Good morning everybody, and thanks for coming to the presentation of my bachelor thesis. My name is M.P. and today I’m going to present the work I carried out during the last semester, which is mainly focused on the phenomenon of Technical Debt. In particular, how Technical Debt affects code quality and if it is possible to exploit it to recommend successive versions of the code.

I will start with a basic overview of the work context and the main elements of the research, then provide the main ideas and the steps of the approach we adopted and finally comment and contextualise the results we obtained from the analysis.

As you know, software systems are constantly changing, since new versions of the code are released periodically: software updates are supposed to improve code, but sometimes they introduce new faults, such as bugs that were not present in previous versions. So the idea for this work comes from a question: is a newer version that maintains the same functionalities always better? Do design updates guarantee that code contains fewer bugs?

If the answer is no, it may be reasonable to keep the previous version, that worked fine without exhibiting the errors, instead of updating software without functional changes.

This is true even more so in a context of Systems of Systems (or SoS), that is a set of independent systems that integrate into a larger system and provide functions that are the result of the combination of the single parts. In fact, a key characteristic of SoSs is emergent behaviour, namely properties that cannot be ascribed to any single part, but are only manifested by the whole system.

For example, consider a set of ten systems, each with its own interface: here an external system S11 wants to interact with the system, and to do so it can simply interact with the interface of the single systems it needs, for example S1, S3 and S10.

Now consider a general design refactoring applied to the systems, so that no functionality has changed but now there is a unique interface. From now on, any other external application that wants to interact with the system has to adapt in order to use the unified interface. Now if the interaction causes a bug, however, it may be an example of a **design change** (since the functionality remained the same) that generated an issue in an emergent property of the system (that is the unified interface), which the single interfaces did not exhibit.

As we said, in this work we focus on design changes, since we are interested in examining cases where the functionality of code does not change, and an evident indicator of the presence of design changes in code is the phenomenon of **Self-Admitted Technical Debt**. Technical Debt is a metaphor introduced by Ward Cunningham in 1992: sometimes developers prefer implementing quick and temporary code that works immediately but will likely need improvements in the future, rather than code that has, for example, better design but needs more time to be implemented. Analogously to financial debt, if TD is not addressed and solved, it will increase over time, and the accumulation will often have negative impact on code quality.

Sometimes TD is introduced involuntarily, as a result of poorly-written code. However, in this work we focus on the technical debts that are **self-admitted**, that is debts that developers introduce intentionally, as a compromise between for example performance and the need to meet a close deadline. The difference with unintentional Technical Debt is that Self-Admitted Technical Debt, or SATD, is documented in form of comments, and by analysing these comments we can understand what part of a program the SATD refers to - for example, Requirements, Documentation or Tests, and in our case **Design Debt**s concern parts of code that somehow violate the principles of good software design.

So, we’ve seen that comments are indicator of SATD in code. Comments can be found inside or outside a code block, which we call **SATD-Method.** Through **Version Control Systems** like Git we can trace the history of a design-comment and then of its SATD-method.

This schema simplifies a SATD’s lifespan: a SATD was introduced in code at some point A and fixed after some time at a point B. Git provides a series of commands to track the introductory and fixing commits. Between SATD-introduction and fixing there may be thousands of commits, but we are considering only those that address Bugs that were reported after SATD-introduction. Each of these commits includes the set of changes that were made in the corresponding version: from the changes we can track the **change impact** of SATD on code, that is the amount of lines changed due to SATD-introduction or fixing. Also, the lines changed in the Bug-Fixing commit that are found inside the SATD-method can help identify which Bug-Reports are actually related to the SATD.

Our dataset consists of 350 SATD Design comments, selected across 4 open source projects. For each SATD-comments we retrieved the introductory and fixing commits and the corresponding change files. Additionally, we built a custom tool that parses the code to extract the SATD-method from the comment and, given a change file, is able to extract changed lines that are probably connected to the SATD.

With this information we are able to address the three research questions we stated for our work.

First, we analyse the **change impact** of SATD-introduction versus removal, that is the amount of lines of code changed due to SATD-introduction / fixing. A greater change impact indicates greater maintenance effort. Thus, the question is: is it worth to remove SATD if the change impact is high, although the functionality is already working in case of design debt?

Secondly, we investigate the number of bugs that were reported after the introduction and after fixing. Again, is it worth to fix a SATD, if the versions of the code between introduction and fixing were less error-prone? A smaller number of bugs during the “SATD-phase” means that the corresponding code, despite showing design problems, worked better than the one after SATD-fixing.

Finally, we study how methods evolve since a SATD was introduced until it was removed. Does the method grow or shrink? Is it removed and maybe replaced by an improved method? In all these cases, we want to study what impact the changes have on the code.

So, to answer the first question, we compared the change impact, that is the amount of SATD-related lines that were changed, at introduction vs. fixing for all SATD instances in our dataset.

The results we obtained show that between 43% and 67% of the SATD instances required greater effort to pay back the technical debt in terms of lines of code changed. The fact we’re dealing with design debt explains this result, since it makes sense that, in order to solve a design debt, the code undergoes important changes, in a larger quantity than when the debt was introduced, as the debt can be quick and dirty while the fixing must be accurate, and therefore require more effort.

To answer the second question, we developed a heuristic tool that identifies Bug Reports that are related to SATD instances, based on the lines changed in the BR fixing files. Then, for each SATD we counted how many bugs are there between SATD-introduction and fixing versus between fixing and the present. What we observed is expected, in the sense that for between 53% and 80% of the instances, it is true that the presence of a SATD increased the number of bugs, meaning that paying back the technical debt helped improve quality. We noticed, however, that between 14 and 30% of the SATD  have more bugs after fixing, so in these cases it may be reasonable to keep the version where the SATD was present.

FInally, we counted how many lines the SATD-methods contain and compared these quantities at introduction vs. fixing. We observed that 26-31% of the methods shrink when the technical debt is resolved, while between 11% and 24% grows until the debt is fixed. It’s interesting to notice that 42-50% of the methods is completely removed at fixing time, while only 3-9% of methods do not change between SATD-introduction and fixing. These values make sense, considering that we are dealing with Design debt instances: most functionalities featuring a design TD are shrunk in order to pay the debt back, and even more often they are suppressed to make room for more performing methods.

So, to wrap things up, we wanted to investigate if we can exploit SATDs and related changes to identify potentially risky patterns that may result in code defects.

The results showed that it is on average more expensive to pay back a design debt than introducing it., and it is true that the presence of SATD is more likely to cause code defects and that paying back the debt apparently helps solving errors too, at least for debts of type design.

For what we observed, it is necessary to evaluate the characteristics of a SATD-instance from time to time, to decide whether to preserve the debt or to solve it. We saw that fixing design debts is generally more expensive, so it is up to the developer to assess if it is worth fixing a debt, despite the cost in terms of lines modified. Thus, it is a trade-off between higher change and better quality.

We found, however, a few instances where later versions of the code, after the SATD was fixed, contained more bugs. We could not find a common pattern that causes this increase, but we believe these SATDs are a promising path to explore: if we are able to observe that some later releases caused more bugs, we can program a downgrade to a previous more stable version.

Ours was, however, a theoretical study of the phenomenon: as a future work, we will search for concrete examples in SoS to test whether automatic reverting / upgrading software versions based on SATD presence is feasible and brings real benefits to code quality.