Politecnico di Torino

Computer, Cinema and Mechatronics Engineering commission

Master's Degree Course in **Computer Engineering**

PROGRAMMING ASSIGNMENTS EVALUATION USING TECHNICAL DEBT

Supervisors
Prof. Marco TORCHIANO
Prof. Antonio VETRÒ

Candidate
Alessio MASON

SUMMARY

The background

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.

By this metaphor, for every bug, code smell or generally bad designed piece of code introduced in a program a debt is signed. 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. The only way to pay off this debt is to rearrange and refactor the code, reflecting the newly acquired understanding of the problem.

Technical debt is usually identified through a static program analysis that highlights bugs and code smells.

The goal of the thesis

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.

This 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 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, the thesis 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.

The analysis methodology

The students perspective was researched first: the existing Continuous Integration/Continuous Development pipeline for the course, which currently builds and tests the students' projects, was expanded with a second stage, performing the aforementioned static program analysis. SonarQube was chosen as the software to perform said analysis.

Subsequently, the teacher perspective was considered and a way to conduct a massive analysis (to analyse, for example, all the projects belonging to a specific exam or assignment) was investigated.

Two options were considered. The first one consisted in creating a large project, containing different sources: specifically, every student's project would have to be indicated as a different source.

The other idea was to exploit Maven's multi-module functionality: Maven is in fact the software already used in the course to help building and testing the projects. In Maven a project can be composed of different modules, each being a standalone project in itself, that are then grouped in the parent project.

The latter option was chosen, though both actually being working and viable solution. A Bash script was then developed to automate the analysis.

This script was then used to analyse a real past exam of the aforementioned *Object-oriented programming* course: specifically, the first call for the exams of the 2022/2023 edition of the course, held on June 26th, 2023 was analysed.

Results and conclusions

The analysis of this exam highlighted some bugs and code smells that are the commonest amongst the students.

Generally speaking, the major issues found are related to types management (comparing different types or using nullable types without considering the `null` case), to not checking hazardous cases such as possible divisions by zero, to the comparisons of strings and Boxed types by reference (with `==`) and not by value (using `.equals()`), to code formatting conventions not observed or to methods, fields or pieces of code left unused or commented out.

The correlation between the number of occurrences of every issue and the final mark was investigated: three issues were found to be correlated, and two of these three issues are also correlated with the number of lines of code.
The correlations are for the most part negative, mostly due to the structure of the exam: students are provided with a template for all methods and classes they are supposed to implement. These methods are usually either empty or returning values which are supposed to be changed. 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, hence the negative correlation.

A t-test was then carried out to verify whether the difference between the average mark of the students who incurred a specific issue and the average mark of those who did not is significant: this analysis found 9 issues who had an impact on the final mark or, vice versa, for which the variation in the received mark had an influence in finding more or less of said bugs and code smells.

This thesis, in conclusion, 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 to decide how to deal with them, but in applying the same methods used in said companies to students' programming assignments to identify the major issues and to guide hypotheses 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.