMap<>();

    // ...

    public Map<String, List<String>> findSlots(String date, String speciality) {
        Map<String, List<String>> tmp = new HashMap<>();
        for (String code : doctors.keySet()) {
            if (doctors.get(code).getSpec().equals(speciality)) {

                if (doctors.get(code).getSchedule(date)!=null) {
                    tmp.put(code, new ArrayList<>());
                    for (Schedule i : doctors.get(code).getSchedule(date)) {
                        tmp.get(code).add(String.format("%s-%s", i.getStart(),i.getEnd()));
                    }
                }
            }
        }
        return tmp;
    }

    // ...
}
```

In this example both the keys and the values for the `doctors` map are needed; however, the student iterated over the `keySet` alone, resorting than to call `doctors.get(code)` multiple times to obtain the actual value. A leaner solution would be to iterate over the `entrySet`:

```java
    for (Map.Entry<String, Doctor> entry : doctors.entrySet()) {
        String code = entry.getKey();
        Doctor doctor = entry.getValue();
        // ...
    }
```

**Analysis**

| Description | Value |
|---|---|
| Total occurrences | 49 |
| Number of projects involved | 18 (8.33%) |
| Correlation with final mark | 0.073532734 |
| Correlation with number of lines of code | 0.091394548 |

**Table 4.27:** entrySet iteration code smell correlation analysis

| | Smell found | Smell not found |
|---|---|---|
| Mean | 21.72666667 | 21.38335025 |
| Variance | 16.86465882 | 18.57068566 |
| Observations | 18 | 197 |
| Hypothesized mean difference | 0 | |
| Alpha | 0.05 | |
| Degrees of freedom | 21 | |
| t Stat | 0.338083988 | |
| P(T<=t) two-tail | 0.738656425 | |
| t Critical two-tail | 2.079613845 | |

**Table 4.28:** entrySet iteration code smell t-test analysis

No correlation with the final mark is found. The t-test does not signal any significance.

### 4.2.14 Code smell: Map.get() and value test

#### Code smell explanation

SonarQube description of the code smell is ""Map.get" and value test should be replaced with single method call": it is common to test for the presence of a specific key in a map and then, if absent or present, adding or modifying the value for that key. This could be done in an easier and perhaps more readable way with the `computeIfPresent()` and `computeIfAbsent()` methods. However, these methods might not be easy to approach for students.

### Code smell example

```java
    public void addSpecialities(String... specialities) {
        for(String s : specialities) {
            if(!this.specialities.containsKey(s))
                this.specialities.put(s, new ArrayList<>());
        }
    }
```

In the example, the same key is used to test its presence in the map and then to perform the insertion. A compliant solution would be:

```java
    public void addSpecialities(String... specialities) {
        for(String s : specialities) {
            this.specialities.computeIfAbsent(s, s -> new ArrayList
<>());
        }
    }
```

### Analysis

| Description | Value |
|---|---|
| Total occurrences | 47 |
| Number of projects involved | 43 (19.91%) |
| Correlation with final mark | 0.025960833 |
| Correlation with number of lines of code | 0.118133061 |

**Table 4.29:** Map.get() and value test code smell correlation analysis

| | Smell found | Smell not found |
|---|---|---|
| Mean | 21.60767442 | 21.33421965 |
| Variance | 17.45375161 | 18.71145825 |
| Observations | 43 | 173 |
| Hypothesized mean difference | 0 | |
| Alpha | 0.05 | |
| Degrees of freedom | 66 | |
| t Stat | 0.381398187 | |
| P(T<=t) two-tail | 0.704132899 | |
| t Critical two-tail | 1.996564419 | |

**Table 4.30:** Map.get() and value test code smell t-test analysis

No significant evidence is registered from the analyses.

### 4.2.15 Code smell: Use of String constructor

#### Code smell explanation

SonarQube states that "Constructors should not be used to instantiate "String", "BigInteger", "BigDecimal" and primitive-wrapper classes". This is because doing so is less clear and uses more memory than simply using the desired value in the case of strings or `valueOf` for everything else (for example, `Double.valueOf(1.1)`).

#### Code smell example

```java
public class Stop {
    private int ID;
    private String Name;

    public Stop(int id, String name) {
        ID=id;
        Name= new String(name);
    }
}
```

In this case `Name = name` would be sufficient, without using the `String` class constructor.

#### Analysis

| Description | Value |
|---|---|
| Total occurrences | 16 |
| Number of projects involved | 6 (2.78%) |
| Correlation with final mark | 0.103300077 |
| Correlation with number of lines of code | 0.019562819 |

**Table 4.31:** Use of String constructor code smell correlation analysis

The analyses do not highlight correlations or significant differences in the means.

| | **Smell found** | **Smell not found** |
|---|---|---|
| Mean | 23.925 | 21.31619048 |
| Variance | 11.58999 | 18.45144858 |
| Observations | 6 | 210 |
| Hypothesized mean difference | 0 | |
| Alpha | 0.05 | |
| Degrees of freedom | 5 | |
| t Stat | 1.835765991 | |
| P(T<=t) two-tail | 0.125830937 | |
| t Critical two-tail | 2.570581836 | |

**Table 4.32:** Use of String constructor code smell t-test analysis

### 4.2.16 Code smell: Use of raw types

#### Code smell explanation

SonarQube describes the issue saying that "Raw types should not be used", meaning that generic types should not be used without type parameters in variable declarations or return values, as doing so bypasses generic type checking and defers the catch of unsafe code to runtime. This code smell is likely raised mostly when students forget the diamond operator `<>` upon initializing a variable.

#### Code smell example

```java
    private List<Booking> bookings= new LinkedList();
```

In this example the correct initialization would be `new LinkedList<>()`, as the diamond operator helps infer the type from the type of the variable the object will be assigned to.

#### Analysis

| **Description** | **Value** |
|---|---|
| Total occurrences | 17 |
| Number of projects involved | 8 (3.7%) |
| Correlation with final mark | -0.0192284 |
| Correlation with number of lines of code | -0.060174327 |

**Table 4.33:** Use of raw types code smell correlation analysis

The analyses report no significant evidence.

|  | **Smell found** | **Smell not found** |
|---|---|---|
| Mean | 20.0325 | 21.44081731 |
| Variance | 14.73539286 | 18.52935633 |
| Observations | 8 | 208 |
| Hypothesized mean difference | 0 |  |
| Alpha | 0.05 |  |
| Degrees of freedom | 8 |  |
| t Stat | -1.013464477 |  |
| P(T<=t) two-tail | 0.340505494 |  |
| t Critical two-tail | 2.306004135 |  |

**Table 4.34:** Use of raw types code smell t-test analysis

### 4.2.17 Code smell: Public variable fields

#### Code smell explanation

SonarQube explains this bug stating that "Class variable fields should not have public accessibility". This means that a field of a class should either be marked as `static final` or should be private and have public setters and getters if needed. This allows to provide additional validation, ensures that the internal representation is not exposed and that the field can be mutated only by actors internal to the class.

#### Code smell example

```java
public class Stop {
    private int id;
    private String name;
    public boolean isLast;
    public boolean isFirst;

     // ...
}
```

In this case both `isLast` and `isFirst` should not be public, but rather they should have public getters and setters and be private.

#### Analysis

No correlation with the final mark or with the number of lines of code is found and the difference between the average mark of the students that incurred this error and the one of those who did not is not significant.

| Description | Value |
|---|---|
| Total occurrences | 53 |
| Number of projects involved | 14 (6.48%) |
| Correlation with final mark | 0.015538302 |
| Correlation with number of lines of code | -0.071000649 |

**Table 4.35:** Public variable fields code smell correlation analysis

| | Smell found | Smell not found |
|---|---|---|
| Mean | 20.28 | 21.46549505 |
| Variance | 29.49984615 | 17.67216617 |
| Observations | 14 | 202 |
| Hypothesized mean difference | 0 | |
| Alpha | 0.05 | |
| Degrees of freedom | 14 | |
| t Stat | -0.800240067 | |
| P(T<=t) two-tail | 0.436941056 | |
| t Critical two-tail | 2.144786688 | |

**Table 4.36:** Public variable fields code smell t-test analysis

### 4.2.18 Code smell: Lambdas or method refs

#### Code smell explanation

SonarQube states that "Lambdas should be replaced with method references", as method or constructor references are commonly agreed to be, most of the time, more compact and readable than using lambdas, and are therefore preferred.

#### Code smell example

```
    public Collection<String> getCarsByClass(String klass) {
        return cars.values().stream()
                                .filter(c->c.getC().getClasse().equals(
   klass))
                                .map(c->c.getId())
                                .collect(Collectors.toList());
    }
```

In this example a lambda expression is used in the map method to retrieve the car id. The method reference (i.e., `Car::getId`), would be preferred.

### Analysis

| Description | Value |
|---|---|
| Total occurrences | 141 |
| Number of projects involved | 64 (29.63%) |
| Correlation with final mark | 0.204141917 |
| Correlation with number of lines of code | 0.108435018 |

**Table 4.37:** Lambdas or method refs code smell correlation analysis

| | Smell found | Smell not found |
|---|---|---|
| Mean | 22.995625 | 20.71203947 |
| Variance | 12.67301865 | 19.34268919 |
| Observations | 64 | 152 |
| Hypothesized mean difference | 0 | |
| Alpha | 0.05 | |
| Degrees of freedom | 145 | |
| t Stat | 4.004008365 | |
| P(T<=t) two-tail | 9.90753E-05 | |
| t Critical two-tail | 1.976459563 | |

**Table 4.38:** Lambdas or method refs code smell t-test analysis

No correlation is registered by the analysis, but, according to the t-test, the average mark of the students that registered this issue is significantly different from the one of those who did not.

## 4.2.19 Code smell: Local vars returned

### Code smell explanation

The code smell is explained by SonarQube stating that "Local variables should not be declared and then immediately returned or thrown": although it might be convenient to be able to name such variable to better describe what the code does, the method name ought to be sufficient.

### Code smell example

```java
    private int hti(String time) {
        int min = Integer.parseInt(time.split(":")[0])*60 + Integer.
    parseInt(time.split(":")[1]);
        return min;
    }
```

In this example the `min` variable is instantiated and then immediately returned: this could be avoided by directly returning the result of the expression.

### Analysis

| **Description** | **Value** |
|---|---|
| Total occurrences | 86 |
| Number of projects involved | 47 (21.76%) |
| Correlation with final mark | -0.103246467 |
| Correlation with number of lines of code | 0.095540766 |

**Table 4.39:** Local vars returned code smell correlation analysis

| | **Smell found** | **Smell not found** |
|---|---|---|
| Mean | 20.20148936 | 21.71881657 |
| Variance | 17.29031295 | 18.29754264 |
| Observations | 47 | 169 |
| Hypothesized mean difference | 0 | |
| Alpha | 0.05 | |
| Degrees of freedom | 75 | |
| t Stat | -2.19891298 | |
| P(T<=t) two-tail | 0.030967272 | |
| t Critical two-tail | 1.992102154 | |

**Table 4.40:** Local vars returned code smell t-test analysis

The code smell has no significant correlation neither with the final mark nor with the number of lines of code of the project. However, the t-test analysis shows a significant difference between the average mark of the students who registered this code smell and the one of those who did not.

### 4.2.20 Code smell: Collapsible "if" statements

#### Code smell explanation

SonarQube states that "Collapsible "if" statements should be merged", as this would improve the code's readability.

#### Code smell example

```java
    public Collection<String> listAppointments(String code, String date) {
        List<String> res = new LinkedList<>();
        for(Map.Entry<String, Appointment> e: apps.entrySet()) {
            Appointment app = e.getValue();
            Doctor d = app.getDoctor();
            if(d.getId().equals(code)) {
                if(app.getSlot().getDate().equals(date)) {
                    res.add(String.format("%s=%s",app.getSlot().getFrom(), app.getPatient().getSSN()));
                }
            }
        }

        return res;
    }
```

In this case the two enclosed `if` statements could be merged using the `&&` operator.

#### Analysis

| Description | Value |
|---|---|
| Total occurrences | 51 |
| Number of projects involved | 28 (12.96%) |
| Correlation with final mark | 0.086141669 |
| Correlation with number of lines of code | 0.120823023 |

**Table 4.41:** Collapsible "if" statements code smell correlation analysis

| | Smell found | Smell not found |
|---|---|---|
| Mean | 22.34142857 | 21.24675532 |
| Variance | 6.536834921 | 20.04441669 |
| Observations | 28 | 188 |
| Hypothesized mean difference | 0 | |
| Alpha | 0.05 | |
| Degrees of freedom | 56 | |
| t Stat | 1.877134861 | |
| P(T<=t) two-tail | 0.065712369 | |
| t Critical two-tail | 2.003240719 | |

**Table 4.42:** Collapsible "if" statements code smell t-test analysis

No significant evidence is found in the analysis.

### 4.2.21 Code smell: Class fields shadowed

#### Code smell explanation

SonarQube's description of the code smell is "Local variables should not shadow class fields": this issue is purely related to the naming choices, but could lead to worse readability or even to bugs, if the student mistakenly thinks they're using one variable but are really using another.

#### Code smell example

```java
public class MedManager {
    private Set<String> accepted = new TreeSet<>();

    // ...

    public double showRate(String code, String date) {
        long accepted = this.appointments.values().stream()
            .filter(s->s.getDate().equals(date))
            .filter(s->s.getDoctor().getId() == code)
            .filter(s->s.getAccepted())
            .count();
        long tot = this.appointments.values().stream()
            .filter(s->s.getDate().equals(date))
            .filter(s->s.getDoctor().getId() == code)
            .count();
        return (double)accepted/tot;
    }
}
```

In this example the `accepted` local variable shadows the homonymous attribute of the class. In this specific case this does not lead to any bug, but it could create confusion for the reader.

**Analysis**

| **Description** | **Value** |
|---|---|
| Total occurrences | 40 |
| Number of projects involved | 29 (13.43%) |
| Correlation with final mark | 0.160043127 |
| Correlation with number of lines of code | 0.049694189 |

**Table 4.43:** Class fields shadowed code smell correlation analysis

| | **Smell found** | **Smell not found** |
|---|---|---|
| Mean | 23.03206897 | 21.13379679 |
| Variance | 12.05702414 | 18.95665271 |
| Observations | 29 | 187 |
| Hypothesized mean difference | 0 | |
| Alpha | 0.05 | |
| Degrees of freedom | 43 | |
| t Stat | 2.639719653 | |
| P(T<=t) two-tail | 0.011513761 | |
| t Critical two-tail | 2.016692199 | |

**Table 4.44:** Class fields shadowed code smell t-test analysis

The analysis reveals that the average mark of the group of students whose code contained this code smell is significantly different from the one of those whose code did not.

### 4.2.22 Code smell: Homonymous class and field

#### Code smell explanation

SonarQube describes the issue stating that "A field should not duplicate the name of its containing class": just like for the previous code smell, this renders the code harder to read, understand and maintain.

### Code smell example

```java
public class Stop {
   private String stop;

    // ...
}
```

### Analysis

| Description | Value |
|---|---|
| Total occurrences | 17 |
| Number of projects involved | 17 (7.87%) |
| Correlation with final mark | -0.019357526 |
| Correlation with number of lines of code | 0.019257481 |

**Table 4.45:** Homonymous class and field code smell correlation analysis

| | Smell found | Smell not found |
|---|---|---|
| Mean | 21.10529412 | 21.41286432 |
| Variance | 18.30146397 | 18.4833266 |
| Observations | 17 | 199 |
| Hypothesized mean difference | 0 | |
| Alpha | 0.05 | |
| Degrees of freedom | 19 | |
| t Stat | -0.284417102 | |
| P(T<=t) two-tail | 0.779167386 | |
| t Critical two-tail | 2.093024054 | |

**Table 4.46:** Homonymous class and field code smell t-test analysis

No significant evidence is revealed by the analysis.

## 4.2.23 Code smell: Field names convention

### Code smell explanation

Regarding this code smell, SonarQube states that "Field names should comply with a naming convention": this issue is usually registered when students capitalize the first letter of a variable's name or use different conventions, such as snake case, or a mix of different conventions.

### Code smell example

```java
public class TrainManager {
    Set<String> Classes = new TreeSet<>();
    TreeMap<String, Car> Vagoni = new TreeMap<>();
    int idFactory=1;
    int idFactory_book=1;
    TreeMap<Integer, Stop> Fermate = new TreeMap<>();
    TreeMap<String, Booking> Prenotazioni = new TreeMap<>();

    // ...
}
```

In this example the `Classes`, `Vagoni`, `idFactory_book`, `Fermate` and `Prenotazioni` variables do not comply with the standard naming conventions for Java.

### Analysis

| Description | Value |
|---|---|
| Total occurrences | 158 |
| Number of projects involved | 60 (27.78%) |
| Correlation with final mark | -0.056263383 |
| Correlation with number of lines of code | 0.105757184 |

**Table 4.47:** Field names convention code smell correlation analysis

| | Smell found | Smell not found |
|---|---|---|
| Mean | 21.28283333 | 21.42935897 |
| Variance | 16.91065794 | 19.06673894 |
| Observations | 60 | 156 |
| Hypothesized mean difference | 0 | |
| Alpha | 0.05 | |
| Degrees of freedom | 113 | |
| t Stat | -0.230508501 | |
| P(T<=t) two-tail | 0.81811346 | |
| t Critical two-tail | 1.981180359 | |

**Table 4.48:** Field names convention code smell t-test analysis

This code smell is found in a large number of projects. However, no significant evidence is registered by the analysis.

### 4.2.24 Code smell: Methods and vars naming

#### Code smell explanation

SonarQube explains this code smell stating that "Local variable and method parameter names should comply with a naming convention". This issue is extremely similar to the previous one, but refers not to class fields but to local variables and method parameters.

#### Code smell example

```java
 public void addPrenotato(String seat, Integer Da, Integer A,
String ssn, String DaN, String AN)
 {
     if(!PostiPrenotati.containsKey(seat))
     {
         //Creo la lista e la aggiungo
         List<Tratta> L = new LinkedList<>();
         L.add(new Tratta(Da, A, ssn, DaN, AN));
         PostiPrenotati.put(seat, L);
     }
     else
     {
         //Aggiungo la tratta alla lista
         PostiPrenotati.get(seat).add(new Tratta(Da, A, ssn, DaN,
AN));
     }

 }
```

In this example the `Da`, `A`, `DaN` and `AN` method parameters, alongside the local variable `L`, do not comply with the standard naming conventions for Java.

#### Analysis

| Description | Value |
|---|---|
| Total occurrences | 166 |
| Number of projects involved | 46 (21.3%) |
| Correlation with final mark | -0.009300424 |
| Correlation with number of lines of code | 0.102641933 |

**Table 4.49:** Methods and vars naming code smell correlation analysis

| | **Smell found** | **Smell not found** |
|---|---|---|
| Mean | 21.84717391 | 21.26458824 |
| Variance | 15.55472739 | 19.18197172 |
| Observations | 46 | 170 |
| Hypothesized mean difference | 0 | |
| Alpha | 0.05 | |
| Degrees of freedom | 78 | |
| t Stat | 0.867522006 | |
| P(T<=t) two-tail | 0.388317191 | |
| t Critical two-tail | 1.990847069 | |

**Table 4.50:** Methods and vars naming code smell t-test analysis

This issue is reported on a considerable portion of the analysed projects; no correlation is in any case found with the final mark and no difference is registered by the t-test analysis.

### 4.2.25 Code smell: Vars declared on same line

#### Code smell explanation

SonarQube describes the code smell stating that "Multiple variables should not be declared on the same line", again to improve readability.

#### Code smell example

```java
public class Reservation {
    private String codF, partenza, arrivo;
    private String nome, cognome;
    private String carId, seat;
    private int codU;

    // ...
}
```

**Analysis**

| Description | Value |
|---|---|
| Total occurrences | 197 |
| Number of projects involved | 64 (29.63%) |
| Correlation with final mark | 0.195613131 |
| Correlation with number of lines of code | 0.108705666 |

**Table 4.51:** Vars declared on same line code smell correlation analysis

| | Smell found | Smell not found |
|---|---|---|
| Mean | 22.4634375 | 20.93611842 |
| Variance | 18.85452768 | 17.62324775 |
| Observations | 64 | 152 |
| Hypothesized mean difference | 0 | |
| Alpha | 0.05 | |
| Degrees of freedom | 115 | |
| t Stat | 2.383689608 | |
| P(T<=t) two-tail | 0.01877768 | |
| t Critical two-tail | 1.980807541 | |

**Table 4.52:** Vars declared on same line code smell t-test analysis

According to the t-test analysis, the difference between the average mark of the students who incurred this issue and the one of those who did not is significant.

### 4.2.26 Code smell: Redundant casts

#### Code smell explanation

SonarQube describes the issue stating that "Redundant casts should not be used": this happens when the type of an expression is already the one it will be cast to, so the cast itself is unnecessary and only renders the code harder to read. In the analysis this is normally found when the student is not sure of the type of the expression or when they do not know that some conversions happen automatically, without the need of an explicit cast.

### Code smell examples

```java
    public class Traincar {
        private int row;
        private char lastSeat;

        // ...

        public int getNumSeats() {
            int a =(int)(lastSeat - 'a');
            return a*row;
        }
    }
```

In this example the student might not have been aware of the fact that subtracting two chars already yields an integer, so the cast is redundant.

```java
    public double showRate(String code, String date) {
        double numApp = (double) appointments.values().stream()
        .filter(a -> a.getDocId().equals(code))
        .filter(x -> x.getDate().equals(date))
        .count();

        double numPres = (double) appointments.values().stream()
                .filter(a -> a.getDocId().equals(code))
                .filter(x -> x.getDate().equals(date))
                .filter(b -> b.getPatient().isAsseganto())
                .count();

        double tasso=numPres/numApp;

        return 1-tasso;
    }
```

In this case the student might not have known that, in an assignment, variables are automatically converted to the type of the variable they are assigned to if the conversion is widening (that is, if the conversion can be performed without losing information): this means that the conversion to `double` ahead of the two assignments is unnecessary.

### Analysis

| Description | Value |
|---|---|
| Total occurrences | 69 |
| Number of projects involved | 32 (14.81%) |
| Correlation with final mark | 0.033936908 |
| Correlation with number of lines of code | -0.099464267 |

**Table 4.53:** Redundant casts code smell correlation analysis

| | Smell found | Smell not found |
|---|---|---|
| Mean | 22.266875 | 21.23592391 |
| Variance | 19.57400282 | 18.13244176 |
| Observations | 32 | 184 |
| Hypothesized mean difference | 0 | |
| Alpha | 0.05 | |
| Degrees of freedom | 42 | |
| t Stat | 1.223312728 | |
| P(T<=t) two-tail | 0.228030968 | |
| t Critical two-tail | 2.018081703 | |

**Table 4.54:** Redundant casts code smell t-test analysis

No evidence is registered by the analysis.

## 4.2.27 Code smell: Redundant bool literals

### Code smell explanation

The description of the code smell provided by SonarQube is "Boolean literals should not be redundant": that is, boolean expressions should not be checked against boolean literals (`true` or `false`) but should be used as is. According to SonarQube this helps improve readability, but arguably, in some situations, the contrary could also be true.

## Code smell example

```java
public class MedManager {
    private Map <String , Medico> medici = new TreeMap<>();

    // ...

    public void addDoctor(String id, String name, String surname,
String speciality) throws MedException {
        if (medici.containsKey(id)==true) {
            throw new  MedException();
        }
        if (specialita.contains(speciality)==false) {
            throw new  MedException();
        }
        Medico m = new Medico(id,name,surname,speciality);
        medici.put(id, m);
    }
}
```

## Analysis

| Description | Value |
|---|---|
| Total occurrences | 100 |
| Number of projects involved | 39 (18.06%) |
| Correlation with final mark | 0.096309203 |
| Correlation with number of lines of code | 0.145990666 |

**Table 4.55:** Redundant bool literals code smell correlation analysis

| | Smell found | Smell not found |
|---|---|---|
| Mean | 22.41717949 | 21.1620339 |
| Variance | 16.35682605 | 18.64828107 |
| Observations | 39 | 177 |
| Hypothesized mean difference | 0 | |
| Alpha | 0.05 | |
| Degrees of freedom | 59 | |
| t Stat | 1.732656093 | |
| P(T<=t) two-tail | 0.088381503 | |
| t Critical two-tail | 2.000995378 | |

**Table 4.56:** Redundant bool literals code smell t-test analysis

The code smell is found in a considerable number of projects, but no evidence is highlighted by the analysis.

### 4.2.28 Code smell: Diamond operator

#### Code smell explanation

SonarQube describes the code smell stating that "The diamond operator ("<>") should be used" to reduce the verbosity of generics code: instead of having to declare a type in both the declaration and the constructor, the constructor declaration can be simplified with `<>`, and the compiler will infer the type.

#### Code smell example

```
 public class MeetServer {
     SortedSet<String> categoriesSet = new TreeSet<String>();
     SortedMap<String,String> meetingsMap = new TreeMap<String,
String>();

     SortedMap<String,String> optionsMap = new TreeMap<String,
String>();
     SortedMap<String, List<String>> showSlotsMap = new TreeMap<
String, List<String>>();

     HashMap<String,Boolean> pollCheckSet = new HashMap<String,
Boolean>();
     SortedMap<String,String> preferencesMap = new TreeMap<String,
String>();

     // ...
 }
```

In this case all the constructors for `TreeMap` and `HashMap` could be simplified with the diamond operator.

**Analysis**

| **Description** | **Value** |
|---|---|
| Total occurrences | 153 |
| Number of projects involved | 45 (20.83%) |
| Correlation with final mark | 0.028438392 |
| Correlation with number of lines of code | -0.00385885 |

**Table 4.57:** Diamond operator code smell correlation analysis

| | **Smell found** | **Smell not found** |
|---|---|---|
| Mean | 21.50755556 | 21.35736842 |
| Variance | 16.8699098 | 18.88778892 |
| Observations | 45 | 171 |
| Hypothesized mean difference | 0 | |
| Alpha | 0.05 | |
| Degrees of freedom | 72 | |
| t Stat | 0.215580208 | |
| P(T<=t) two-tail | 0.829924522 | |
| t Critical two-tail | 1.993463567 | |

**Table 4.58:** Diamond operator code smell t-test analysis

The code smell is registered in a large number of projects, but no significant evidence is highlighted by the analysis.

### 4.2.29 Code smell: Unnecessary imports

#### Code smell explanation

SonarQube states that "Unnecessary imports should be removed" to reduce confusion and improve readability.

### Code smell example

```java
package it.polito.med;

import java.sql.Time;

public class Slot {
    int start;
    int end;
    boolean available = false;
    public Slot(int start, int end) {
        this.start = start;
        this.end = end;
    }
}
```

The `java.sql.Time` class is never used in this file: it was probably used by a prior version of the `Slot` class, but given that it is no longer used its import statement should be removed.

### Analysis

| Description | Value |
|---|---|
| Total occurrences | 163 |
| Number of projects involved | 89 (41.2%) |
| Correlation with final mark | -0.013113072 |
| Correlation with number of lines of code | -0.032166057 |

**Table 4.59:** Unnecessary imports code smell correlation analysis

| | Smell found | Smell not found |
|---|---|---|
| Mean | 21.55651685 | 21.27102362 |
| Variance | 13.97874796 | 21.58419656 |
| Observations | 89 | 127 |
| Hypothesized mean difference | 0 | |
| Alpha | 0.05 | |
| Degrees of freedom | 210 | |
| t Stat | 0.499240028 | |
| P(T<=t) two-tail | 0.618132904 | |
| t Critical two-tail | 1.971324793 | |

**Table 4.60:** Unnecessary imports code smell t-test analysis

The issue is found in a large number of projects, but no significant evidence is registered by the analysis.

### 4.2.30 Code smell: Unused local variables

#### Code smell explanation

Regarding this code smell SonarQube states that "Unused local variables should be removed" to improve readability and remove dead code: this issue is usually registered in methods that are not finished, so perhaps a variable is used to perform some initial computation but the work is never finalized. More rarely, the code smell actually refers to unused variables in an otherwise normal code.

#### Code smell example

```
    public Collection<String> listBookings(String car, String seat) {
        Collection<String> res = new LinkedList<>();
        Vagone vag = vagoni.get(car);

        prenotazioni.forEach((k,v)->{

        });
        return null;
    }
```

This method is unfinished, so all the variables are considered unused.

#### Analysis

| Description | Value |
|---|---|
| Total occurrences | 152 |
| Number of projects involved | 68 (31.48%) |
| Correlation with final mark | -0.102709208 |
| Correlation with number of lines of code | 0.043168332 |

**Table 4.61:** Unused local variables code smell correlation analysis

The code smell is registered in a large number of projects, but no significant evidence is highlighted by the analysis.

|  | **Smell found** | **Smell not found** |
|---|---|---|
| Mean | 20.63029412 | 21.73709459 |
| Variance | 17.13665663 | 18.69912551 |
| Observations | 68 | 148 |
| Hypothesized mean difference | 0 |  |
| Alpha | 0.05 |  |
| Degrees of freedom | 135 |  |
| t Stat | -1.799366133 |  |
| P(T<=t) two-tail | 0.074194403 |  |
| t Critical two-tail | 1.977692277 |  |

**Table 4.62:** Unused local variables code smell t-test analysis

### 4.2.31 Code smell: Empty methods

#### Code smell explanation

SonarQube describes the code smell stating that "Methods should not be empty": while in general this code smell could be registered for several reasons (for example sometimes an intentionally-blank override is needed), in the analysis it is found when students decided not to work on a method whose template was already provided in the teacher's initial project.

#### Code smell example

```
    public void addClasses(String... classes) {

    }
```

This method's signature was included in the template provided by the teacher for the exam: given that the method returns `void`, it was left blank in the template not to provide suggestions on how it ought to be completed. The student then decided not to complete it, thus incurring this code smell.

#### Analysis

The code smell is found in a considerable number of projects. Its number of occurrences is negatively correlated with the final mark (with a medium "effect size"): this means that the more issues of this kind registered, the lower the mark. This makes sense, as registering less of these issues means completing more methods and more requirements (generally speaking, although sometimes students

| Description | Value |
|---|---|
| Total occurrences | 95 |
| Number of projects involved | 56 (25.93%) |
| Correlation with final mark | -0.450191265 |
| Correlation with number of lines of code | -0.112511559 |

**Table 4.63:** Empty methods code smell correlation analysis

| | Smell found | Smell not found |
|---|---|---|
| Mean | 18.22571429 | 22.4956875 |
| Variance | 16.92979221 | 14.2550121 |
| Observations | 56 | 160 |
| Hypothesized mean difference | 0 | |
| Alpha | 0.05 | |
| Degrees of freedom | 90 | |
| t Stat | -6.825089263 | |
| P(T<=t) two-tail | 9.92511E-10 | |
| t Critical two-tail | 1.986674541 | |

**Table 4.64:** Empty methods code smell t-test analysis

start working on a method but do not finish it), thus likely getting a better final evaluation.

The difference between the average mark of the students who incurred this code smell and the one of those who did not is also found to be significant.

### 4.2.32 Code smell: Empty arrays returned

#### Code smell explanation

SonarQube states that "Empty arrays and collections should be returned instead of null": this code smell is registered, just like the previous one, when a student decides not to work on a method whose template was already provided in the teacher's initial project. Differently from the previous issue, this code smell is found in methods that return an object: to avoid complaints from the compiler, the template for the exam cannot be left empty, so not to provide suggestions a `return null` statement is added, statement which is meant to be replaced by the student with one actually returning an object of the specified type.
Therefore again, as per the previous code smell, this issue is not related to the software quality itself, but rather to the completeness of the project.

### Code smell example

```java
    public Collection<String> getClasses() {
        return null;
    }
```

### Analysis

| Description | Value |
|---|---|
| Total occurrences | 299 |
| Number of projects involved | 163 (75.46%) |
| Correlation with final mark | -0.529908661 |
| Correlation with number of lines of code | -0.305823997 |

**Table 4.65:** Empty arrays returned code smell correlation analysis

| | Smell found | Smell not found |
|---|---|---|
| Mean | 20.64159509 | 23.68622642 |
| Variance | 14.88865793 | 22.52486241 |
| Observations | 163 | 53 |
| Hypothesized mean difference | 0 | |
| Alpha | 0.05 | |
| Degrees of freedom | 76 | |
| t Stat | -4.237086242 | |
| P(T<=t) two-tail | 6.29166E-05 | |
| t Critical two-tail | 1.99167261 | |

**Table 4.66:** Empty arrays returned code smell t-test analysis

This code smell is found in the vast majority of projects: differently from the previous issue, which would not be raised if the student even just started working on a method without finishing it, this code smell is found even if the student started working on the method but omitted to substitute the `return null` statement. This explains the difference in the number of projects involved.

The code smell is negatively correlated both with the final mark and with the number of lines of code in the project (with a large and medium "effect size" respectively): this again means that the higher the mark received or the more lines of code produced, the less issues of this kind registered. This is coherent with the idea of students replacing the provided template with actual code: doing so more times means solving more requirements and likely receiving a better final evaluation.

The difference between the average mark of the students who incurred this issue and the one of those who did not is again found to be significant.

### 4.2.33 Code smell: Commented out code

#### Code smell explanation

Regarding this code smell SonarQube states that "Sections of code should not be commented out", as this bloats programs and reduces readability.

#### Code smell example

```java
    public Map<String, List<String>> findSlots(String date, String
    speciality) {
//      dottori.values().stream().filter(d->d.getSpec().equals(
    speciality))
//                      .flatMap(d->d.getList().stream())
//                      .filter(o->o.getData().equals(date))
//                      .collect(Collectors.groupingBy(Doctor::getName));
        Map<String, List<String>> mapp = new HashMap<>();
        for (Doctor d : dottori.values()) {
            if ( d.getSpec() == speciality) {
                List<String> lista = new LinkedList<>();
                for (Orario or : d.getList()) {
                    if (or.getData() == date) {
                        lista.add(or.getIntervallo());
                    }
                }
                mapp.put(d.getId(), lista);
            }
        }
        return mapp;
    }
```

#### Analysis

| Description | Value |
|---|---|
| Total occurrences | 139 |
| Number of projects involved | 66 (30.56%) |
| Correlation with final mark | 0.006499403 |
| Correlation with number of lines of code | 0.003817572 |

**Table 4.67:** Commented out code code smell correlation analysis

| | **Smell found** | **Smell not found** |
|---|---|---|
| Mean | 21.47378788 | 21.3512 |
| Variance | 16.51621774 | 19.32725358 |
| Observations | 66 | 150 |
| Hypothesized mean difference | 0 | |
| Alpha | 0.05 | |
| Degrees of freedom | 134 | |
| t Stat | 0.199101269 | |
| P(T<=t) two-tail | 0.842485318 | |
| t Critical two-tail | 1.977825758 | |

**Table 4.68:** Commented out code code smell t-test analysis

The code smell is registered in a large number of projects, but no significant evidence is highlighted by the analysis.

## 4.3 Summary of results

To sum up, a few interesting results have been found in the analysis. These findings compose a general picture that will be discussed in the final chapter.

### 4.3.1 Issues correlation with the final mark and with the number of lines of code

Firstly, only the occurrences of few issues are correlated with the final evaluation received or with the number of lines of code of the student's project. Specifically, two issues have a medium correlation and one has a strong correlation with the final mark, and two of these three issues also have a medium correlation with the number of lines of code.

| Issue type | Issue | Correlation with final mark | Correlation type |
|---|---|---|---|
| Code smell | "Stream.toList()" method should be used when unmodifiable list needed | 0.446797723 | Medium |
| Code smell | Methods should not be empty | -0.450191265 | Medium |
| Code smell | Empty arrays and collections should be returned instead of null | -0.529908661 | Strong |

**Table 4.69:** Single issues correlation with final mark

| Issue type | Issue | Correlation with # of lines | Correlation type |
|---|---|---|---|
| Code smell | "Stream.toList()" method should be used when unmodifiable list needed | 0.325362376 | Medium |
| Code smell | Empty arrays and collections should be returned instead of null | -0.305823997 | Medium |

**Table 4.70:** Single issues correlation with number of lines of code

All the analysed bugs and code smells are credible as issues produced by students at their initial approach with programming or, specifically, with object-oriented programming; even more so considering the time constraints of the exam: many of these issues, the bugs at least, could be identified by testing the program on a wider range of inputs and debugging it, but the time constraints for the exam and the students inexperience may represent an obstacle in doing so. However, almost no issue has a correlation with the final mark: this can explained by the fact that the final mark is determined almost uniquely by the number of passed tests, that do not consider the quality of the software. In fact, the three correlated issues are code smells: although being more severe, bugs have no correlation with the evaluation.

There is only one positively correlated (both with the final mark and with the number of lines of code) code smell, which is anyway related to a quite specific issue, perhaps outside the interest of novice students: this could also explain the positivity of the correlation. The more lines of code written, the more likely a student is to produce an issue they might even not know the existence of.

On the contrary, as previously mentioned, the negative correlations derive from the structure of the exam: at the start of the exam students are provided with a template for all methods and classes they are supposed to implement. These methods usually are set to return `null`, regardless of the return type, if they are meant to return an object (so that the code compiles but no indications are provided on how they ought to be completed), or they are left empty if no return value is needed. This means that the higher the mark a student receives, the more requirements they have solved, the more of these methods they have changed from this issue-raising initial version to a more complete one.

### 4.3.2 Difference between the means of issue found and issue not found groups

The second kind of analysis, performed using the t-test statistic, aimed at highlighting the significant differences between the average mark of the students who incurred a specific issue and the one of those who did not. The test found 9 issues where this difference is significant, i.e. where the P(T<=t) two-tail value is less than alpha, the chosen significance level, whose value is 0.05.

| **Issue type** | **Issue** | **P(T<=t) two-tail** | **Average mark** | |
|---|---|---|---|---|
| | | | **Issue found** | **Issue not found** |
| Bug | Zero should not be a possible denominator | 0.001827194 | 24.1335 | 21.10857143 |
| Code smell | Cognitive Complexity of methods should not be too high | 0.034694501 | 22.79709677 | 21.15264865 |
| Code smell | "Stream.toList()" method should be used when unmodifiable list needed | 3.62281E-07 | 22.39086667 | 19.11090909 |
| Code smell | Lambdas should be replaced with method references | 9.90753E-05 | 22.995625 | 20.71203947 |
| Code smell | Local variables should not be declared and then immediately returned or thrown | 0.030967272 | 20.20148936 | 21.71881657 |
| Code smell | Local variables should not shadow class fields | 0.011513761 | 23.03206897 | 21.13379679 |
| Code smell | Multiple variables should not be declared on the same line | 0.01877768 | 22.4634375 | 20.93611842 |
| Code smell | Methods should not be empty | 9.92511E-10 | 18.22571429 | 22.4956875 |
| Code smell | Empty arrays and collections should be returned instead of null | 6.29166E-05 | 20.64159509 | 23.68622642 |

**Table 4.71:** Single issues t-test analysis

This being a two-tailed t-test, nothing can be said regarding which mean is statistically higher than the other: it can be noted, however, that either these issues had an impact on the final mark or, vice versa, the variation in the received mark had an influence in finding more or less of said bugs and code smells.

# Chapter 5

# Conclusions

Getting to the end of this thesis, it is possible to draw some conclusions on the work that has been done and on possible future further developments.

The work done throughout the thesis has proven useful for two different necessities and contexts: for students and for teachers.

## 5.1 The students perspective

For the students, the existing pipeline that builds and tests the programming assignments has been expanded with a new stage that verifies the software quality.

Possible future developments could be, at first, to design a way for students to access the results of the analysis. This was not done as the goal of the thesis shifted towards understanding whether the concepts of technical debt and static program analysis could generally be applied in this context, and this of course had to be done from a teacher's perspective first and then proposed to the students only after properly investigating the idea.

Letting the students access the results of the analysis initiated by the pipeline would not be a trivial matter: at first the problems of authentication and authorization would need to be solved. Every student would need to have a set of credentials to access the information, either the ones provided by the university (in which case an integration with the university login system would need to be developed) or specific ones (in which case the problem of creating and delivering the credentials would be posed). Then, after logging in, each student would need to be able to access only the results relative to their analyses, and not the ones of the other students. It then should be evaluated whether to give students access directly to SonarQube or to develop a specific application: this would feasible using SonarQube's APIs and could allow to better customize the information displayed to the student that, being new to technical debt, static program analysis and code smells, could probably benefit from a simplified environment.

After this first step, a further analysis could be conducted to understand how students absorb these changes, whether a full analysis report is too difficult to comprehend and whether technical debt is actually useful in pointing out deficiencies in the preparation and in understanding how to solve them.

Contextually, it should be verified whether all rules SonarQube checks against are appropriate for a student at their first approach to programming. This thesis only selected the commonest rules to be analysed, all of which were deemed quite related to the context, but it should probably be verified how every rule is interpreted and understood by the students.

## 5.2 The teachers perspective

It is on the teachers side where this thesis focused the most and revealed interesting results.

Firstly, automating a massive static program analysis revealed to be beneficial to the teachers. The commonest problems, where students could probably benefit from additional help, were in fact highlighted: all the found bugs and code smells, or at least the most widely spread ones, are credible as issues produced by students at their initial approach with programming or, specifically, with object-oriented programming; even more so considering the time constraints of the exam.
This kind of analysis can be an important tool to evaluate the students preparation, both throughout the course and at the final exam. Teachers can run this kind of analysis rather quickly and can find out what elements might need to be revised in class.

It might be interesting, as a possible future further investigation, to ask a significant group of teachers to test this idea and to report their feedback, to try and understand what is the real experience with applying this idea "in the wild". This would also help understand whether all rules SonarQube uses are suitable for an analysis of this kind, although it has to be said that, differently from the students, teachers will likely have no problem in quickly distinguishing which rules are of any interest and which ones are not relevant to their use case.

Subsequently, an in-depth statistical analysis was performed on the data retrieved from a real exam analysis: generally speaking, it can be said that the final mark is not correlated to the quality of the software. This is because the evaluation is mostly performed using automated tests that simply were not written to verify the quality of the software.

A change that could be made in the future, after developing a way to let students peruse their analyses, could be to award an assignment with great software quality (so, with few bugs and code smells) with bonus points. This could probably be only a small component of the final mark: it should be considered whether achieving good software quality could be introduced as a goal of the course or only as a bonus, if the course is still early on in the students approach to programming. This would in any case help increase the correlation between the number of bugs and code smells found in code and the final evaluation received by the students.

## 5.3 In the end

To conclude, this thesis showed that applying the concept of technical debt to students' programming assignments can be done and can prove useful. Specifically, the real benefit is not found in the concept of technical debt itself, which is a metaphor much more useful in large companies to describe the imperfections found in code and the initial lack of and then ever-growing understanding of the problem at hand, but in applying the same methods used in said companies to recognize technical debt to students' programming assignments to identify the major issues and to guide hypothesis on how to address them: these modalities proved beneficial in highlighting the commonest problems amongst the students and in painting a general picture of the students situation.

After proving the usefulness of this approach, now a wider and deeper integration could be carried out in a more thorough way, introducing the students too to these concepts and methodologies, proposing it to more teachers of different courses and subjects or including the considerations gathered from the analysis in the final evaluation.

# Appendix A

# Scripts and code used

## A.1 Initial files for students' repositories CI/CD

These are the initial files for running the CI/CD pipeline, prior to any modification.

Initial gitlab-ci.yml

```
# Build JAVA applications using Apache Maven (http://maven.apache.org)
# For docker image tags see https://hub.docker.com/_/maven/
#
# For general lifecycle information see https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html

variables:
  # This will suppress any download for dependencies and plugins or upload messages which would clutter the console log.
  # `showDateTime` will show the passed time in milliseconds. You need to specify `--batch-mode` to make this work.
  MAVEN_OPTS: "-Dhttps.protocols=TLSv1.2 -Dmaven.repo.local=$CI_PROJECT_DIR/.m2/repository -Dorg.slf4j.simpleLogger.log.org.apache.maven.cli.transfer.Slf4jMavenTransferListener=WARN -Dorg.slf4j.simpleLogger.showDateTime=true -Djava.awt.headless=true"
  # As of Maven 3.3.0 instead of this you may define these options in `.mvn/maven.config` so the same config is used
  # when running from the command line.
  # `installAtEnd` and `deployAtEnd` are only effective with recent version of the corresponding plugins.
  MAVEN_CLI_OPTS: "--batch-mode --errors --fail-at-end --show-version -DinstallAtEnd=true -DdeployAtEnd=true"

# This template uses jdk11 for verifying and deploying images
image: maven:3.8.3-adoptopenjdk-11
```

```

# Cache downloaded dependencies and plugins between builds.
# To keep cache across branches add 'key: "$CI_JOB_NAME"'
cache:
  paths:
    - .m2/repository

verify:
  stage: test
  script:
    - 'mvn $MAVEN_CLI_OPTS verify'
  artifacts:
    when: always
    reports:
      junit:
        - target/surefire-reports/TEST-*.xml
        - target/failsafe-reports/TEST-*.xml
```

Initial pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>lab.oop.polito.it</groupId>
    <artifactId>OOP_LAB_DietExtended_sol</artifactId>
    <version>0.0.1-SNAPSHOT</version>

    <dependencies>
        <!-- https://mvnrepository.com/artifact/junit/junit -->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.13.2</version>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <sourceDirectory>src</sourceDirectory>
        <testSourceDirectory>test</testSourceDirectory>
        <plugins>
            <plugin>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.8.1</version>
                <configuration>
                    <release>11</release>
                    <encoding>UTF-8</encoding>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>2.22.0</version>
                <dependencies>
                    <dependency>
                        <groupId>org.apache.maven.surefire</groupId>
                        <artifactId>surefire-junit4</artifactId>
                        <version>2.22.0</version>
                    </dependency>
                </dependencies>
                <configuration>
                    <includes>
                        <include>**/*.java</include>
                    </includes>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-site-plugin</artifactId>
                <version>3.9.1</version>
                <configuration>
                </configuration>
            </plugin>
        </plugins>
    </build>
    <reporting>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-surefire-report-plugin</artifactId>
                <version>2.22.0</version>
            </plugin>
        </plugins>
    </reporting>
</project>
```

## A.2 Modified files for students' repositories CI/CD

These are the modified files for running the CI/CD pipeline, including an added second stage for the static program analysis with SonarQube.

Modified gitlab-ci.yml

```
stages:
  - test
  - codequality

variables:
  MAVEN_OPTS: >-
    -Dhttps.protocols=TLSv1.2
    -Dorg.slf4j.simpleLogger.showDateTime=true
    -Djava.awt.headless=true
    -XX:+DisableAttachMechanism

  MAVEN_CLI_OPTS: >-
    --batch-mode
    --errors
    --fail-at-end
    --show-version
    --no-transfer-progress

image: maven:3.8.3-adoptopenjdk-11

verify:
  stage: test
  tags:
    - oop
  script:
    - 'mvn $MAVEN_CLI_OPTS test'
  artifacts:
    when: always
    reports:
      junit:
        - target/surefire-reports/TEST-*.xml

sonarqube:
  stage: codequality
  tags:
    - oop
  variables:
    SONAR_HOST_URL: "http://172.17.0.2:9000"
    SONAR_TOKEN: "85df9fae1b73e56abee20330db2528e5cd9a6e9a"
    SONAR_USER_HOME: "${CI_PROJECT_DIR}/.sonar"  # Defines the
      location of the analysis task cache
  script:
    - mvn clean install sonar:sonar
```

Modified pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>it.polito.oop.lab</groupId>
    <artifactId>University-Solution</artifactId>
    <version>1.0.0</version>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <sonar.core.codeCoveragePlugin>jacoco</sonar.core.codeCoveragePlugin>
        <sonar.jacoco.reportPath>${project.basedir}/../target/jacoco.exec</sonar.jacoco.reportPath>
        <sonar.language>java</sonar.language>
    </properties>

    <dependencies>
        <!-- https://mvnrepository.com/artifact/junit/junit -->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.13.2</version>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <sourceDirectory>src</sourceDirectory>
        <testSourceDirectory>test</testSourceDirectory>
        <plugins>
            <plugin>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.8.1</version>
                <configuration>
                    <release>11</release>
                </configuration>
            </plugin>

            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>2.22.0</version>
                <dependencies>
                    <dependency>
                        <groupId>org.apache.maven.surefire</groupId>
```

```xml
                        <artifactId>surefire-junit4</artifactId>
                        <version>2.22.0</version>
                    </dependency>
                </dependencies>
                <configuration>
                    <includes>
                        <include>**/*.java</include>
                    </includes>
                </configuration>
            </plugin>

            <plugin>
                <groupId>org.sonarsource.scanner.maven</groupId>
                <artifactId>sonar-maven-plugin</artifactId>
                <version>3.7.0.1746</version>
            </plugin>
        </plugins>
    </build>
</project>
```

## A.3 Files for the batch analysis of an assignment

### A.3.1 Multi-project batch analysis

For the multi-project batch analysis approach, the previously modified *pom.xml* file is needed, alongside a *sonar-project.properties* configuration file.

sonar-project.properties

```
    sonar.projectKey=Multiproject-LAB03-Diet

    sonar.sources=LAB03_Diet_s000001/src/diet,LAB03_Diet_s000002/src/diet,LAB03_Diet_s000003/src/diet
```

### A.3.2 Multi-module batch analysis

For the multi-module batch analysis approach, the parent POM file resides inside the parent folder, whereas the child POM file resides inside every submodule folder. The Bash script automates the setup of the POM files, compiles the project and runs the analysis.
The SQL script, on the other hand, retrieves all students numbers and the occurrences of the bug or code smell being considered for each student: this is helpful to retrieve how many occurrences of every bug or code smell were found for every student.

parent_pom.xml

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <packaging>pom</packaging>

  <groupId>it.polito.oop</groupId>
  <artifactId>parentArtifactId</artifactId>
  <version>1.0.0</version>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <sonar.language>java</sonar.language>
    <sonar.java.source>17</sonar.java.source>
    <sonar.host.url>http://localhost:9000</sonar.host.url>

    <!-- exclude all files containing @Test (test files) -->
    <sonar.issue.ignore.allfile>r1</sonar.issue.ignore.allfile>
    <sonar.issue.ignore.allfile.r1.fileRegexp>@Test</sonar.issue.ignore.allfile.r1.fileRegexp>

    <!-- exclude additional files -->
    <sonar.exclusions>**.html,**.xml</sonar.exclusions>

    <!-- exclude code duplication and coverage -->
    <sonar.cpd.exclusions>**</sonar.cpd.exclusions>
    <sonar.coverage.exclusions>**</sonar.coverage.exclusions>

    <!-- disable Source Control Manager -->
    <sonar.scm.disabled>true</sonar.scm.disabled>
  </properties>

  <modules>
    <module>module</module>
  </modules>

</project>
```

child_pom.xml

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <parent>
        <groupId>it.polito.oop</groupId>
        <artifactId>parentArtifactId</artifactId>
        <version>1.0.0</version>
    </parent>

    <groupId>it.polito.oop</groupId>
    <artifactId>childArtifactId</artifactId>
    <version>1.0.0</version>

    <dependencies>
        <!-- https://mvnrepository.com/artifact/junit/junit -->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.13.2</version>
        </dependency>
    </dependencies>

    <build>
        <sourceDirectory>.</sourceDirectory>
        <plugins>
            <plugin>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.8.1</version>
                <configuration>
                    <release>11</release>
                </configuration>
            </plugin>

            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>2.22.0</version>
                <dependencies>
                    <dependency>
                        <groupId>org.apache.maven.surefire</groupId>
                        <artifactId>surefire-junit4</artifactId>
                        <version>2.22.0</version>
                    </dependency>
                </dependencies>
                <configuration>
                    <includes>
                        <include>**/*.java</include>
                    </includes>
                </configuration>
            </plugin>

            <plugin>
                <groupId>org.sonarsource.scanner.maven</groupId>
                <artifactId>sonar-maven-plugin</artifactId>
                <version>3.7.0.1746</version>
            </plugin>
        </plugins>
    </build>

</project>
```

script.sh

```
#! /bin/bash

parentArtifactId=$1
sonarToken=$2

if [ "$parentArtifactId" = "" ]; then
    echo "Missing project name in arguments."
    exit
fi

if [ "$parentArtifactId" = "parentArtifactId" ]; then
    echo "Project name not allowed."
    exit
fi

if [ "$sonarToken" = "" ]; then
    echo "Missing SonarQube token in arguments."
    exit
fi

echo "Preparing the necessary POM files to build the submodules..."

# substitute the artifactId in the parent pom
sed "s|<artifactId>parentArtifactId</artifactId>|<artifactId>
    $parentArtifactId</artifactId>|" ./templates/parent_pom.xml >pom.
    xml

# find only directories in current folder, excluding hidden ones
subfolders=$(find . -mindepth 1 -maxdepth 1 -type d -regex '
    \./[^\.].*')

for dir in $subfolders; do
    if [ "$dir" = "./target" ] || [ "$dir" = "./templates" ]; then
        continue
    fi

```

```
34     childArtifactId=$(echo "$dir" | cut -d '/' -f 2)    # cut the string on / and take the second field (the folder name)
35 
36     if [ "$childArtifactId" = "childArtifactId" ] || [ "$childArtifactId" = "module" ]; then
37         echo "Subfolder name $childArtifactId not allowed."
38         exit
39     fi
40 
41     # create the child pom
42     cd "$dir" || {
43         echo "Directory $dir not found!"
44         exit
45     }
46     sed "s|<artifactId>parentArtifactId</artifactId>|<artifactId>$parentArtifactId</artifactId>|" ../templates/child_pom.xml >tmp.xml
47     sed "s|<artifactId>childArtifactId</artifactId>|<artifactId>$childArtifactId</artifactId>|" tmp.xml >pom.xml
48     rm tmp.xml
49     cd ..
50 
51     # add the submodule to the parent pom
52     cp pom.xml tmp.xml
53     sed "s|<module>module</module>|<module>$childArtifactId</module>\n    <module>module</module>|" tmp.xml >pom.xml
54     rm tmp.xml
55 done
56 
57 # finalize parent pom
58 cp pom.xml tmp.xml
59 sed "/<module>module<\/module>/d" tmp.xml >pom.xml
60 rm tmp.xml
61 
62 # run analysis
63 echo "Building and then running the analysis..."
64 mvn clean install --fail-never sonar:sonar -Dsonar.token="$sonarToken"  # builds all the submodules (skipping the ones that fail) and then launches the analysis
```

This script is intended to work with version 10 of SonarQube and subsequent ones: for prior versions, the `-Dsonar.token` parameter at line 64 should be substituted with `-Dsonar.login`.

The command to launch the script needs, as parameters, the parent's artifact id and the SonarQube token for authentication.

```
    script.sh parentArtifactId sonarToken
```

issues_per_student_by_rule.sql

```sql
-- select all students numbers (having at least one issue), then left join to append number of issues of the student (if present)
SELECT DISTINCT SUBSTRING(kee, 31, 7) AS student, issues_per_student.count
FROM components
LEFT JOIN (
    -- select student number and number of issues for every student for a specific rule
    -- the components table contains the name of the folder (the component, in the column kee): that is, the student number
    SELECT SUBSTRING(C.kee, 31, 7) AS student, COUNT(*) AS count
    FROM issues I, components C
    WHERE I.component_uuid = C.uuid AND C.kee LIKE 'it.polito.oop:Exam_2023-06-27:s%'
      AND rule_uuid = 'AYgvIl4PPrvCAqFZdVNH'
    GROUP BY SUBSTRING(C.kee, 31, 7)
) AS issues_per_student ON SUBSTRING(components.kee, 31, 7) = student
WHERE kee LIKE 'it.polito.oop:Exam_2023-06-27:s%'
ORDER BY student
```

# Appendix B

# GitHub repository

The GitHub repository for this thesis contains files, code and data used throughout the work. The repository is located at `https://github.com/alessiomason/masters-degree-thesis`.

#### Folders content

The "Initial analysis" folder contains a short presentation and summary of the gathered data and a subfolder containing all the projects analyzed for this initial phase.

The "Multi-project analysis" folder contains the commands to create the Docker container hosting SonarQube and to run the analysis, in addition to a folder used for testing the hierarchy and placement of folders and files needed for this solution.

The "Multi-module analysis" folder again contains a test folder (with all subfolders and files correctly placed to verify the viability of this solution), the templates for the parent and child POM, the script to automate the analysis and the command to launch it.

The "Exam 2023-06-27 analysis" folder contains a presentation of the major bugs and code smells found in the analysis, the SQL scripts employed to retrieve the data from the database SonarQube uses and two Excel files summarizing all gathered data, which were used to perform the statistical analysis - one file containing all the analysed students and the other only the ones who actually received an evaluation for the exam.
The actual folder containing all the analysed projects is not included due to privacy concerns for the students (in the Excel files the students numbers are anonymized).

# Bibliography

[1] Flavien Huynh. *Software quality: Origin story.* Ed. by Coders Kitchen. June 8, 2020. URL: `https://www.coderskitchen.com/software-quality-origin-story/` (visited on 10/09/2023) (cit. on p. 4).

[2] Abel Avram. *Ensuring Product Quality at Google.* Ed. by InfoQ. Mar. 11, 2011. URL: `https://www.infoq.com/news/2011/03/Ensuring-Product-Quality-Google/` (visited on 10/09/2023) (cit. on p. 4).

[3] Love Sharma. *10 nonfunctional requirements to consider in your enterprise architecture.* Ed. by Red Hat. Aug. 4, 2022. URL: `https://www.redhat.com/architect/nonfunctional-requirements-architecture` (visited on 10/09/2023) (cit. on p. 4).

[4] Agile Alliance, ed. *What is Test Driven Development (TDD)?* URL: `https://www.agilealliance.org/glossary/tdd/` (visited on 10/11/2023) (cit. on p. 5).

[5] Don Wells. *Code the Unit Test First.* Ed. by Extreme Programming. 2000. URL: `http://www.extremeprogramming.org/rules/testfirst.html` (visited on 10/11/2023) (cit. on p. 5).

[6] Luís Soares. *Our approach to quality at Volkswagen Software Dev Center Lisbon.* Ed. by Medium. Sept. 18, 2019. URL: `https://lsoares.medium.com/our-approach-to-quality-in-volkswagen-software-dev-center-lisbon-72f5728e2235` (visited on 10/09/2023) (cit. on p. 6).

[7] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. *Manifesto for Agile Software Development.* Ed. by Agile Alliance. Feb. 13, 2001. URL: `http://agilemanifesto.org` (visited on 10/09/2023) (cit. on p. 6).

[8] Kai Petersen, Claes Wohlin, and Dejan Baca. «The Waterfall Model in Large-Scale Development». In: *Product-Focused Software Process Improvement.* Ed. by Frank Bomarius, Markku Oivo, Päivi Jaring, and Pekka Abrahamsson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 386–400. ISBN: 978-3-642-02152-7 (cit. on p. 6).

[9] Ondrej Burkacky, Johannes Deichmann, Stefan Frank, Dominik Hepp, and André Rocha. *When code is king: Mastering automotive software excellence.* Ed. by McKinsey & Company. Feb. 17, 2021. URL: `https://www.mckinsey.com/industries/automotive-and-assembly/our-insights/when-code-is-king-mastering-automotive-software-excellence` (visited on 10/13/2023) (cit. on p. 6).

[10] Terence Parr. *Extreme Programming.* Ed. by University of San Francisco. Nov. 29, 2006. URL: `https://www.cs.usfca.edu/~parrt/course/601/lectures/xp.html` (visited on 10/11/2023) (cit. on p. 6).

[11] CertiProf, ed. *Agile Adoption Report 2022.* Slides 17-18. 2022. URL: `https://app.hubspot.com/documents/4122918/view/441544638?accessId=4d36b9` (visited on 10/13/2023) (cit. on p. 7).

[12] Sean Gallagher. *Airbus confirms software configuration error caused plane crash.* Ed. by Ars Technica. June 1, 2015. URL: `https://arstechnica.com/information-technology/2015/06/airbus-confirms-software-configuration-error-caused-plane-crash/` (visited on 10/09/2023) (cit. on p. 7).

[13] Edgar Alvarez. *To keep a Boeing Dreamliner flying, reboot once every 248 days.* Ed. by Engadget. July 19, 2019. URL: `https://www.engadget.com/2015-05-01-boeing-787-dreamliner-software-bug.html` (visited on 10/09/2023) (cit. on p. 7).

[14] James Billington. *Nest not working: Smart thermostat bug plunges customers into cold.* Ed. by International Business Times. Jan. 14, 2016. URL: `https://www.ibtimes.co.uk/google-owned-nest-thermostat-plunges-customers-into-cold-after-software-glitch-1537979` (visited on 10/09/2023) (cit. on p. 7).

[15] Mark Dowson. «The Ariane 5 Software Failure». In: *SIGSOFT Softw. Eng. Notes* 22.2 (Mar. 1997), p. 84. ISSN: 0163-5948. DOI: `10.1145/251880.251992` (cit. on p. 7).

[16] J.-M. Jazequel and B. Meyer. «Design by contract: the lessons of Ariane». In: *Computer* 30.1 (1997), pp. 129–130. DOI: `10.1109/2.562936` (cit. on p. 7).

[17] Nathaniel Popper. *Knight Capital Says Trading Glitch Cost It $440 Million.* Ed. by The New York Times. Aug. 2, 2012. URL: https://archive.nytimes.com/dealbook.nytimes.com/2012/08/02/knight-capital-says-trading-mishap-cost-it-440-million/ (visited on 10/09/2023) (cit. on p. 7).

[18] Bengt Halvorson. *Software Now To Blame For 15 Percent Of Car Recalls.* Ed. by Popular Science. June 2, 2016. URL: https://www.popsci.com/software-rising-cause-car-recalls/ (visited on 10/09/2023) (cit. on p. 7).

[19] UBM Tech Electronics, ed. *2013 Embedded Market Study.* Slide 24. 2013. URL: https://www.bogotobogo.com/cplusplus/files/embed/EmbeddedMarketStudy_2013.pdf (visited on 10/13/2023) (cit. on p. 8).

[20] Chris Grams. *How Much Time Do Developers Spend Actually Writing Code?* Ed. by The New Stack. Oct. 15, 2019. URL: https://thenewstack.io/how-much-time-do-developers-spend-actually-writing-code/ (visited on 10/13/2023) (cit. on p. 8).

[21] Philip Koopman. *Embedded System Software Quality.* Ed. by Edge Case Research. ISSRE 2016 Keynote. Oct. 25, 2016. URL: https://web.archive.org/web/20201127153133/https://users.ece.cmu.edu/~koopman/lectures/2016_issre.pdf (visited on 10/13/2023) (cit. on p. 8).

[22] Esko Hannula. *A short history of software quality.* Ed. by LinkedIn. Nov. 22, 2016. URL: https://www.linkedin.com/pulse/short-history-software-quality-esko-hannula (visited on 10/13/2023) (cit. on p. 8).

[23] Cunningham & Cunningham, Inc. Wiki, ed. *WyCash.* URL: http://wiki.c2.com/?WyCash (visited on 10/13/2023) (cit. on p. 8).

[24] Ward Cunningham. *The WyCash Portfolio Management System.* Ed. by Cunningham & Cunningham, Inc. OOPSLA '92 Experience Report. Mar. 26, 1992. URL: http://c2.com/doc/oopsla92.html (visited on 10/13/2023) (cit. on p. 8).

[25] Ward Cunningham. *Ward Explains Debt Metaphor.* Ed. by Cunningham & Cunningham, Inc. Wiki. Feb. 15, 2009. URL: http://wiki.c2.com/?WardExplainsDebtMetaphor (visited on 10/13/2023) (cit. on p. 9).

[26] Cunningham & Cunningham, Inc. Wiki, ed. *Technical Debt.* URL: http://wiki.c2.com/?TechnicalDebt (visited on 10/13/2023) (cit. on p. 9).

[27] Jim Highsmith. *Zen & the Art of Software Quality.* Ed. by Information Architects, Inc. Slide 20. 2008. URL: https://vdocuments.mx/2008-information-architects-inc-1-zen-the-art-of-software-quality-jim.html?page=9 (visited on 10/13/2023) (cit. on p. 9).

[28] Cunningham & Cunningham, Inc. Wiki, ed. *Complexity As Debt.* July 25, 2001. URL: `http://wiki.c2.com/?ComplexityAsDebt` (visited on 10/13/2023) (cit. on p. 10).

[29] G. Suryanarayana, G. Samarthyam, and T. Sharma. *Refactoring for Software Design Smells: Managing Technical Debt.* Elsevier Science, 2014. ISBN: 9780128016466. URL: `https://books.google.it/books?id=1SaOAwAAQBAJ` (cit. on p. 10).

[30] Stephen Chong. *Dynamic Analysis.* Ed. by Harvard John A. Paulson School of Engineering and Applied Sciences. Oct. 1, 2015. URL: `https://groups.seas.harvard.edu/courses/cs252/2015fa/lectures/Lec08-DynamicAnalysis.pdf` (visited on 10/14/2023) (cit. on p. 11).

[31] Anjana Gosain and Ganga Sharma. «A Survey of Dynamic Program Analysis Techniques and Tools». In: *Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA) 2014.* Ed. by Suresh Chandra Satapathy, Bhabendra Narayan Biswal, Siba K. Udgata, and J.K. Mandal. Cham: Springer International Publishing, 2015, pp. 113–122. ISBN: 978-3-319-11933-5 (cit. on pp. 11, 12).

[32] Saket Khatiwada, Miroslav Tushev, and Anas Mahmoud. «Just enough semantics: An information theoretic approach for IR-based software bug localization». In: *Information and Software Technology* 93 (2018), pp. 45–57. ISSN: 0950-5849. DOI: `https://doi.org/10.1016/j.infsof.2017.08.012`. URL: `https://www.sciencedirect.com/science/article/pii/S0950584916302269` (cit. on p. 12).

[33] Mark Weiser. «Program Slicing». In: *Proceedings of the 5th International Conference on Software Engineering.* ICSE '81. San Diego, California, USA: IEEE Press, 1981, pp. 439–449. ISBN: 0897911466 (cit. on p. 12).

[34] Barton P. Miller, Lars Fredriksen, and Bryan So. «An Empirical Study of the Reliability of UNIX Utilities». In: *Commun. ACM* 33.12 (Dec. 1990), pp. 32–44. ISSN: 0001-0782. DOI: `10.1145/96267.96279`. URL: `https://doi.org/10.1145/96267.96279` (cit. on p. 12).

[35] Michael D Ernst. «Static and dynamic analysis: Synergy and duality». In: *WODA 2003: ICSE Workshop on Dynamic Analysis.* 2003, pp. 24–27 (cit. on pp. 12, 13).

[36] Flemming Nielson, Hanne Nielson, and Chris Hankin. *Principles of Program Analysis.* Jan. 1999. ISBN: 978-3-642-08474-4. DOI: `10.1007/978-3-662-03811-6` (cit. on pp. 12, 13).

[37] N. Jovanovic, C. Kruegel, and E. Kirda. «Pixy: a static analysis tool for detecting Web application vulnerabilities». In: *2006 IEEE Symposium on Security and Privacy (S&P'06)*. Berkeley/Oakland, CA, USA, 2006, pp. 6 pp.–263. DOI: 10.1109/SP.2006.29 (cit. on p. 12).

[38] Patrick Cousot and Radhia Cousot. «Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints». In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '77. Los Angeles, California: Association for Computing Machinery, 1977, pp. 238–252. ISBN: 9781450373500. DOI: 10.1145/512950.512973. URL: https://doi.org/10.1145/512950.512973 (cit. on p. 13).

[39] B.C. Pierce. *Types and Programming Languages*. The MIT Press. MIT Press, 2002. ISBN: 9780262162098. URL: https://books.google.it/books?id=ti6zoAC9Ph8C (cit. on p. 13).

[40] Ranjit Jhala and Rupak Majumdar. «Software model checking». In: *ACM Computing Surveys (CSUR)* 41.4 (2009), pp. 1–54 (cit. on p. 13).

[41] Martin Fowler. *CodeSmell*. Feb. 9, 2006. URL: https://martinfowler.com/bliki/CodeSmell.html (visited on 10/16/2023) (cit. on p. 14).

[42] Andrew Binstock. *In Praise Of Small Code*. Ed. by InformationWeek. June 21, 2011. URL: https://www.informationweek.com/it-sectors/in-praise-of-small-code (visited on 10/16/2023) (cit. on p. 14).

[43] M. Fowler and K. Beck. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley object technology series. Addison-Wesley, 1999. ISBN: 9780201485677. URL: https://books.google.it/books?id=UTgFCAAAQBAJ (cit. on p. 14).

[44] TrustInSoft, ed. *How Exhaustive Static Analysis Will Make Next-Generation Automotive Software Safer, More Reliable, and More Secure*. Mar. 22, 2022. URL: https://trust-in-soft.com/blog/2022/03/22/static-analysis-for-automotive/ (visited on 10/16/2023) (cit. on p. 15).

[45] Marco Torchiano and Giorgio Bruno. «Integrating Software Engineering Key Practices into an OOP Massive In-Classroom Course: An Experience Report». In: *2018 IEEE/ACM International Workshop on Software Engineering Education for Millennials (SEEM)*. 2018, pp. 64–71 (cit. on pp. 17, 18).

[46] Politecnico di Torino, ed. *Object oriented programming syllabus*. URL: https://didattica.polito.it/pls/portal30/gap.pkg_guide.viewGap?p_cod_ins=04JEYOA&p_a_acc=2020&p_lang=EN (visited on 10/17/2023) (cit. on p. 17).

[47] Apache Maven Project, ed. *Maven - Introduction.* URL: https://maven.apache.org/what-is-maven.html (visited on 10/18/2023) (cit. on p. 18).

[48] Apache Maven Project, ed. *Maven - POM Reference.* URL: https://maven.apache.org/pom.html (visited on 10/18/2023) (cit. on p. 18).

[49] Martinig & Associates, ed. *Methods & tools: Practical knowledge source for software development professionals.* Jan. 2010. URL: https://www.methodsandtools.com/PDF/mt201001.pdf (visited on 10/18/2023) (cit. on p. 19).

[50] Mads Jon Nielsen. *GitLab CI Local.* URL: https://github.com/firecow/gitlab-ci-local (visited on 10/18/2023) (cit. on p. 19).

[51] Marco Taboga. *Linear correlation.* Ed. by Kindle Direct Publishing. Lectures on probability theory and mathematical statistics. Online appendix. 2021. URL: https://statlect.com/fundamentals-of-probability/linear-correlation (visited on 10/23/2023) (cit. on p. 29).

[52] Pritha Bhandari. *Correlation Coefficient | Types, Formulas & Examples.* Ed. by Scribbr. Aug. 2, 2021. URL: https://www.scribbr.com/statistics/correlation-coefficient/ (visited on 10/23/2023) (cit. on p. 29).

[53] Microsoft Support, ed. *CORREL function.* URL: https://support.microsoft.com/en-us/office/correl-function-995dcef7-0c0a-4bed-a3fb-239d7b68ca92 (visited on 10/23/2023) (cit. on p. 29).

[54] Rebecca Bevans. *An Introduction to t Tests | Definitions, Formula and Examples.* Ed. by Scribbr. Jan. 31, 2020. URL: https://www.scribbr.com/statistics/t-test/ (cit. on p. 33).