Good morning everybody, and welcome to the presentation of my bachelor thesis. My name is M.P. and today I’m presenting the work I carried out during the last semester, which is focused on Studying the Change Impact of Self-Admitted Technical Debt for Reverting / Upgrading Software Versions in Emergent Systems of Systems.

I will provide the idea and the approach of the study - the context, Systems of Systems, and the artifact, Self-Admitted Technical debt -, describe the study design and finally present the analysis and the results.

The idea for this work comes from a question: is a newer version that maintains the same functionalities always better?

This applies in particular 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 referred to any single part, but are only manifested by the whole system.

To make an example, think of two methods that are merged into a single one and together manifest emergent behavior that the single methods did not exhibit.

In this work we focus on design changes, since we are interested in cases where the functionality does not change, and an indicator of the presence of design changes in code is the phenomenon of **Technical Debt**. Technical Debt is a metaphor introduced by Ward Cunningham in 1992, and refers to the fact that sometimes developers prefer an easy, fast solution that works immediately to a better implementation that requires more time. Analogously to financial debt, not addressed and solved, technical debt will increase and have negative impact on code quality.

TD can be introduced voluntarily or involuntarily, where the latter is usually a result of poorly-written code.

Here we focus on **intentional technical debts**, that are introduced as a compromise between for example performance and the need to meet a close deadline. Self-Admitted Technical Debts, or SATD, are intentional technical debts that are also documented in code in form of comments, which point out issues in different parts of code - for example, Requirements, Documentation or Tests, and finally **Design Debt**s concern parts of code that violate the principles of good software design.

The first comment, for example, refers to some feature that should be moved in another class, while in the second “Just to help out during the load” means we are dealing with a workaround, which is another subcategory of design debt.

Comments can be found inside or outside a code block, which we call **SATD-Method.**.

This schema simplifies a SATD’s lifespan: A is the commit where the SATD was introduced in code and B is the commit when it was removed. Git provides the commits and so the dates when the SATD was introduced and removed, and through a bug tracking system we extract the bug reports that were reported after the introductory date. In the next slides I’ll provide more details on how we selected SATD-related Bug Reports - since of course not all BRs are necessarily connected to SATD. Anyway, each Bug Report is associated to a Bug Fixing commit, which contains the SATD-method block.

Our dataset consists of 350 Design SATD comments, selected across 4 open source projects. To extract data we mainly used 3 git commands: git log to retrieve introductory and fixing commits; git show to extract code as it was at any version; and Git diff to obtain the change set between any two versions.

Additionally we built two custom tools in Java: the first is to extract the SATD-method from the introductory and fixing commits using the SATD-comment. The second is to identify SATD-related changes in a bug-fixing change set - and so to identify SATD-related Bug Reports - based on three heuristic rules: so the tool selects the lines changed within the SATD-method, the lines containing calls of the SATD-method from other methods and finally lines changed within other methods called from SATD-method

We used this information to address the three research questions we stated for our work.

First, we analyse the **change impact** of SATD-introduction versus removal, where CI is the amount of lines of code changed due to SATD-introduction / fixing.

Secondly, we investigate the number of bugs reported after the introduction and after fixing, to investigate whether code quality increases with respect to the time the SATD was present.

Finally, we study how a method evolves from SATD-introduction to fixing.

To answer the first question, we compared the change impact at introduction vs. fixing for all SATD instances in our dataset.

The results we obtained show that most SATD instances required greater effort to pay back the technical debt in terms of lines of code changed, while a minority required greater effort at introduction. 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 greater changes, as the debt can be quick and dirty while the fixing must be accurate.

To answer the second question, we compared the amount of Bug Reports found between SATD-introduction and fixing versus between fixing and the same time window after fixing.

We observed that most SATD instances had more bugs during the SATD-phase, while only a small number SATD feature more bugs after fixing. So we’ve seen that paying back the technical debt generally helped improve quality. We noticed, however, that some SATD instances instead have more bugs after fixing, and we’ll investigate more deeply these cases as future work.

FInally, we compared the sizes of SATD-methods at introduction versus fixing. Here It’s interesting to notice that the majority of the methods was completely removed or renamed at fixing time. By manually analysing the code, we observed that most of these were marked as deprecated before being removed, so it’s a common practice to remove a SATD-method to make room for a better one, rather than make greater effort to fix the method itself.

So, to wrap things up, we wanted to investigate if we can exploit SATDs and related changes to program automated upgrade or reverts.

The results showed that it is generally more expensive to pay back a design debt than introducing it, that there are more bugs before SATD-fixing in most cases, and that the majority of methods containing a SATD is removed or renamed at fixing.

We found, however, a few instances with larger change impact at SATD-introduction and a few cases with more bugs after SATD-fixing. We tried classifying these instances, but we were not able to find a general cause for these to happen. So our main future work will focus on finding these causes and understand if they are generalisable.

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

\_\_\_\_\_\_\_\_\_\_\_

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.