**Exploiting self-admitted technical debts for emergent events in SoS**

1. **Introduction**
   1. **SATD**
   2. **SOS architectures**
2. **Self-Admitted Technical Debt**
   1. **Definition**
   2. **Classification with examples**
   3. **Previous Work**
3. **Study Design**
   1. **Research goal**
   2. **Research Questions**
      1. **Does code quality increase by fixing a SATD? -> # BugFix[satd] < # BugFix[satd-fix]**
      2. **SATD method dependencies -> Size(C) < Size(C’)**
      3. **Evolution of SATD method -> M’ == 0, M’ == M or M != M**
   3. **Context**
      1. **Jmeter SATD design comments with fix**
   4. **Data Extraction**
      1. **Tools description, git log etc**
   5. **Data Analysis**
4. **Results**
   1. **RQ1**
   2. **RQ2**
   3. **RQ3**
      1. **Example SATD #87 where method is marked deprecated then removed**
      2. **Other examples**
5. **Threats to validity**
   1. **in some fix diffs, SATD is simply removed without relevant code changes**
   2. **False positives or negatives in changes**
6. **Conclusion and Future Work**
   1. **Research goal revisited**
   2. **Extend to other projects**
   3. **Apply to SoS**
7. **References**

*- Analysis of self admitted technical debt categories*

*- Code mining to trace code related to satd*

*- Recommending code releases in emergent events of SoS*

The project will aim to develop and write the student’s bachelor thesis. The research will cover the analysis of self-admitted technical debt (SATD) instances and how they can be exploited in recommending code releases for emergent events in Systems of Systems (SoS).

The thesis project will include the following steps:

- Study and classification of self-admitted technical debt categories observable in existing software systems. The research will use a tool in order to identify SATD examples in an automated fashion, by parsing commented code, and will span multiple software releases;

- Code mining to trace code related to SATD: for each comment recognised as SATD, code will be analysed in order to identify the method(s) referred to by the comment, also considering successive code versions.

- The research will focus on how emergent events can be handled in a context of Systems of Systems. To to accomplish this, different releases of the code will be tested on the same event. The aim is to find which code version is best to handle emergent events in SoS.

ARCHITECTURAL STYLES OF SELF-ADAPTATION FOR SOS

1. Local Adaptation
2. Regional Monitoring - Local Adaptation
3. Collaborative adaptation

The common architecture that self-adaptive Systems of Systems share involves two or more systems, each one including a managing system / controller and a managed system / target. The controller monitors the behavior of the target, and the target adapts to the directives of the controller, in order to achieve specific domain functionalities.

There are three basic architectures to achieve interaction between systems, featuring feedback loops in the form of runtime data:

1. Local Adaptation

There are local feedback loops that do not coordinate directly, only between pairs of managed systems. Feedback loops share no information with each other, so there is a uncertainty about other systems and the environment: indirect interaction may trigger side-effects / emergent events between individually optimised systems of systems.

For example, in a traffic lights managing system, each installation in an intersection may have an independent control system that check if all three lights are working. If it detects a faulty light, it may trigger an alarm signal while waiting for the traffic light to be repaired, for instance by activating an emergency blinking yellow light.

1. Regional Monitoring - Local Adaptation

In addition to the local adaptation architecture, controllers receive feedbacks from neighboring managed systems to support decision making of adaptations, which also reduce potential side effects of indirect feedback architectures and increases dependency. For instance, a traffic monitoring system including cameras distributed along the road may allow information sharing between multiple cameras, in order to detect traffic jams and providing information to clients.

1. Collaborative adaptation

Both controller and managed systems send feedback loops between each other and local loops may cooperate with one another.

For instance, considering independent groups of GPS devices that interact with a server, with each group consisting of a master and multiple slave devices, there may be two feedback loops: the first local loop deals with the activation / deactivation of the GPS service; the second loop acts in a group context and allows the master device to collect data from the slaves and adapt the group in case one GPS service fails.

**Classification of Self-Admitted Technical Debt**

[Maldonado]

**Code debt**. “Problems found in the source code that can affect negatively the legibility of the code making it more difficult to be maintained”.

* Lower internal quality (unjustified issues): low readability, misuse of programming constructs, unnecessary code complexity…
  + A method that reads input string, writes split string and reads split string again;
  + A method that throws a wrong exception;
  + Multiple nested if-else blocks that may be simplified.
* Workaround (justified low quality code): compromise between code quality and specific software requirements.
  + Necessary “ugly” code for a specific unavailable feature, until some better function is developed;
  + Hacks for backward compatibility.

**Design debt.** “Debt that can be discovered by analysing the source code by identifying the use of practices which violated the principles of good OO-design”.

* Code smells: violation of OO design
  + Feature envy: a method that should be moved to a different class;
  + Code clones: duplicated block of code that may be reduced;
  + Lexical bad smells: poor lexicon that can lead to poor comprehensibility or increase software fault proneness;
  + Long method: method containing too many lines of code.
* Design patterns: need for introducing a design pattern

**Documentation debt:** “Problems found in software documentation that can identified by looking for missing, inadequate or incomplete documentation”.

* Inconsistent comments: misleading information, already addressed technical debts, comments still pointing to an issue that has been already closed and classified as “won’t fix.
* Licensing

**Defect debt:** “Known defects that should be fixed, but due to competing priorities and limited resources have to be deferred to a later time”.

* Defects: known issues to be solved, temporary patches implemented while waiting for an official fix.
* Low external quality: problems that might result in a code defect.
  + Missing thrown exception;
  + Missing input parameter control.

**Test debt.** “Issues which can affect the quality of testing activities”.

* Impossibility to reproduce bug behaviour;
* Failing assert statement to be checked;
* Low quality code in test suites.

**Requirement debt**. “Tradeoffs made with respect to what requirements the development team needs to implement or how to implement them”.

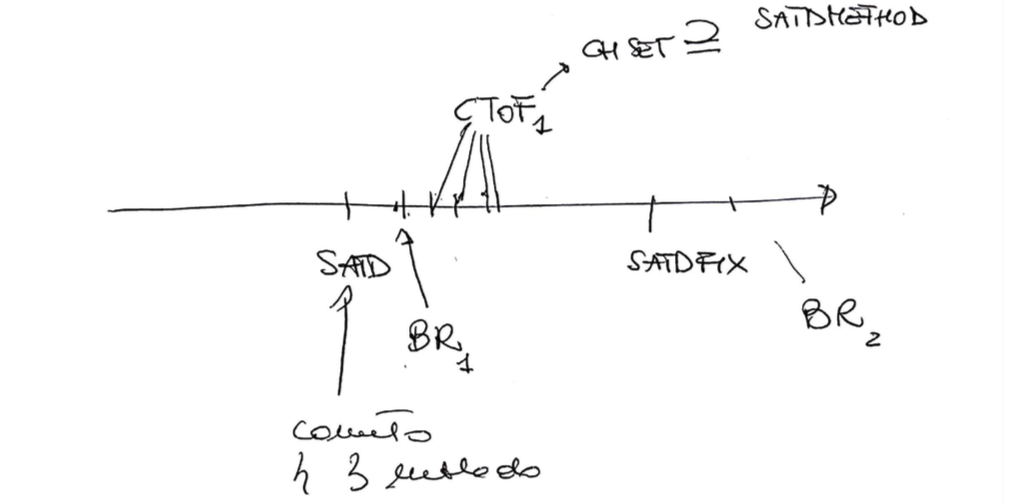
* Functional: existing features that need to be improved or new features to be implemented.
  + Comments reporting implementations that go against the requirement specification;
  + Comments referring to missing features;
  + Comments reporting doubts about the implementation of requirements.
* Non functional: performance issues.

Deriving appropriate threshold values is a challenging open problem that has attracted the attention and effort of several researchers [20], [21], [22]. As a matter of fact, the approaches based on source code analysis suffer from high false positive rates [23] (i.e., they flag a large number of source code elements as problematic, while they are not perceived as such by the developers), because they rely only on the structure of the source code to detect code smells without taking into account the developers’ feedback, the project domain, and the context in which the code smells are detected.

However, relying solely on the developers’ comments to recover technical debt is not adequate, because developers might be unaware of the presence of some code smells in their project, or might not be very familiar with good design and coding practices (i.e., inadvertent debt). As a result, the detection of technical debt through source code comments can be only used as a complementary approach to existing code smell detectors based on source code analysis.

Despite the advantages of recovering technical debt from source code comments, the research in self-admitted technical debt, thus far, heavily relies on the manual inspection of code comments. The current-state-of-the art approach [14] uses 62 comment patterns (i.e., words and phrases) derived after the manual examination of more than 100K comments.

We create a heuristic that removes comments that are placed before the class declaration. Since we know the line number that the class was declared we can easily check for comments that are placed before that line and remove them. In order to decrease the chances of removing a self-admitted technical debt comment while executing this filter we calibrated this heuristic to avoid removing comments that contain one of the predefined task annotations (i.e., “TODO:”, “FIXME:”, or “XXX:”)



1. Associate SATD with Bug-fix before SATD-fix
2. Associate SATD-fix with bug-fix. Find time window between SATD and SATD-Fix (count number of commits between)
3. If time window between SATD and SATD-fix is very large (| SATD ﹣ SATD-fix | >> 0) compute average of commit number (excluding outliers)
4. Associate SATD block to SATD-fix block

RQ: By fixing a SATD, does the code quality increases? ( # BugFix[satd] < # BugFix[satd-fix] )

RQ: Increase set size of SATD comment

RQ: Study evolution of a block of code including a SATD comment

* Analysis of 35 TODO comments from the Apache JMeter project repository [how were these selected exactly?] https://github.com/apache/jmeter/
* Built bash script to automate git command: git log -S’//TODO comment’ [—oneline] —file-path to retrieve the sha of the commits where the TODO was introduced and removed (not necessarily fixed)
* Realised that running the git log using just a portion of the comment (eg, without //TODO), some introduction or fixing shas are different: in some cases, the comment is also modified across different versions -> study of the evolution of the comment
  + Built script to compare shas using the whole comment vs a portion of the comment
  + Study of some interesting cases (4 out of 35) where the git log command returned more than 2 commits (i.e. not only the SATD introductory commit and the removing commit). For example, SATD #87 is introduced, then the method it refers to is marked as deprecated, and finally removed

-built java function that, from file path and comment text, returns the method containing the comment in the file (cases when the comment is inside a block or is external and refers to the block below)

**Java tool to parse code block from SATD comment**

We built a tool in Java that takes as input a SATD comment in the form “//TODO satd comment example” and the code of the class where the comment was found, and returns the code block that the comment refers to.

By analysing the set of SATD comments, we identified three recurring patterns defined as follows:

* Case A: the comment is contained in a code block and concerns one or more lines of code within a method (22 cases out of 35 comments).

[comment ID # 69]

public void clear() {

...

sequenceNumber=0; //TODO is this the right thing to do?

}

* Case B: the comment is outside a block refers to a method that is below the comment itself, recognizable from a method declaration (8 cases out of 35).

[comment ID #82]

//TODO - does not appear to be called directly

public static Vector getControllers(Properties properties) {

…

}

* Case C: the comment is again outside a block and refers to a single statement rather than a block of code, which usually lies immediately below the comment itself, or in some cases above the comment (4 cases out of 35). The statement may be a variable instantiation and is distinguishable from case B since it ends with a semicolon and does not have curly braces at the end of the line or in the line below.

[comment ID #100]

// TODO should the engine be static?

private static final JexlEngine jexl = new JexlEngine();

[comment ID #75]

/\*\*

\* Clear the TestElement of all data.

\*/

public void clear();

// TODO - yet another ambiguous name - does it need changing?

The tool addresses these three cases as follows:

* Case A: it searches for the SATD comment within the Java class. Then a parser analyses the code backwards until a method declaration is found, in the form (public|private|protected) [static] [final] return\_type method\_name(parameters) { ...

At this point, the parser moves forwards and keeps track of the curly braces until the count is greater or equal than 1 and the count of open braces is equal to the count of closed braces. The result is the code block that contains the comment.

* Case B: starting from the SATD comment line, the parser finds the first method declaration below (it may not be in the directly following line, if we deal with a multi-line comment). Then the parser proceeds like case A, by counting open and closed braces, and returns the complete block once the counters have the same value.
* Case C: starting from case B, if the parser finds a semicolon at the end of the end of the method declaration, it stops and returns it, as it is in fact an autonomous statement that does not precede a code block.   
  This technique will not work in the few cases where the comment is below the statement, as comment #75 shows. In these cases, however, developers clarified which statement the comment refers to by inserting separation blank lines between the statement itself and the surrounding ones. Therefore, the parser recognises these empty lines and is able to return the correct statement.

**Identification of bug reports related to SATD comments**

1. From the diff file of a Bug Report commit between SATD and SATD-fix, search for lines of code changed (starting with + or -) that are included in a SATD method block;
2. Within the diff file, search for calls of the SATD method related to lines of code that was changed in the BR commit.
3. Within the SATD-method, search for other method calls (also in different classes) that have changed lines in the BR commit.
4. Search for external variables, instantiated outside the SATD-method and user inside it, that have been changed in the BR commit.

1.

2.

git diff <sha>^ <sha> -U1000 | grep "methodName"

**Example of evolution of a SATD comment**

Starting from a set of technical debt comment to be used for manual validation, selected from the project Apache Jmeter, we executed the command:

Relevant cases:

*// TODO only called by UserParameterXMLParser.getXMLParameters which is a deprecated class*

By running the command git log -S using the whole comment above, the results are the commits where the comment was introduced and then removed, as expected.

$ git log -S'// TODO only called by UserParameterXMLParser.getXMLParameters which is a deprecated class' --oneline

**0c80f9588** deprecate some methods in JMeterUtils Contribution by Benoit Wiart #resolve #148 https://github.com/apache/jmeter/pull/148/

**c932ee6a2** Tidy; add usage comments

However, few lines lines before the removal in the file version when the comment was deleted, the following code displays:

[commit **0c80f9588**, Mar 18 2016]

...

+ \* @deprecated (3.0) was only called by UserParameterXMLParser.getXMLParameters which has been removed in 3.0

\*/

- // TODO only called by UserParameterXMLParser.getXMLParameters which is a deprecated class

+ @Deprecated

public static XMLReader getXMLParser() {

...

The //TODO was removed, but not the method the comment was referred to. In fact, another comment is introduced: it specifies that the method below is now deprecated and was only called by a function that has been removed in a previous version of the application. By running again the git log, this time inserting the comment without the //TODO and also removing the final words, so that it matches the newly added @deprecated comment, we obtain the version where the latter is also removed:

$ git log -S'only called by UserParameterXMLParser.getXMLParameters which' --oneline

**3392c7314** sonar: fix errors **Remove deprecated methods**

c932ee6a2 Tidy; add usage comments

[commit **3392c7314**, Dec 28 2016]

…

- \* @deprecated (3.0) was only called by UserParameterXMLParser.getXMLParameters which has been removed in 3.0

- \*/

- @Deprecated

- public static XMLReader getXMLParser() {

...

The whole method is removed at this point, while it was only marked as deprecated before this version.

Therefore, the history of the SATD comment may be retraced as follows:

* The //TODO was added to point out that a method was only called by a deprecated function (commit **c932ee6a2** on May 16 2012);
* The //TODO portion was then deleted, but part of the comment remained to mark the function getXMLParameters, which was deprecated and has now been removed (commit **0c80f9588** on March 18 2016). Also the method referred to by the initial comment was marked as deprecated;
* Finally, the whole comment, together with the now deprecated method, is suppressed (commit **3392c7314** on December 28 2016).

From the case described above, we realised that not always a SATD comment is introduced and then removed as it is, but there are examples of comments that ‘evolve’ together with the code. Searching through the history of a project considering portions rather than a whole comment is a technique to identify those changes and retrace the evolution of a SATD comment.