**1. Research goal**

The goal of our research was to study the phenomenon of Self-Admitted Technical Debt and examine the impact of SATD on code quality. More specifically, we analysed the evolution of code related to SATD comments, focusing on the code changes between the introduction of a SATD and the corresponding fix, and the changes after the SATD fix.



**2. Research Questions**

Our study aims to address the following research questions:

1. SATD-related Bug Reports

*By fixing a SATD, does the code quality increase with respect to the time when the SATD was present?* This research question aims to identify Bug Reports that are related to existing SATD functionalities and verify whether the code that follows the SATD fixing can be considered less error-prone than the previous version, when the SATD was used.

*BR[satd] < BR[satd-fix]*

2. SATD method dependencies

*Considering C as the set of lines of code changed when a SATD is introduced and C’ as the set of lines changed at SATD-fixing, is Size(C) > Size(C’)?* This question analyses the impact of a SATD on the code in two moments, i.e. the introduction and the fixing of a SATD. If the hypothesis is proven false for some SATD, then more effort was required to fix a SATD functionality.

3. Evolution of SATD methods

*Considering M a the state of a method when a SATD is introduced and M’ as the state of the method when SATD has been fixed, how does M evolves into M’?* In this research question we study the changes that are introduced when M becomes M’: the method may grow or shrink (M’ != M), remain the same (M’ == M) or even may be removed together with the SATD comment (M’ == 0). In any observable case, what impact do the changes have on the code?

**3. Context**

In order to study the impact of SATD comments on code, we need to analyse the source code of existing software projects.

In a recent study, Bavota and Russo [X1] investigated the diffusion and evolution of SATD in active open-source software projects belonging to the Apache and Eclipse ecosystems, as previously done my Maldonado and Shihab while using Natural Language Processing to detect SATD [X2].

Our context consists of the change history of Apache JMeter, whose source code can be found on GitHub [X3]. In particular, we selected the SATD comments categorised as “Design Debt” belonging to the JMeter project, by exploiting the existing classification performed by Maldonado [link to m’s repo].

We decided to focus on Design SATD because we are especially interested in analysing the behaviour of code workarounds, for their peculiarity of being a temporary solution to be replaced when a better implementation is available.

Additionally, we chose a Java project in order to deepen previous research studies that analysed projects written in Java: although many Git functionalities can be applied on projects in any languages, the tool we built for extracting SATD methods should be adapted in order to work with programming languages other than Java.

**4. Data Extraction**

Since JMeter is an active project currently updated, some SATD comments have been introduced but not yet fixed as we write. Our goal is to study how SATD fixing impacts code quality, therefore we extract only those comments that feature a fixing commit.

Starting from a list of comments classified as “Design Debt”, we used a series of Git commands to extract the data we needed from the JMeter repository.

**4.1. Git commands**

**4.1.1. Git log**

git log -S’comment text’ [--name-only] [--oneline] [-- path/to/file]

-S<string> is used to look for occurrences of the specified string that have been inserted or removed in a file. The command is useful for searching an exact block of code and knowing the history of that block since it first came into being. To search for a block that matches a regular expression rather than a specific string, the command -G<regex> may instead be used.

--name-only is an optional parameter that shows the path of the file including the comment that has been changed. The file path refers to the class where the SATD comment was found and can be used to limit the search to the history of the current class.

If the file path is known, it can be inserted as parameter instead of --name-only.

This is also optional in many cases, since a SATD comment is often very specific to a single block of code. However, in our work some instances of SATD - usually more generic comments such as //TODO replace with proper Exception or //TODO make static? - occurred more than once in a single project, therefore the file path was always included to limit the research to the class that first contained the comment.

Finally, --oneline is an optional parameter that may be used to print relevant results on a single line, i.e. the short SHA and the message of the commit. By omitting the parameter, the result would include additional information, e.g. the author of the commit, the date and the git subversion id (git-svn-id). For our purpose, --oneline was used to handle results more easily.

A sample output of the command is the following:

$ git log -S'// TODO: should this just call super()?' --oneline --name-only

2633ade66 sonar: fix errors

src/jorphan/org/apache/jorphan/gui/RateRenderer.java

bb63ad9f5 Initial version of JTable rendering utility classes

src/jorphan/org/apache/jorphan/gui/RateRenderer.java

The results of the git log command are the short SHAs of the commit that introduced the SATD comment and the commit that removed the comment, followed by the file containing the comment.   
By viewing the version of the file at the point of the introductory commit, we can notice:

...

+ if (! (value instanceof Double)) {

+ setText("#N/A"); // TODO: should this just call super()?

+ return; }

…

While the version where the TODO was removed will look like this:

...

if (!(value instanceof Double)) {

- setText("#N/A"); // TODO: should this just call super()?

+ setText("#N/A");

return; }

…

**4.1.2. Git diff**

git diff -U<n> SHA1 SHA2 [-- file-path] > output-file-path

We used this command to show the differences between the commits SHA2 and SHA1, which are then saved to an output text file to be handled in further analysis.

Again, the file-path parameter must be included if we wish to visualise only the changes in a single file. In case a commit includes changes to several classes and we want to save all differences between the two commits regardless of the file they are contained in, we should simply omit the file-path.

The diff command may also be exploited to view the differences between a commit’s ancestor and the commit itself, the syntax to be used is git diff SHA^ SHA, where SHA^ is the target commit’s ancestor.

To study changes among different versions of the code, it was necessary to visualize the changed files in their entirety - not just the changed lines with few code before and after, as Git visualises by default.

For this purpose, the parameter -U<n> comes in handy. The flag -U specifies how many lines of neighbouring text are present around the point when a difference between two versions occurs. The -U default value in Git is 3, but it can be changed to any positive integer n and the larger n is, the more context lines will be included in the output diff file.

In most cases - including our study - a large number such as 1000 is sufficient to show the whole changed file.

**4.1.3. Git show**

git show SHA:file-path > output-file

The show command is simply used to show the version of a file at the time of the commit SHA. The difference with the command git diff -U1000 SHA^ SHA -- filepath is the lack of changed lines: it represents the file as the programmer viewed it when it was written, without the changes from a previous version.

This is useful to trace the original methods when the SATD comment was introduced.

More Git commands and examples may be found here: <https://git-scm.com/docs>

* Wikipedia pages for examples of git commands

**4.2. Retrieving SATD introduction and fixing commits**

To study the evolution of the code between the introduction and the fixing of a SATD comment, we need the identifiers of the commits where these changes happened. The Git command git log is useful since it allows to retrieve when a chunk of code - the SATD comment, in this case - was introduced and when it was removed.

In some cases, git log shows only these two commits: the insertion of a SATD and, at a later time, its deletion, easily verified by checking that their classes are the same and there is exactly one introduction and one removal:

$ git log -S'// TODO Should this method be synchronized ?' --oneline

63c750cac Bug 57114 - Performance : Functions that only have values as instance...

1a3195d8d Add TODO

[commit 1a3195d8d]

+ // TODO Should this method be synchronized ? all other function execute are

...

[commit 63c750cac]

- // TODO Should this method be synchronized ? all other function execute are

...

However, not all the comments found by Maldonado [link to m’s repo] were suitable for being studied directly.

The comment data-set includes 7856 comments, of which 286 categorised as Design SATD. 263 of these were found to be unique comments (the duplicates were removed from our data-set, since git log is able to retrieve possible multiple occurrence of the same comment).

By analysing them, we found that when git log is run different cases may happen:

1. Only one commit is returned: this means a SATD comment was introduced and has not been fixed at present (126 cases out of 263);
2. Two commits are found (48 / 263), but in some occurrences they both include additions and no deletions. In this case, the comment was found in different classes, but it would not change if they belonged to the same one:

[8d7a86ce]

src/components/org/apache/jmeter/visualizers/RenderAsXPath.java

+ // Should we return fragment as text, rather than text of fragment?

[2d9559c4]

src/components/org/apache/jmeter/extractor/gui/XPathExtractorGui.java

+ private JCheckBox getFragment; // Should we return fragment as text, rather than text of fragment?

1. Two or more commits are found, referring to the same class or different classes (89 / 263). For example, the comment was introduced in two files that were eventually fixed in subsequent commits:

[bcb6c238]

src/core/org/apache/jmeter/gui/util/JLabeledChoice.java

- private ArrayList mChangeListeners = new ArrayList(3);// Maybe move to vector if MT problems occur

[31ecdbb0]

src/core/org/apache/jmeter/gui/util/JLabeledTextField.java

- private ArrayList mChangeListeners = new ArrayList(3); // Maybe move to vector if MT problems occur

[dd9932a0]

src/core/org/apache/jmeter/gui/util/JLabeledChoice.java

+ private ArrayList mChangeListeners = new ArrayList(3); // Maybe move to vector if MT problems occur

src/core/org/apache/jmeter/gui/util/JLabeledTextField.java\_old

+ private ArrayList mChangeListeners = new ArrayList(3);// Maybe move to vector if MT problems occur

In this case, it was frequent to retrieve SATD comment introduced and fixed in some classes and only introduced in others. In order to analyse these comments, the commits were always associated to the path of the class they belong to.

**4.3. Selection of SATD comments having (only) introduction and fixing commits**

In order to ease the SATD skim process, we built a bash script that automatically runs the log command, finds the commits and their file paths and associates corresponding introducing and fixing commits for the same class by checking their change set [gitLog.sh]. Eventually, we obtained 104 unique comment-file pairs that we proceeded to analyse [MaldonadoSatdWithFix 34 + 70]: those some SATD comments included multiple times were found in different classes, as described in case C, and therefore represent separate paths of SATD introduction-fixing.

This number is also inline with the result found by Potdar and Shihab [X4]: in their analysis of SATD in five open-source systems, they observed that between 26.3% and 65.3% of self-admitted technical debt is resolved by programmers. After our selection, we found 104 out of 263 design comments removed, which is approximately 40%.

* Analysis of 35 TODO comments from the Apache JMeter project repository 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)

**5. Data Analysis**

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

To track the history of a code block affected by a SATD, we must search for the SATD comment within the file at the version when the comment was introduced and save the corresponding code chunk, so that we can search for it in following versions and retrace its evolution.

We built a tool in Java that takes as input a SATD comment in the form “//TODO satd comment example” and the file path of the class where the comment was found first, 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 (62 out of 104 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 and refers to a method that is below the comment itself, recognisable from a method declaration (18 cases out of 104).

[comment ID #82]

//TODO - does not appear to be called directly

public static Vector getControllers(Properties properties) {

…

}

* Case C: again, the comment is 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 (24 cases out of 104). 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 lines 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**

Next step is to find which Bugs are related to the SATD method, among all the Bug Reports found between the SATD introduction and fix and after the fix.

In RQ1 we defined C as the set of SATD-related lines of code changed when a SATD is introduced. More specifically, by exploring the code of a commit we can observe the set chLOC of lines of code changed (recognisable by the symbols + and - at the beginning). The challenge is to find which lines are connected to the SATD, in order to quantify the impact of change with respect to the introduction and the fixing.

Once we have found C, we can exploit it not only for the initial and fixing commits, but also to find those Bug Report commits whose changes affected the SATD-related methods, as for RQ2.

By analysing the source code, we defined a heuristic that addresses four main rules to identify SATD-related LOC:

1. Within a Bug Report diff file, search for the changed lines that are included in the method identified by a SATD comment, which we located through the tool defined in 3.5.1.

[from SATD # 209, commit e5c10847]

public void valueChanged(TreeSelectionEvent e) {

- log.debug("Start : valueChanged1");

+ lastSelectionEvent = e;

DefaultMutableTreeNode node = (DefaultMutableTreeNode) jTree.getLastSelectedPathComponent();

- if (log.isDebugEnabled()) {

- log.debug("valueChanged : selected node - " + node);

- }

…

}

1. Within the BR diff file, search for the calls of the SATD method that were changed in the BR commit.

[from SATD # 209, commit e5c10847]

+ this.valueChanged(lastSelectionEvent);

1. Within the BR diff file, search for other method calls found in the SATD-method that feature changed lines.

[from SATD # 78, commit cf1c0dc65]

public void run() {

...

notifyListeners(pack.**getSampleListeners**(), result);

...

}

…

+ private List **getSampleListeners**(SamplePackage samplePack, SamplePackage transactionPack, TransactionSampler transactionSampler) {

+ List sampleListeners = samplePack.getSampleListeners();

...

+ return sampleListeners;

+ }

1. Search for global variables, instantiated outside the SATD-method (but within the class) and used inside it, that have been changed in the BR commit.

It is worth mentioning that Java IDEs such as Eclipse offer functionalities to automatically find method calls, for example through the Call Hierarchy, and normally the same result is achievable using Reflection [https://docs.oracle.com/javase/tutorial/reflect/]. However, for our purposes, mining the source code was preferable, rather than running the application and performing tests on the several versions of the code we considered. Therefore we proceeded with direct analysis and parsing of the code, following the techniques described in this section.

1. From the diff file of a Bug Report commit found between SATD and SATD-fix (or after SATD-fix), search for lines of code changed (starting with + or -) that are included in a SATD method block;
2. Within the BR 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 global variables, instantiated outside the SATD-method (but within the class) and used inside it, that have been changed in the BR commit.