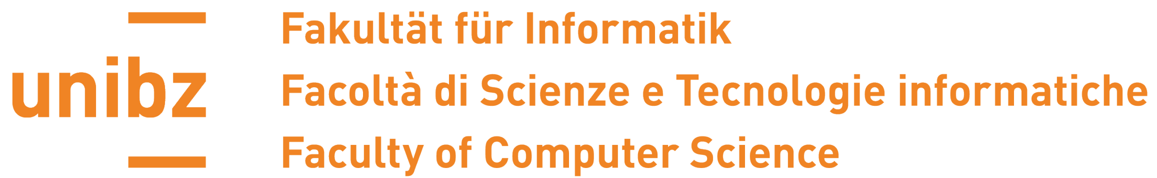


Free University of Bozen-Bolzano
Faculty of Computer Science

Bachelor Thesis

Studying the Change Impact of Self-Admitted Technical Debt for Reverting / Upgrading Software Versions in Emergent Systems of Systems

Marta Pancaldi
12215

Thesis Supervisor: Prof. Barbara Russo

July 2017

Contents

1 Introduction **4**
- 1.1 Systems of Systems . . . 4
- 1.2 Self-Admitted Technical Debt . . . 6

2 Study Design **10**
- 2.1 Research Goal . . . 10
- 2.2 Research Questions . . . 10
- 2.3 Approach . . . 11
- 2.4 Context . . . 12
- 2.5 Data Extraction . . . 13
- 2.6 Data Analysis . . . 16
- 2.7 Comment on the Results . . . 20
- 2.8 Replication Package . . . 20

3 Results **21**
- 3.1 RQ1: change impact of SATD-introduction and fixing . . . 21
- 3.2 RQ2: SATD-related Bug Reports . . . 23
- 3.3 RQ3: Evolution of SATD-methods . . . 26

4 Threats to Validity **33**
- 4.1 Regular Expression . . . 33
- 4.2 Comment removal vs. SATD-fixing . . . 33
- 4.3 Identification of Bug Reports related to SATD . . . 33

5 Conclusion and Future Work **34**
- 5.1 Application of Results . . . 34
- 5.2 Application to other projects . . . 34
- 5.3 Evolution of comments . . . 35

Acknowledgements **36**

Bibliography **36**

Abstract

Software systems are constantly changing, as new versions of the code are released periodically: software updates are supposed to improve code, but sometimes they introduce new faults. Also, software systems may have to interact with each other, aggregating into Systems of Systems and often resulting in Emergent Events, i.e. behaviour that emerges from the systems as whole and that the single parts do not manifest.

This paper studies whether newer releases of code that maintain the same functionality are always better than previous versions, based on documentation that developers themselves insert in the code, in the form of Self-Admitted Technical Debts (SATD). Technical debt is a metaphor introduced by W. Cunningham that reflects the latent cost of additional revisions of code, caused by choosing an easy solution now instead of a better approach that would take longer. The easy implementation may help the project be launched onto the market more quickly, however the accumulation of debts will have negative impact on the quality over time.

Here we trace the history of more than 350 SATD occurrences found in 4 open-source projects and identify the blocks of code affected by SATD, then we count how many bugs exist that are related to SATD. The aim is to study how SATD instances affect the code, in terms of lines changed to introduce/fix debts and reported faults: knowing this, it may be possible to automatically upgrade/revert code to a version that is known to be less error-prone, in order to narrow the amount of faults due to emergent events.

Our main findings show that (i) 43-67% of the SATD instances required greater effort to pay back the debt, and (ii) 53-80% of the SATDs were connected to more bugs in the "SATD-phase" (i.e. between SATD-introduction and fixing) than in the "after-SATD-phase" (between fixing and present time). We also observe that only 14-30% of the SATDs feature more bugs after the fixing, while in 7-15% of the cases the number of bugs does not change. Finally, we observe that the majority of the methods are completely removed at SATD-fixing (43-47%), due to code refactoring. The rest of the SATD-methods grow (11-24%), shrink (26-31%) or remain the same (4-13%) when the SATD is fixed.

Sommario

I sistemi software sono in costante evoluzione, poichè nuove versioni del codice vengono rilasciate periodicamente: gli aggiornamenti software dovrebbero migliorare il codice, tuttavia a volte introducono errori che prima non erano presenti. Inoltre, i sistemi software a volte devono interagire tra loro, aggregandosi in Sistemi di Sistemi, spesso avendo come conseguenza gli Eventi Emergenti, cioè comportamenti che emergono dall'intero sistema e che le singole parti non manifestano.

In questo lavoro testiamo se una nuova release del programma che mantiene la stessa funzionalità è sempre migliore di una versione precedente, in base alla documentazione che gli stessi sviluppatori inseriscono nel codice, sotto forma di "Self-Admitted Technical Debt" (SATD, cioè debiti tecnici inseriti consapevolmente). "Debito tecnico" è una metafora introdotta da W. Cunningham che riflette il costo delle revisioni aggiuntive del codice, causate dalla scelta di una soluzione di sviluppo facile anziché un approccio migliore che richieda più tempo. La soluzione facile può far sì che il software venga rilasciato sul mercato in tempi più brevi, tuttavia l'accumulo di debiti avrà nel tempo un impatto negativo sulla qualità del codice.

Qui ricostruiamo la storia di più di 350 SATD presenti in 4 progetti open-source e identifichiamo i blocchi di codice interessati dal SATD, quindi contiamo quanti bug riconducibili a un SATD esistono nel codice. L'obiettivo è di studiare come i SATD influenzano il codice, in termini di linee modificate per introdurre e risolvere i debiti e di bug segnalati: sapendo ciò, sarebbe possibile fare automaticamente l'upgrade/downgrade a una versione che si è vista contenere meno errori, in modo da limitare la quantità di bug dovuta a eventi emergenti.

I nostri risultati mostrano che (i) il 43-67% dei SATD ha richiesto maggiore sforzo per ripagare il debito e (ii) il 53-80% dei SATD è connesso a più bug nella "fase SATD" (cioè tra l'introduzione e la risoluzione del debito) che nella fase "dopo il SATD" (tra la risoluzione e il presente). Inoltre osserviamo che solo il 14-30% dei SATD presenta più bug dopo la risoluzione, mentre nel 7-15% dei casi il numero di bug rimane invariato. Infine, osserviamo (iii) che la maggior parte dei metodi sono completamente rimossi quando il SATD viene risolto (43-47%), nell'ambito di un refactoring del codice. Il resto dei metodi SATD crescono in dimensioni (11-24%), si riducono (26-31%) oppure rimangono uguali (4-13%) quando il SATD viene risolto.

Kurzfassung

Software-Systeme ändern sich aufgrund von regelmäßigen Quellcode-Aktualisierungen ständig. Deren Ziel ist es, den Code zu verbessern, aber manchmal führen sie neue Fehler ein. Außerdem müssen Software-Systeme manchmal miteinander interagieren, wenn diese zu einem System von Systemen aggregiert werden. Diese Interaktion führt daher oft zu „Emergent-Ereignissen“, d.h. gewisse Verhaltensweisen treten aus den Systemen als Ganzes hervor, und manifestieren sich nicht in ihren einzelnen Teilen.

Diese Bachelor-Arbeit untersucht, ob neue Versionen von Code, die die gleiche Funktionalität beibehalten, immer besser sind als frühere Versionen, auf der Grundlage von Quellcode-Dokumentationen vorherzusagen. Diese Dokumentationen fügen die Entwickler in Form von „Self-Admitted Technical Debts“ (SATD, sinngemäß: bewusste technische Schulden) selbst ein. „Technische Schulden“ ist eine Metapher, die W. Cunningham eingeführt hat, um die latenten Kosten für zusätzliche Code-Revisionen zu beschreiben, die durch die Bevorzugung einer schnellen Lösung gegenüber einer zeitlich aufwändigeren, aber besseren Implementierung verursacht wurden. Solche Design-Entscheidungen/Bevorzugungen können dazu beitragen, dass das Projekt schneller auf den Markt gebracht wird, aber die damit einhergehende Anhäufung von „technischen Schulden“ wird sich im Lauf der Zeit negativ auf die Code-Qualität auswirken.

Die Studie rekonstruiert den Verlauf von mehr als 350 SATD-Instanzen, die in vier Open-Source-Projekten gefunden wurden. Die von SATD betroffenen Codeblöcke werden bestimmt und mit ihnen die Anzahl der Fehler, die auf SATD zurückzuführen sind. Ziel ist es zu untersuchen, wie diese SATD-Instanzen den Code beeinflussen, und zwar in Form von Code-Zeilen, die geändert wurden, um „technische Schulden“ einzuführen und zu beheben und Fehler zu melden: Wenn man das kennt, wäre es möglich, den Code automatisch auf eine Version zu aktualisieren, die bekanntermaßen weniger fehleranfällig ist, um die Anzahl der Fehler aufgrund von zu „Emergent“-Ereignissen zu verringern.

Unsere wichtigsten Ergebnisse zeigen, dass (i) 43-67% der SATD-Instanzen eine größere Anstrengung erforderten, um die Schulden „zurückzuzahlen“, und (ii) 53-80% der SATDs mit mehr Bugs in der “SATD-Phase” (d.h. zwischen SATD-Einführung und dessen Behebung) verbunden wurden als in der “SATD-Nachphase”. Wir beobachteten auch, dass nur 14-30% der SATDs mehr Bugs nach der Behebung aufwiesen, während sich in 7-15% der Fälle die Anzahl der Bugs nicht geändert hat. Schließlich beobachten wir, dass (iii) die Mehrheit der Methoden bei der SATD-Fixierung (43-47%) vollständig entfernt ist, aufgrund von Code-Refactoring. Der Rest der SATD-Methoden wachsen (11-24%), schrumpfen (26-31%) oder bleiben die gleiche (4-13%), wenn die SATD fixiert ist.

Chapter 1

Introduction

One of the challenges in Systems of Systems (SoS) is dealing with emergent behaviour, i.e. events that may arise as a result of the interaction between the two (or more) systems and that the systems did not exhibit as standalone elements. More generally, active software systems are constantly changing, as new versions of the code are released periodically: although a software update is expectantly supposed to enhance the code, for example by fixing bugs and improving existing functionalities, it is always possible that changes in code introduce further problems: intuitively, the larger a software project is, the more bugs and errors it will contain.

The idea for this work comes from a series of questions related to emergent events in SoS and to software releases that unknowingly introduce new bugs: is a newer version that maintains the same functionalities always better? When can we recommend to update a system to a newer version?

To address these questions, we tested how the so-called *Self-Admitted Technical Debts* (SATD) affect the code: a SATD is a concept in programming that describes the phenomenon where developers prefer writing quick, temporary code that works immediately but will likely need improvements in the future, rather than code that has, for example, better design but is slower to implement. Analogously to financial debt, if not addressed and solved, technical debt increases over time, and debt accumulation often has negative repercussions on code quality. It is important to notice that there are several types of SATD, some that affect the functionality of code (Requirement, Defect, Test), some that do not (Design, Documentation), as we will see in the next sections. We focus on Design Debt because they do not change the functionality of a system but they may affect the way systems interact with each other (e.g. via interfaces), which is our primary goal as we investigate SoS.

Programmers document the debts they introduce in comments next to the code block they refer to, and the comments remain in code until the debt has been solved. Our goal is to trace the change history of these SATD comments across software versions, understand if they are connected to any bugs and observe how code quality is affected by the introduction and removal of the SATD occurrences. We focus on design SATD so that functionalities of the systems that compound SoS are not changed. Additionally, we consider changes at method level, in order to scope our analysis on those method involved in external functionalities that link a system to anther one.

At the end, we should be able to determine whether the presence of Design SATD represents a danger to the correct behaviour of the code and if we are able to recommend or advise against a code version – to handle events in SoS or simply suggest a former release to customers – based on the quality of code affected by SATD.

1.1 Systems of Systems

A System of Systems is described as a collection of task-oriented independent systems that integrate into a larger system, providing functions that are the result of the combination of the single parts.

There is no commonly agreed definition of SoS, but several studies address the issue. For example, Boardman and Sauser [1] defined SoS in terms of autonomy, connectivity, diversity and emergence. Maier [2] stated the two main properties that collaborative SoS possess: *Operational Independence* (if separated, the parts must be able to operate autonomously) and *Managerial Independence* (the components are acquired and integrated separately, but maintain a continuing operational existence independent of the SoS) of the components. Typical examples of SoS are integrated surveillance systems, networked smart homes, transportation or healthcare systems. Weyns and Andersson [3] described the three main architectural styles for self-adaptive SoS (Figure 1.1).

Adaptation is a key characteristic in SoS: a system is adaptive when it is able to adjust its behaviour based on information acquired from the environment [4]. In case of a SoS, adaptation refers to how each part changes according to the other systems in the environment. An adaptive SoS is therefore a set of cooperating entities forming a unity. Together the entities are able to respond to contextual changes or changes in the interacting parts [5].

The common architecture that self-adaptive SoS share involves two or more systems, each one including a managing system (the *controller* [6]) and a managed system (the *target* [6]). The controller monitors the behaviour 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, managed through local feedback loops in the form of runtime data. The three styles provide increased levels of collaboration and local knowledge sharing.

- **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 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 checks 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.

  In a context of SATD, consider a design change applied to the first system, composed of a target system S1 and its controller C1 (Figure 1.1): with feedback loops at target system level, only S2 will have to adapt to the changes in S1 – and therefore probably all the other systems that interact through S2.

- **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.

  Compared to Local Adaptation, here a Design change in S1 will have a greater impact on the way the neighbouring systems interact with S1: not only S2 but also the controller C2 has to adapt to the changes, as controllers also receive information feedbacks from the neighbours.

- **Collaborative adaptation**
  Both controller and managed systems send feedback loops between each other. Additionally, 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.

  In Collaborative Adaptation, targets interact with neighbouring targets and controllers with controllers: Design changes, e.g. in S2, will only affect the way S1 and S3 interact with S2. Differently from Local Adaptation, however, the same happens if design changes are applied to the managing systems.

1.1.1 Emergence in SoS

Another characteristic that Systems of Systems share is emergence. Karcanias and Hessami [7] defined emergence as a property that cannot be ascribed to any part of the system, but is manifested by the system as a whole.

Emergent properties are divided into weak or strong properties [8], where weak emergence can be viewed as the predictable integration of the inherent properties of the parts, while strong emergence is a property that the entire system of systems exhibits and is irreducible to the single parts. The concept that the whole is more than the sum of its components has been largely explored in both science and philosophy [9]. One reason for explaining why SoS manifest strong emergence is the combinatorial number of interactions between the systems, which result in new forms of behaviour of the whole system [7].

In order to contextualise emergence with the phenomenon of Design SATD, consider a set S of ten systems, each with its own interface that allows interaction with other applications. A general design refactoring is applied to the systems, so that no functionality has changed but now there is a unique interface for all ten systems. From now on, any other external system that wants to interact with S has to adapt in order to use the unified interface, even if it used to interact with one of the systems in S from before the design refactoring. If the interaction between the external system and the new interface causes a bug, however, it may be an example of Design SATD (since the design of the systems changed, but not the functionality) that generated an issue in an emergent property of the system (the unified interface), which the single interfaces did not exhibit.

Figure 1.2 shows an example where new properties (e.g., system behaviour) emerge due to changes in the system configuration (i.e. a new system is added to the SoS), showing that connectivity is required in order to overcome the diversity of systems and thus enable system emergence.

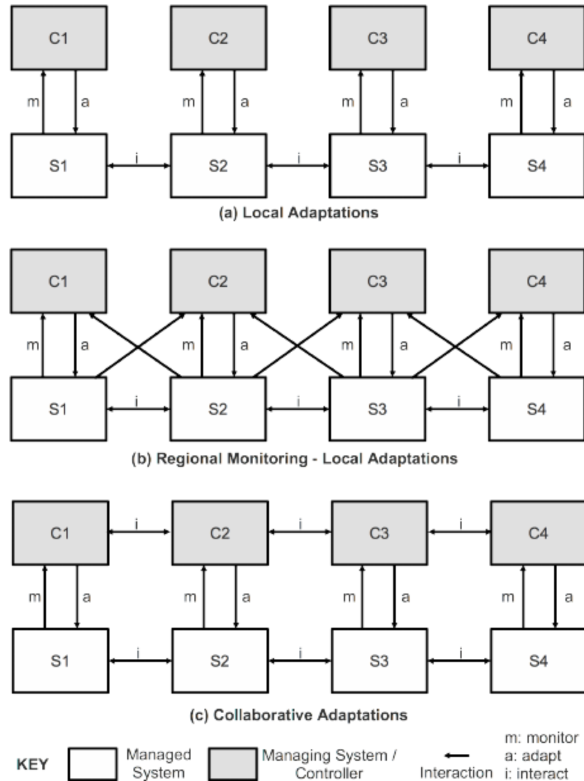


Figure 1.1: Architectures of Systems of Systems, by Weyns and Andersson [3].

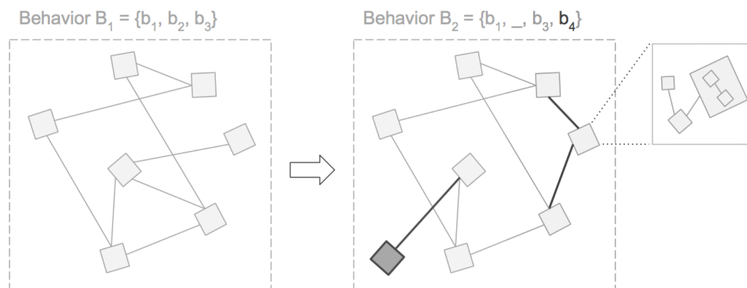


Figure 1.2: Example of emergence of new properties, due to changes in the system configuration (i.e. a new system is added to the SoS) – by Wachholder and Stary [10].

1.2 Self-Admitted Technical Debt

Technical Debt is a concept of programming that reflects the additional development work occurring when programmers prefer code that is easy to implement in the short term, instead of applying the best overall solution in terms of performance, design and functionality.

Ward Cunningham introduced the term in 1992, as a metaphor that equates software development and financial debt, where long-term code quality is traded for short-term gain, creating future pressure as remedy to the expedient: "*Shipping first-time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. Objects make the cost of this transaction tolerable. The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt.*" [11]

We can distinguish two main types of technical debt [12]: the first kind is the one that is incurred *unintentionally*, as the result of poorly-written code. Examples of such debt are a coding procedure that turns out to be error-prone or a faulty program written by inexperienced programmers.

The second kind is the technical debt that is incurred *intentionally*, as a voluntary, strategic choice made by developers, for example in order to meet a deadline, lower production cost and in general deliver a software system to the market in the shortest time possible. In other words, developers sometimes accept a compromise in one dimension (e.g. sacrifice performance) in order to meet an urgent demand in another dimension (e.g. market pressure from competitors), as explained by Brown *et al.* [13]. The difference between planned and

actual delivery is the project's technical debt, which typically accumulates and remains even after release [14], and while the use of a temporary solution may be easy in the short term, there is great chance that an unsolved debt has negative impact on code quality in the long term. [15]

Recently, Potdar and Shihab [16] coined the concept of *Self-Admitted Technical Debt*, which refers to the second kind of code debts – the debts that programmers voluntarily insert. The difference with unintentional debt instances is that Self-Admitted Technical Debt (SATD) is extensively documented in the code in form of comments.

Examples of such comments are: `//TODO: is this the right thing to do?` and `//FIXME: needs to be rethought`.

1.2.1 Categories of SATD

As a quite recent research area, in the last few years there have been many attempts to formalise the phenomenon of technical debt. In particular, the goal is to organise the different types of TD, such as the TD Quadrant by Fowler [17], where debts are classified according to the following characteristics: *Reckless* / *Prudent* and *Deliberate* / *Inadvertent*, as Figure 1.3 shows.

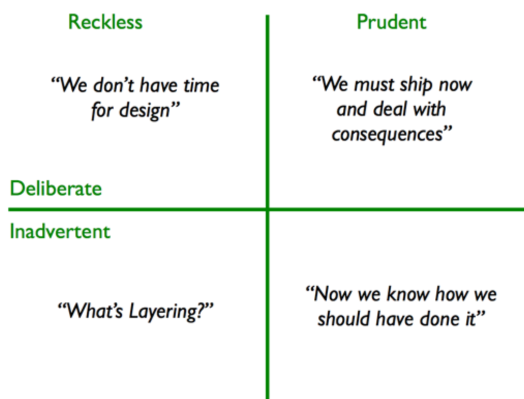


Figure 1.3: Technical Debt Quadrant classification, by M. Fowler. [17]

Alves *et al.* [18] realised an ontology of terms on TD, considering multiple types of debt as they observed in software systems, along with the indicators that characterise the different types. The result is a set of definitions of the kinds of TD, of which we report the most general and important ones.

In fact, Alves *et al.* found 13 categories of SATD; however, Maldonado and Shihab [19] performed a research on real software projects and obtained code examples of the following groups only (because some categories, like *People* or *Infrastructure Debt*, appeared to be rare to observe in source code, and others such as *Build* may be found more easily in languages other than Java), which we are also reporting in this work:

- **Requirement Debt.** "*Requirement debt refers to tradeoffs made with respect to what requirements the development team needs to implement or how to implement them.*" [18]
  Requirement debts can highlight functional problems, i.e. existing features that need to be improved or new features to be implemented (comments reporting missing or incomplete functions, implementations that go against the requirements, expressing doubts about the implementation of requirements), or non-functional issues (e.g. performance).

  `/* TODO: The copy function is not yet completely implemented - so we will have some exceptions here and there."` – [from ArgoUML]
  `//TODO: this is actually against the spec but the requirement is rather silly.` – [from Apache TomEE]

- **Design Debt.** "*Debt that can be discovered by analyzing the source code by identifying the use of practices which violated the principles of good Object-Oriented design.*" [18]
  Design debts include problems like code that should be moved to a different class (*feature envy*), long methods that may be reduced, duplicate blocks of code (*clones*), temporary solutions to known problems (*workarounds*, such as a temporary patch to solve backward compatibility issues) and lexical smells that may lead to poor comprehensibility or increase software fault proneness.

  `// TODO: This is in the wrong place.  It's not profile specific.` – [from ArgoUML]
  `//just to help out during the load (ugly, i know)` – [from Hibernate]

- **Defect Debt.** "*Known defects, usually identified by testing activities or by the user that [...] should be fixed, but due to competing priorities, and limited resources have to be deferred to a later time.*" [18]
  Defect debts refer to all those problems and errors that are known to the developers, but for some reasons have not been corrected yet. This group also includes problems that might lead to defects such as *Bug Hazards* [20] (circumstances that increase the fault proneness), missing thrown exceptions or input parameter controls [21].

  `// We know this fails - Bug# 1700093` – [from SQL12]
  `// FIXME formatters are not thread-safe` – [from Apache Ant]

- **Documentation Debt.** "*Problems found in software project documentation and that can be identified by looking for missing, inadequate, or incomplete documentation.*" [18]
  Examples of documentation debt are inconsistent comments (misleading information, already addressed TD, comments still pointing to an issue that has been already closed or classified as "won't fix"), comments pointing out missing documentation and problems related to licensing.

  `// TODO Can't see anything in SPEC` – [from Apache JMeter]
  `// TODO: - please add some javadoc - ugly classname also` – [from ArgoUML]

- **Test Debt.** "*Issues found in testing activities which can affect the quality of testing activities.*" [18]
  Examples of test debts include a general low quality in testing suites, low coverage of the tests, the impossibility to reproduce a bug when testing activities are performed and failing assert statements that are not checked.

  `// this is the wrong test if the remote OS is OpenVMS, but there doesn't seem to be a way to detect it.` – [from Apache Ant]
  `//TODO - need a lot more tests` – [from Apache JMeter]

Maldonado and Shihab [19] observed that the majority of the SATD is of type Design, with a range between 42% and 84% in the projects analysed, followed by Requirement debt instances, which ranged from 5% to 45%. The remaining types have lower frequency, as they represent together less than 10% of the occurrences.

It is important to notice that not all categories are mutually exclusive [18], but may be recognised into two ore more groups: for example, `//TODO: The copy function is not yet completely implemented - so we will have some exceptions here and there.` [from ArgoUml] refers to a defect debt due to a requirement debt. However, as the authors point out [19], the prevalent feature is usually exploited to assign each SATD instance to a group – in this case, the defect would not exist if there was no previous requirement debt.

1.2.2 Related Work on SATD

Different perspectives were adopted to study the phenomenon of technical debt in the last years.

Since SATD are identified by comments, through which developers provide details about the technical debts they introduced [16], we are also going to exploit the source comment approach to detect TD in code, which was adopted by several prior works [22, 23, 24]. For example, Fluri *et al.* [25] experimented how comments change as the source code grows – whether comments are updated together with the code, or only new comments are added when code changes.

Tan *et al.* [26] implemented a tool to detect inconsistencies between comments and source code as software evolves, and Tan *et al.* [27] also developed an approach that exploits Javadoc Comments to detect inconsistencies within the code.

Other works preferred a different approach, with a focus on understanding and quantifying the phenomenon of SATD, in particular why and how much SATD is removed after its introduction, differently from the prior work which examined the inconsistencies between code and comment co-evolution.

Potdar and Shihab [16] worked on identification of shared comment patterns that indicate the presence of SATD, measured its diffusion and studied why it is introduced and eventually removed.

Kruchten *et al.* [28] provides an in-depth definition of technical debt, considering how industry has embraced TD, by making it visible, integrating it into planning and considering it into future risk. Also, their work pointed out that, while low-quality code usually leads to technical debt, intentional decisions done in order to deliver a product to the market in the fastest way possible do not necessarily mean bad quality.

The work by Zazworka *et al.* [29] compared the amount of technical debt instances recognised by a team of developers in their software project and the amount of TD identified via an automated tool, observing that an identification process performed by humans is necessary to detect some types of technical debt, which automated tools are not usually able to identify.

We also mention a case study, performed by Guo *et al.* [30], where the authors explored the effect of TD in a real software project, by tracking a single delayed task throughout its life-cycle and simulated how explicit TD management might have changed project outcomes.

Finally, Lim *et al.* [31] showed that managing technical debt involves finding a compromise between fast-delivered projects and code quality, as software practitioners recognise that TD is unavoidable and necessary within business realities.

As for the actual recognition of technical debt in code, since comments are written in natural language, it is very difficult to automatically analyse them. Many studies have so far made use of static code analysis, in order to detect the most frequent types of technical debt. In particular, we cite again the work by Potdar and Shihab [16], who manually read through more than 100K comments, obtained by parsing four open source projects' code, and classified 62 recurring patterns that characterise comments containing TD. Examples of such patterns are `FIXME`, `HACK`, `temporary solution` and `this isn't quite right`. Following the patterns, they observed that occurrences of SATD are present in 2.4% – 31% of the files in a project and that only between 26.3% and 63.5% of SATD gets removed from code after introduction, although SATD is supposed to be eventually addressed or removed.

Maldonado *et al.* [32] performed a similar work, focusing on SATD of type design and requirement – the two most common kinds [19]: they developed and tested a heuristic based on Natural Language Processing (NLP) to identify design and requirement SATD, considering the 10 most common words that best indicate the presence of a SATD in code, and building a maximum entropy NLP-based classifier to identify the most common types of SATD. The heuristic is capable of excluding patterns that are most certainly not related to SATD (such as Javadoc comments or commented source code), and was proven to be more accurate in identifying SATD than the simple analysis of comment patterns and the manual inspection of code.

Bavota and Russo [21] also worked on a large-scale study of the diffusion and evolution of SATD across 159 open-source projects, by performing code inspection using the 62 comment patterns by Potdar and Shihab and manually classifying occurrences of comments recognisable as technical debt.

The approaches we presented so far share a significant drawback, that is the necessity of manual effort to summarise patterns and classify TD comments, which is time-consuming. The most recent work completed by Huang *et al.* [33] uses instead a Machine Learning-based text mining technique, which takes the patterns found by Potdar and Shihab as starting model and is then able to automatically identify SATD comments. The results were consequently compared to the ones by Maldonado's NLP-based procedure and the accuracy of the former in identifying comments containing SATD was the best among the approaches we have seen so far. Also, differently from Maldonado's work, Huang's approach is meant to detect all known types of SATD, not only design and requirement.

Finally, the work by Wehaibi *et al.* [34] is perhaps the closest to our own study, as they empirically analysed the impact of SATD on software quality by mining source code of existing projects. In particular, they investigated whether files including SATD contain more defects than files without SATD and if the defects increase after SATD-introduction, whether the changes related to SATD introduce new defects and finally if the SATD-related changes have greater change impact – in terms of modified lines, files and directories – than the non-SATD changes. We use, however, a different approach to select parts of code interested by SATD occurrences, identify SATD-related bugs and analyse the change impact of bug fixings. We further compare Wehaibi's and other's approaches with our own in the following chapter.

Chapter 2

Study Design

2.1 Research Goal

As we explained in the previous chapter, in a context of Systems of Systems two or more software environments have to interact to accomplish common goals. As result of the interaction between two or more systems, emergent events, which the single elements did not exhibit, may arise. This is not a prerogative of SoS contexts: software updates that are released to improve the software functionalities may include bugs or other unexpected responses that the previous version of the code did not contain. In fact we can say, intuitively, that a project with large amount of lines of code will include more bugs than a smaller one.

We started asking a series of questions about how to understand the problem and try to solve it: if we can identify potentially risky recurrent patterns in code changes that are likely to result in code errors, we may be able to recognise which version is less error-prone in a software project. Knowing this, we could automatically restore a previous release in a system component, or recommend customers to keep the version they have installed till the release of a fixing update, if we have observed that the latest version causes a bug when some event emerges.

The phenomenon of Self-Admitted Technical Debt (SATD) seems a promising path to address this issue, since technical debt, if not addressed and solved, increases over time analogously to financial debt, and debt accumulation often has negative repercussions on code quality. The goal of our research is to study and examine the impact of SATD on quality. More specifically, we analyse the evolution of code related to SATD across consecutive software versions, focusing on the code changes between the introduction of a SATD and the corresponding fix, and the changes after the SATD-fixing. Then, we will try to understand if they are connected to any bugs and observe how the introduction and removal of the SATD occurrences affects code quality. At the end, we should be able to determine whether the presence of SATD represents a danger to the correct behaviour of the code and if we are able to recommend or advise against a code version based on the quality of code containing SATD.

2.2 Research Questions

The research questions aim to address three main areas of study related to SATD.

In the first, we analyse the change impact of SATD-introduction versus removal (i.e. 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 (in case of design debt) is already working?

The second investigates the number of bugs that were reported after the introduction of SATD and after the SATD was removed. Analogously to the first question, 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 being flawed by design perspective, worked better than the one after SATD-fixing.

Finally, in the third question 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? Are there cases where the SATD is removed without relevant changes in the code? In any observable case, what impact do the changes have on the code?

To address these questions, we follow the approach that Figure 2.1 summarises.

2.2.1 RQ1: change impact of SATD-introduction and fixing

What is the relation between the code changed after a SATD-introduction and the code changed after the SATD-fixing?

We define the set $C = ChLOC \cap SATD\text{-}method$, where $ChLOC$ includes all the lines that were changed in a code version and *SATD-method* is the functional block of code that a SATD instance refers to. Thus, C is

the intersection of lines changed in a commit and the methods related to the SATD, i.e. the lines changed contained in the SATD-method. C also represent the *change effort* required to introduce or remove a SATD.

Considering C_A as the set of SATD-related lines of code changed when a SATD is introduced and C_B as the set of SATD-related lines changed at SATD-fixing, we compare $Size(C_A)$ and $Size(C_B)$.

If $Size(C_A) > Size(C_B)$ SATD-introduction required more effort than fixing it. Vice versa, more effort was necessary to modify code in order to solve a SATD.

2.2.2 RQ2: SATD-related Bug Reports

By fixing a SATD, does the code quality increase with respect to the time when the SATD was introduced?

This research question aims to identify Bug Reports (BR) that are related to existing SATD instances and study whether the code that follows the SATD-fixing can be considered less error-prone than the previous version, when the SATD was used.

Once we have found the related bugs, we compare the amount of bugs between SATD-introduction and fixing ($\#BR[Intro - Fix]$) with the amount after SATD-fixing ($\#BR[Fix - Now]$).

If the code during the "SATD-phase" includes more bugs, solving the technical debt also helped making the code less error-prone. Vice versa, if the code after the fixing contains more bugs than the code between the introduction and fixing, it may be reasonable to keep the code as it was, with an unsolved but working SATD.

2.2.3 RQ3: Evolution of SATD-methods

Considering M as the method when a SATD is introduced and M' as the method when the SATD has been fixed, how does M evolves into M'?

In this research question we study the changes that are introduced when M at last evolves into M': the method may grow or shrink ($M' \neq M$), remain the same ($M' = M$) or even may be removed together with the SATD comment ($M' = 0$). We may hypothesise another case, namely where the datamethod is split in two or more sub-blocks ($M' + M''$+...+$M^N \simeq M$): in this case the observed result would probably be $M' = 0$, but the functionality remains.

At the end, we compare the groups found with the categories observed in the previous questions, in order to search for relations between groups (e.g. do shrunk methods exhibit more change effort, or a greater amount of bugs after SATD-fixing?).

2.3 Approach

The source repositories of four open software projects are our main data set: we locate the SATD comments identified by Maldonado and Shihab [19] in the source code, tracing the history of each comment, i.e. the version where the SATD was introduced and the version where it was removed. Not all SATD instances found by Maldonado have been fixed at the present day, so we limit our data set to the instances where a fixing commit exists (section 2.5).

At this point, the task is to identify the change impact of each SATD: to do so, we parse the source code of the introductory version and located the functional block of code that the SATD refers to (i.e. the *SATD-method*) and we develop a tool to trace the SATD-related changes (section 2.6). This phase especially required several reviews, since as we got the first results, we frequently went back to improve the tools and limit the false positives.

Finally, we use the tool to identify SATD changes not only in the introductory version, but also in the fixing (RQ1 and 3) and in all commits that fix a Bug reported after the SATD has been introduced (RQ2).

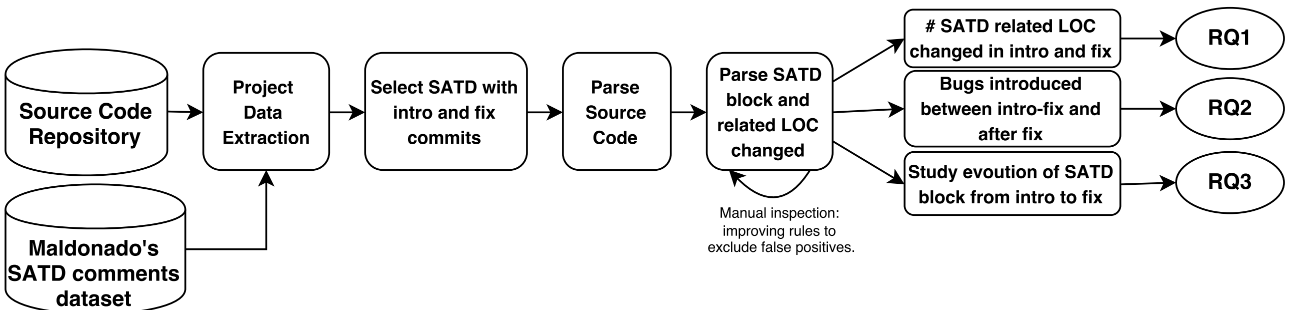


Figure 2.1: Activity diagram of our research approach

A difference we wish to highlight with respect to previous works is that, once we locate a SATD instance in the code, we consider the whole method it refers to as "container". We are aware that, if the comment lies within a smaller block – e.g. `if-else` blocks or `catch` clauses within a larger method –, it is probably related only to that block. However, we are interested in studying the complete functionality affected by the SATD, thus we are using complete methods as SATD-related functionality, rather than the single operations within a method.

This is also different from other previous works, for example the study by Wehaibi *et al.* [34], where the authors abstract up to SATD-*files* (i.e. files containing one or more occurrences of SATD comments) to determine the scope of a SATD instance, and consequently examine the impact of SATD at file level. Similarly, Linares *et al.* [35] consider the change impact at *class* level (speaking only of Java files). Focusing on the method level is in our interest, as our future goal is to relate the functionalities of systems to specific SoS behaviour.

The SATD instances we are addressing were introduced at a point A in the past and were fixed after some time at a point B. We are able to trace these timestamps and retrieve the corresponding versions of the code thanks to the version control system: for example, by viewing the commit corresponding to point A, we can observe all code changes that occurred at that point.

Between the SATD-introduction and fixing – and between the fixing and the current day – there may be thousands of commits. Since we are interested in studying the impact of SATD on software quality, we take the intermediate commits identified as *Bug Report fixings* as milestones, in order to consider changes that are relevant to the evolution of a SATD. However, we are not examining *any* Bug Report starting from the SATD-introduction, since not all bugs are necessarily connected to a SATD. Therefore, our goal is to identify which Bug Reports Fixing files share changed lines with the SATD-methods (i.e. the code block that the SATD comment refers to).

Figure 2.2 represents a schema of a SATD's lifespan: we consider the Bug Reports within the SATD-introduction and fixing and after the fixing. If the Report has been addressed, we find a Bug-fixing commit among the versions, whose change file contains a SATD-method block (since we consider only the bugs reported after SATD-introduction). Details about how we identified relevant Bug Reports may be found in the following sections.

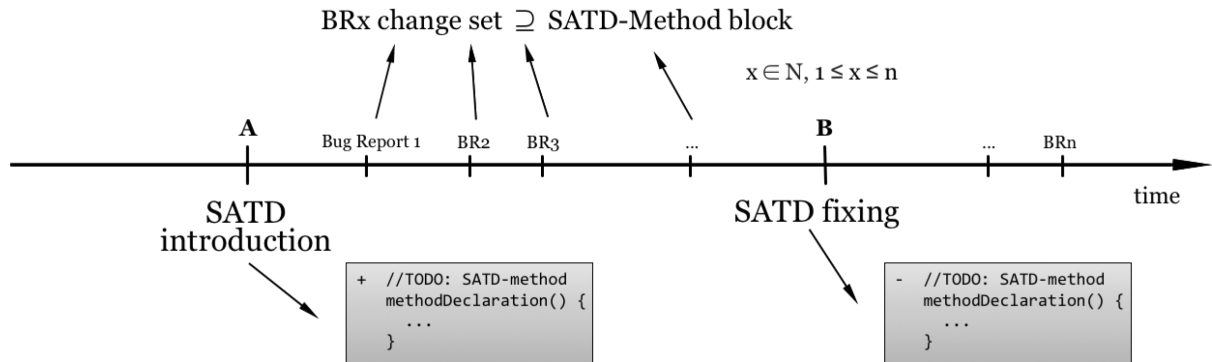


Figure 2.2: A SATD's lifespan schema

2.4 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 [21] investigated the diffusion and evolution of SATD in active open-source software projects belonging to the Apache and Eclipse ecosystems, as previously done by Maldonado and Shihab while using Natural Language Processing to detect SATD comments [19].

Our main context consists of the change history of Apache JMeter, whose source code is stored on GitHub [1]. In particular, we selected the SATD comments categorised as "Design Debt" belonging to the JMeter project, exploiting the existing classification performed by Maldonado and Shihab [36]. At the beginning of the study, we classified a random sample of 35 comments from JMeter into the SATD categories, in order to better learn the existing classification [16]. However, this study focuses on analysing the impact of SATD, rather than testing the validity of the existing categorisation, which has been extensively done in previous works, so we took advantage of the data set provided by Maldonado and Shihab.

[1] https://github.com/apache/jmeter/

We used again the SATD comments data set when we applied our tools to other projects, namely JRuby [2], Apache Ant [3] and Hibernate [4], in order to validate the tools and compare the data.

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. Also, they do not promise to change the functionality of a system, which is our primary goal as we investigate SoS (control for bugs that arise with a new property, avoiding those that come up because of a new version of an individual system in SoS). Finally, design SATD are promising to exploit redundancy in Systems of Systems.

Additionally, we chose Java projects 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 tools we built for extracting SATD-methods should be adapted in order to work with programming languages other than Java.

Table 2.1 provides details about the projects we used in this study: the columns include the current release of the code, the number of classes, contributors and commits in the repository, the total source lines of code (SLOC), the number of comments found and, among those, the number of comments identified as SATD by Maldonado and Shihab, together with the percentage of comments classified as Design debt. We used the tools SLOCCount [37] and CLOC Git [38] to calculate the SLOC (we only considered the SLOC for the Java files), and the plugin JDeodorant [39] for Eclipse to extract the data about code comments.

Additional information such as the number of contributors was extracted from OpenHub [40]. All data are intended to represent the size of the projects we analysed and refer to the present time (June 2017) and will most likely change in the near future, since all projects are currently open. Also, it is important to notice that the number of comments shown for each project does not represent the number of commented lines, but rather the number of individual comments, including single, multi-line and Javadoc instances.

Table 2.1: Project details

Name	**Project Details**					**Comment Details**		
	Release	**# of Classes**	**# of contrib.**	**# of commits**	**SLOC**	**# of comments**	**# of TD comments**	**#,(%) of Design TD**
Apache Ant	1.9.9	1247	89	13,601	138K	21,688	131	96 (73.28)
Hibernate	5.1.7	9,841	340	7,876	653K	11,941	472	355 (75.21)
JMeter	3.2	1,239	43	14,381	125K	21,426	374	316 (84.49)
JRuby	1.7.27	1,766	398	42,840	235K	11,342	622	343 (55.14)
Average		3,526	217	19,674	288K	16,599	402	177 (69.28)
Total		14,103	870	78,698	1,151K	66,397	1610	1110 (-)

2.5 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 [41] to extract the data we needed from the JMeter repository. It is important that, even if we deal with multi-line comments, the list to be processed should only include one-line comments (or a single-line portion of a multi-line comment), since the tools we used parse the code line by line.

2.5.1 Git Commands

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 line of code and knowing the history of that line since it first came into being. To search for a line 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.

[2] https://github.com/jruby/jruby
[3] https://github.com/apache/ant
[4] https://github.com/hibernate/hibernate-orm

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()?
+             setText("\#N/A");
              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; }
```

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 change 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.

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.

2.5.2 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 [36] were suitable for being studied directly. The comment data set of JMeter 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 dataset, 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 occur:

(A) Only one commit is returned: this means a SATD comment was introduced and has not been fixed at present (126 cases out of 263);

(B) 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:

```
[commit 8d7a86ce]
src/components/org/apache/jmeter/visualizers/RenderAsXPath.java
+    // Should we return fragment as text, rather than text of fragment?

[commit 2d9559c4]
src/components/org/apache/jmeter/extractor/gui/XPathExtractorGui.java
+    private JCheckBox getFragment; // Should we return fragment as text, rather than
    text of fragment?
```

(C) 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:

```
[commit bcb6c238]
src/core/org/apache/jmeter/gui/util/JLabeledChoice.java
-    private ArrayList mChangeListeners = new ArrayList(3);// Maybe move to vector if MT
    problems occur

[commit 31ecdbb0]
src/core/org/apache/jmeter/gui/util/JLabeledTextField.java
-    private ArrayList mChangeListeners = new ArrayList(3);  // Maybe move to vector if
    MT problems occur

[commit 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.

2.5.3 Selection of SATD comments having introduction and fixing commits

In order to ease the SATD selection process, we built a bash script [5] 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.

Eventually, we obtained 104 unique comment–file pairs that we proceeded to analyse [6]: 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.

[5] https://github.com/martapanc/SATD-replication-package/blob/master/jmeter/gitLog.sh
[6] https://github.com/martapanc/SATD-replication-package/tree/master/jmeter/104SATDmethods

This number is also inline with the result found by Potdar and Shihab [16]: in their analysis of SATD in five open-source systems, the authors 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%.

2.6 Data Analysis

2.6.1 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. It is possible that some SATD comments exclusively refer to the block that encloses it (for instance, a `while` statement or a `try-catch` clause), but we choose to consider the whole method to study its functionality, as we will examine in the next sections.

We built a tool in Java [7] 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 in JMeter, we identified three recurring patterns defined as follows:

- Case I: 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 II: 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 III: 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?
```

2.6.2 Script for extracting SATD-method information

For each of the 104 comments, the script described in 2.5.3 executes the Java tool in 2.6.1 and generates a text report containing information that are used to extract further data, as described in the following sections.

The data that the report includes are the SATD comment and its original file path, the introduction and fixing commits, the Bug Reports found between introduction-fixing and between fixing-today and finally the start and end indices of the code block that contains the comment (the indices refer to the class where the comment was originally found) and the block itself. Here is an example of such report, referring to comment #69:

[7] https://github.com/martapanc/SATD-replication-package/blob/master/jmeter/MethodFromCommentFinal.jar

```
File path: src/core/org/apache/jmeter/reporters/ResultSaver.java
Comment: //TODO is this the right thing to do?
Initial commit id: 5a0a9ac5e
Final commit id: da1c07e79
   Bugs between [6]:
3e16150b7 Bug 52214 - Save Responses to a file - improve naming algorithm ...
9cca78bc0 Bug 49365 - Allow result set to be written to file in a Path relative ...
d81ad7e22 Bug 43119 - Save Responses to file: optionally omit the file number
59671c56f Bug 44575 - Result Saver can now save only successful results ...
e861ae37d Bug 36755 (patch 20073) - consistent closing of file streams
289264650 Bug 41944 - Subresults not handled recursively by ResultSaver
   Bugs after [2]:
285abc026 Bug 60859 - Save Responses to file: 2 elements with different config ...
11d942f4a Bug 60564 - Migrating LogKit to SLF4J - core/engine,plugin,report, ...

Start block index: 69
End block index: 74
   public void clear()
   {
      //System.out.println("--"+me+this.getName()+" "+Thread.currentThread().getName());
      super.clear();
      sequenceNumber=0; //TODO is this the right thing to do?
   }
```

2.6.3 Identification of bug reports related to SATD comments

Next step is to find which Bugs are related to the SATD-method, among all the BRs found between the SATD-introduction and fixing, and after the fixing till present time. Projects like JMeter use issue-tracking systems such as Bugzilla [8] and JIRA [9] to keep track of bug reports and fixings. For each SATD, we extracted the BRs reported after SATD-introduction using these online tools.

When a Report is addressed in order to fix the corresponding bug (i.e. sharing the same identifier), some changes must be done in the code. Much time can pass between the BR and the fixing, but this does not matter as long as we consider the Reports made between the SATD-introduction and fixing: what we are studying are the lines of code related to the SATD that are changed when the bug is fixed.

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 Fixing commits whose changes affected the SATD-related methods, as for RQ2. From now on we refer to these commit files as BR diff files.

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

1. Within a BR 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 2.6.1

```
[from comment #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);
-       }
...
    }
```

2. Within the BR diff file, search for the calls of the SATD-method that were changed in other methods in the BR commit.

```
[from comment #209, commit e5c10847]
    //A changed line including the SATD-method "valueChanged()"
    public void actionPerformed(ActionEvent event) {
...
+       this.valueChanged(lastSelectionEvent);
...
    }
```

[8] http://jmeter.apache.org/issues.html
[9] https://hibernate.atlassian.net/secure/Dashboard.jspa

3. Within the BR diff file, search for other method calls found in the SATD-method, referring to a method declared somewhere in the diff that feature changed lines.

```
[from comment #78, commit cf1c0dc65]
    //SATD-method that includes a method call
    public void run() {
...
-       notifyListeners(pack.getSampleListeners(), result);
...
    }
...
    //Declaration of the method called above, including changed lines
+   private List getSampleListeners(SamplePackage samplePack, SamplePackage transactionPack,
     TransactionSampler transactionSampler) {
+       List sampleListeners = samplePack.getSampleListeners();
...
+       return sampleListeners;
+   }
```

We built a Java tool to address these rules, as follows:

1. **Lines Changed within SATD-method**

   The tool reads the report text file, previously computed as described in 2.6.2, and saves the SATD-method header. Then it reads the diff file (this can be the SATD introductory or fixing commit, or a BR diff file found after the SATD-introduction) and searches for the method header. Once this has been found, the tool parses the file starting from the header until the end of the method block by counting curly braces – as described in 2.6.1. At the same time, if the parser encounters any line that has been changed (i.e. contains + or – at the beginning), a counter is incremented. When the parser reaches the end of the method block, it returns the counter, indicating how many lines have been changed within the SATD-method in the current commit.

2. **Calls of SATD-method changed in diff file**

   Again, the tools reads the report file and this time only saves the method name.

```
public void valueChanged(TreeSelectionEvent e) {
   => valueChanged
```

   The tool then searches for instances of the method call within the current diff file and increments the count when a changed line including the call is found.

   The result (i.e. the number of SATD-method calls contained in a changed line) is generally accurate, as we realised with a manual analysis. However, some cases of false positive were recorded. A typical case of false positive happens if the SATD-method shares the name with another method (i.e. *Polymorphism* in Object-Oriented Programming), which is invoked by an object of different type of has different arguments.

```
[from comment #107, commit ba3cdcaf4]
//Method declaration line
public void start(String[] args)

//Homonymous method name, but different type of object invoking it
...
+   Thread daemon = new Thread(){
...
+   };
+   daemon.start();
```

   In this example, the SATD-method is called `start()`, but the tool reported a line changed including a method with the same name but different functionality (more specifically, the method that allows a Thread object to run). Consequently, this is not to be treated as a SATD-related change. However, false positives like this were found in only 7 cases among 1245 Bug Reports analysed in JMeter, while more than 200 instances of SATD-method calls were correctly identified, so we can consider the tool fairly accurate for our purposes.

3. **Other method calls from SATD-method in diff file**

   Within a method, several other methods may be invoked. This section of the tool aims to identify those calls within the SATD-method invoking method blocks that were modified inside a diff file of some related commit.

```
public void satdMethodName() {
   var a = b.anotherMethodCall();
}
...
public anotherMethodCall() {
+   added line
-   removed line
...
}
```

   We then built a regular expression that finds all method calls within the SATD-method found previously, then searches for the corresponding declaration, if present, and counts changed lines within the method block.

```
Method calls found:
 * getLocalHostName
 * getJMeterHome
 * getErrorString
 * getLocalHostIP
 * println
 ...
```

   As expected, the tool also found methods that are most likely not connected to SATD, but belong to common Java libraries. In fact, we manually analysed the results of this tool for all SATD instances and found a high false positive rate, between 40% and 60% (respectively for the BRs found between SATD-introduction and fixing and after SATD-fixing).

   An easy solution would be to filter these common methods, such as `println()`, `getStackTrace()`, but the risk is not to consider homonymous methods, such as the `start()` we saw in the previous section. However, we examined the list of methods to find the most common "Java-only" functions that were changed in the code and erroneously treated as SATD-related changes (above all, the overriding of `toString()` was the method that triggered the highest number of false positives).

   Once we applied a filter that removes from the check all those methods, which have most likely nothing to do with SATD, the false positive rate dropped below 20%.

While the first rule of this tool gave evidence to be reliable in all circumstances, i.e. it was able to correctly locate the SATD-method within the diff file and count the lines changed inside it, we observed quite a high false positive rate with the second and especially the third rule (respectively, around 35% and 50%).

We analysed the results one by one in order to identify these cases and exclude from the data set to be further processed those BRs that have actually no correlation with the SATD-method. However, we observed that, compared to the number of BRs analysed, the number of BRs with lines changed according to the second (call of the SATD-method somewhere else in the diff) and third rule (calls of other methods within the SATD-methods) is considerably lower than the number of BRs identified by the first rule. They also represent a small percentage of the total BRs considered within the JMeter project (1245). Finally, we found a very small number of BRs identified *only* by rule 2 or 3, meaning that rule 1 can identify the majority of related BRs on its own.

For example, in case of the BRs between SATD-introduction and fixing, Rule 1 was able to identify 130 of the 147 (88%) total BRs found, while 16 were found through rule 2 and 1 more only through rule 3. Instead, 33 out of 48 (69%) BRs after SATD-fixing were found by rule 1, while second and third rule alone found 6 and 9 BRs respectively.

These data show that the first rule is the most reliable indicator and might be used on its own in some cases, being able to identify between 70% and 90% of the related BRs. However, we made some improvements to rules 2 and 3, mostly by adding matching cases to the regular expressions, in order to ensure higher reliability. After the enhancements, the false positive rates for rules 2 and 3 dropped to 15% and 20%, which may still be improved by increasing the precision of the regular expression, as long as more matching cases can be found. The table that summarises the results in 3.2 already includes the data computed through the improved tools.

2.7 Comment on the Results

The tool analyses separately the BRs found between the SATD-introduction and fixing and between the fixing and the present time, then saves the counter values to a spreadsheet. It also saves the results to a text report including more information for better readability and data analysis.

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 [42]. However, for our purposes, mining the source code was preferable – although we have seen instances (rules 2 and 3) in which the Call Hierarchy would give more accurate result –, 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.

2.8 Replication Package

The tools we wrote and the data we obtained from them may be found in our public repository [43]. The repository includes all the tools (mostly in form of Shell scripts) we developed and used to analyse the code and obtain data for our study, the folders with the results of the four projects and spreadsheet files containing all the data extracted.

All the examples we showed in the three research questions report existing chunks of code taken from the projects, and the numeric notations we used refer to the numeration we assigned to each SATD found – e.g. `[JRuby, comment # 22]` refers to the SATD occurrence with ID # 22 in JRuby. Note that, the mining tools must be applied directly on the desired project repository, which can be cloned from the corresponding GitHub repository.

Chapter 3

Results

This section discusses the results achieved through the code mining tools we built and described in the previous chapter, according to the three formulated research questions.

3.1 RQ1: change impact of SATD-introduction and fixing

We defined C as the intersection between the set of lines changed in a commit (ChLOC) and the SATD-method, along with the lines of code that we found related to the SATD: $C = ChLOC \cap SATD\text{-}method$.

More specifically, we can distinguish between C_A and C_B, where C_A refers to the changes made in the introductory commit of a SATD and C_B as the set of related changes made in the fixing commit.

3.1.1 Results from JMeter

In 2.5.3 we obtained a set of 104 comments that meet the requirements – design comments that have both an introduction and a fixing commit. We found that 24 of these comments refer to single statements (e.g. variable declarations) rather than a method block: since we are interested in studying how the changes in a method affect the rest of the code, we only considered the remaining 80, as comments concerning a method's code block.

By applying the Java tool (2.6.3) to the introductory and fixing commits' diff files of these 80 comments found in JMeter, we were able to compute respectively C_A and C_B for the SATD comments in our data set. Figure 3.1 plots the distribution of C_A and C_B, as we found in our analysis.

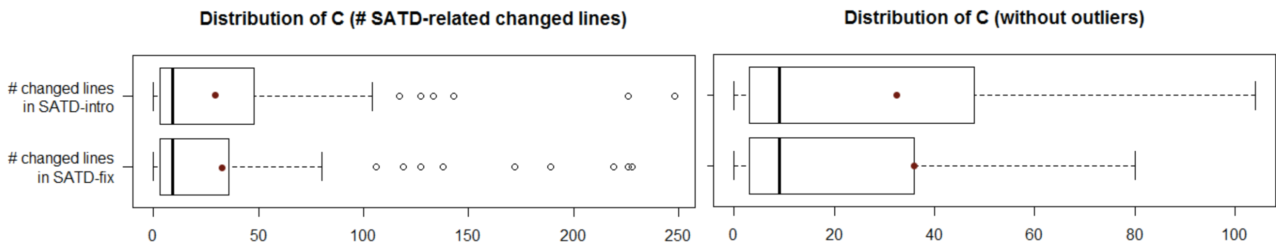


Figure 3.1: Distributions of C_A (SATD-Intro) and C_B (SATD-Fix) in JMeter, with and without outliers.

We also computed the number of SATD instances (of 80) where the set C_A of related changes in the introductory commit is larger than the set C_B of changes in the fixing commit, or vice versa, as Table 3.2 shows. From our analysis, in approximately 1/3 of the SATD the number of related changes is greater in the SATD-introduction, while in 2/3 of the SATD the number of changes is greater in the fixing.

In only 4 instances the number of changes was exactly equal in the introduction and fixing. However, in a few cases the numbers are different but very close to each other (e.g. in SATD #211, C_A=104 and C_B=105, or in SATD #257, C_A=226 and C_B=228).

By inspecting the code, we observed that in the instances that have C_A=C_B, the SATD-method was often created from scratch in the introductory commit and completely removed in the fixing commit.

A similar pattern was also observed in those cases where the numbers are very close, for example with a 2-unit difference (in case of large numbers, e.g. >30, since two small close numbers, say 4 and 6, may not be as much relevant), that is when the SATD-method was written for the first time in the introductory commit, then modified with slight increase or decrease of lines of code, and eventually removed in the fixing.

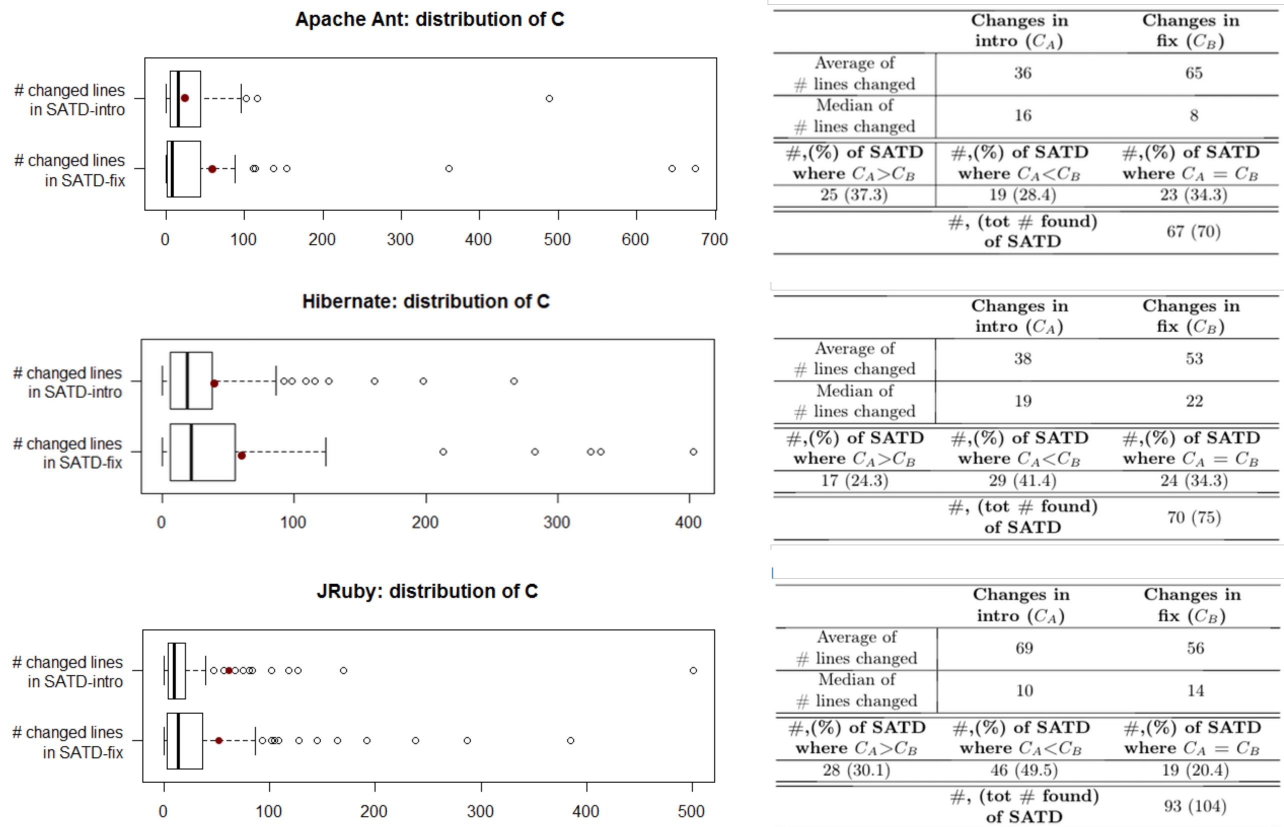
This particular behaviour of a SATD-method that is completely removed in the fixing commit will be further investigated in RQ3, where we specifically address the evolution of SATD-methods.

Table 3.1: Summary of the SATD-related changes in introductory C_A and fixing C_B commits in JMeter.

	Changes in intro (C_A)	**Changes in fix (C_B)**
Average of # lines changed	33	37
Median of # lines changed	9	9
#,(%) of SATD where C_A>C_B	**#,(%) of SATD where C_A<C_B**	**#,(%) of SATD where $C_A = C_B$**
25 (31.2)	51 (63.8)	4 (5)
	#, (tot # found) of SATD	80 (104)

3.1.2 Application to other software projects

In order to collect more data and compare results, we applied the same procedure to three more Java projects, exploiting the SATD comments classified by Maldonado [19]: Apache Ant, Hibernate and JRuby.



	Changes in intro (C_A)	**Changes in fix (C_B)**
Average of # lines changed	36	65
Median of # lines changed	16	8
#,(%) of SATD where C_A>C_B	**#,(%) of SATD where C_A<C_B**	**#,(%) of SATD where $C_A = C_B$**
25 (37.3)	19 (28.4)	23 (34.3)
	#, (tot # found) of SATD	67 (70)

	Changes in intro (C_A)	**Changes in fix (C_B)**
Average of # lines changed	38	53
Median of # lines changed	19	22
#,(%) of SATD where C_A>C_B	**#,(%) of SATD where C_A<C_B**	**#,(%) of SATD where $C_A = C_B$**
17 (24.3)	29 (41.4)	24 (34.3)
	#, (tot # found) of SATD	70 (75)

	Changes in intro (C_A)	**Changes in fix (C_B)**
Average of # lines changed	69	56
Median of # lines changed	10	14
#,(%) of SATD where C_A>C_B	**#,(%) of SATD where C_A<C_B**	**#,(%) of SATD where $C_A = C_B$**
28 (30.1)	46 (49.5)	19 (20.4)
	#, (tot # found) of SATD	93 (104)

Figure 3.2: Summary of the SATD-related changes in introductory C_A and fixing C_B commits in Ant, Hibernate and JRuby.

As the tables in Figure 3.2 show, for Hibernate, Ant and JRuby we found respectively 70 (out of 101 unique comments), 75 (out of 311) and 104 (out of 296) Design SATD comments with both introductory and fixing commits. Again, we applied our tool to both commit diff files in order to compute C_A and C_B for every SATD referring to a method (excluding single statements).

For Hibernate and JRuby, on average the set of SATD-related changes is larger in the fixing commit: in respectively 41.4% and 49.5% of the SATD instances, C_B is larger than C_A. Instead, on average C is larger in the introductory commit for Apache Ant: about 37% of the SATD instances had C_A>C_B, against 28.4% where C_A<C_B. For all three projects, however, we found that the number of SATD where $C_A = C_B$ was significantly larger than the number we had observed for JMeter: this means that they contain more SATD comments referring to methods that were introduced from scratch in the SATD-introduction commit and completely removed in the SATD-fix.

Additionally, we want to know whether the inequality between C_A and C_B is systematic or not. For this purpose, we run the Wilcoxon signed-rank test, taking as samples the paired sets of C_A and C_B for each SATD found in the four projects. The results we obtained for Ant, Hibernate, JMeter and JRuby are respectively p-values of 0.77, 0.13, 0.21 and 0.24, which are not small enough to reject the test's null hypothesis. Thus, the results we observed with the test tell us that we cannot say there is a systematic inequality between the two variables.

3.1.3 Interpretation of Results

Linares *et al.* [35] defined the impact of a class C_i in the change set $ChLOC$ as the number of methods in the difference set $ChLOC - C_i$ affected by any change in C_i. When a class is created, modified or deleted, the impact value is the number of methods modified in $ChLOC - C_i$ due to changes in C_i.

We adopt a different approach to study the change impact: for our purposes we do not consider the impact as the changes in the *class* in a commit diff file, but rather the changes related to the *method* we are addressing, as a SATD instance – unless explicitly declared to refer to an entire class – refers to a limited numbers of LOC. We consider, however, the method containing the SATD and not the smaller code blocks (e.g. `if-else` blocks, `catch` clauses, etc.), since we are interested in studying functionalities, which small blocks cannot provide.

The change impact is the set of LOC that have been changed and can be associated to the introduction or removal of a SATD. Of course, such lines can have been changed for other reasons that the SATD. This is a limitation of our approach that we cannot control, but we can discuss. For example, looking at the box-plots in Figure 3.1 and 3.2, we can observe that 75% of the SATD-methods LOC is below 50 lines (and in some cases much lower). Such value is relatively small and lets us think that the change is primarily connected to the SATD.

By expressing the numbers we found in terms of effort, when the set C_A of SATD-related changes in introductory commit is larger than the set C_B of related changes in fixing commit, we can say that more effort was required to apply changes to the code when a technical debt was introduced than when it was resolved. Vice versa, if $C_A < C_B$, fixing a SATD cost more effort than introducing it.

In three out of four software systems analysed, we observed that between 41% and 64% of the SATD occurrences, when solved, affected a larger amount of lines of code ($C_A < C_B$), against a quantity between 24% and 37% that concerned more lines of code at the time of SATD-introduction ($C_A > C_B$).

Excluding the cases where the SATD-method is introduced and completely removed ($C_A = C_B$), that is between 5% and 34% of the SATD, we can observe that between 43% and 67% of the SATD instances in the four projects under study feature a larger set C_B, meaning that paying the technical debt required more effort in those cases.

The explanation for this phenomenon may be found in the definition of technical debt, in particular of *Design debt*: unlike for example *Defect* or *Requirement debt*, which indicate problems concerning the functional behaviour of the code, *Design debt* refers to how the code was designed. A block of code including some design debt may be "ugly" or present performance issues, such as a workaround or a temporary patch for a known bug, but should accomplish its functionality either way. Therefore, 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 fix must be accurate.

3.2 RQ2: SATD-related Bug Reports

We summarise the three criteria we used to find SATD-related Bug Reports (BRs) based on their diff files.

1. Lines changed within the SATD-method;
2. Calls of the SATD-method changed in the BR diff file;
3. Calls of other methods from the SATD block, where the calls refer to methods that were modified in the BR diff file.

We run the tool containing these rules on the diff files of the BRs found after a SATD-introduction and we obtained the number of SATD-related changes for each diff file. If the tool returned 0 changed lines for a diff file, we can assume that the corresponding BR is unrelated to the SATD.

3.2.1 Testing and improving the parsing tool by manual inspection on JMeter

We used JMeter to analyse the accuracy of our tools after some improvements of the BR parsing rules.

The false positive rates refer to the values that were found after these improvements, as described in 2.6.3. In fact, the original heuristic did not take into account some common false positive patterns, which were discovered

Table 3.2: Results of the Java Tool to find SATD-related changes in JMeter, following the rules of BR discovery as in 2.6.3.

	BRs Between SATD-Intro and Fix			BRs After Fix		
	Rule 1	**Rule 2**	**Rule 3**	**Rule 1**	**Rule 2**	**Rule 3**
Total BRs with SATD-related changes	130	42	12	33	18	20
Total False Positives	0	7	4	0	3	3
False Positive rate (%)	0	16.7	33	0	16.7	15
Total #,(%) of BRs with at least one change	153 (18)			51 (13)		
Total #,(%) of BRs with no changes	699 (82)			342 (87)		
Total # of BR analysed	852			393		

by directly mining the source code after the first results: for example, the third rule often identified method calls that were overridden in the code (see Table 3.2), but had little to do with the SATD-method (e.g. the Java `toString()` function). However, considering that the first rule alone is usually able to identify most of the SATD-related changes, and combining the results found by the three rules, the overall false positive rate was acceptable, with only 11 errors out of 164 related BRs between SATD-introduction and fixing (7% FP) and 6 errors out of 57 reports after fixing (10% FP).

Note that we directly applied the improved parsing rules for the three remaining projects we used. Table 3.2 summarises the BRs data we found in JMeter according to the rules.

3.2.2 Application to other software projects

Again, we run our tool using SATD instances from software projects other than JMeter – Apache Ant, Hibernate and JRuby – and we got results in line with the one previously observed. The number of SATD-related BRs found varies, depending on the size of the projects. Combining the results from Table 2.1 and 3.2 we can see that there is a relation between the size of the project and the number of BRs: we can notice, for instance, that Hibernate – the project with the largest number of BRs found – is the one that includes most classes and SLOC.

Table 3.3: Results of the Java Tool to find SATD-related changes, applied to the projects Apache Ant, Hibernate and JRuby. The false positive rates refer to the improved BR parsing rules.

	Apache Ant		Hibernate		JRuby	
	Between	**After**	**Between**	**After**	**Between**	**After**
Total BRs with SATD-related changes	32	13	248	113	61	50
Total False Positives	4	0	36	19	7	7
False Positive rate (%)	12.5	0	14.5	16.8	13	14
Total #,(%) of BRs with at least one change	28 (12)	13 (9)	212 (21)	94 (19)	54 (5)	43 (10)
Total #,(%) of BRs with no changes	214 (88)	129 (91)	791 (79)	406 (81)	968 (95)	405 (90)
Total # of BR analysed	242	142	1003	500	1022	448

3.2.3 Interpretation of Results

With these data we were able to have a clearer view of the impact of SATD on code quality. For each SATD instance in our data sets, we counted how many BRs our tool was able to identify, keeping separate the count for BRs between SATD-introduction and fixing and for BRs between fixing and present time. Our goal was to find which of the two phases contains the largest amount of bugs, thus which is larger between *#BR[Intro-Fix]* and *#BR[Fix-Now]*.

From the previous research question, we observed that the amount of lines changed in the fixing commit is generally greater that the respective amount in the introductory commit, meaning that paying back a technical debt requires more effort on average. In fact, our hypothesis concerned the quantity of bugs observable in the two different phases: if we observe that the versions of the code after the SATD-fixing generally include more

bugs than the versions when the technical debt was present, we will be able to conclude that, despite being temporary, "ugly" or "not quite right", the phase between SATD-introduction and fixing of the code is less error-prone anyway. Therefore, it would be reasonable to preserve the technical debt, instead of wasting time and effort by solving it and probably introducing more problems than it had before. Vice versa, if the presence of SATD increased the number of bugs, it would be reasonable to fix the debt instead.

Based on the observations in RQ1, we would expect that the amount of bugs reported *after* a SATD-fixing is indeed larger than the amount *between* SATD-introduction and fixing. However, as tables 3.2 and 3.3 show, on the contrary we observed that the versions between introduction and fixing generally contain more bugs: for example, 75% of the SATD-related BRs identified in JMeter were found between introduction and fixing, against 25% bugs found after SATD-fixing. 68%, 69% and 56% of the BRs were also found between introduction and fixing respectively in Ant, Hibernate and JRuby.

To have a better overview, we counted how many bugs are related to each SATD instance (per software project), and compared the amount of bugs found between introduction and fixing with the amount found after fixing. The following table 3.4 summarises these amounts.

Table 3.4: Amount of Bugs found before and after SATD-fixing for each software project.
A: # of BRs found between SATD-introduction and fixing; B: # of BRs found after SATD-fixing.

	Ant	**Hibernate**	**JMeter**	**JRuby**	**General**
#,(%) of SATD instances where A>B	12 (17.1)	48 (64)	38 (36.6)	17 (16.4)	115 (32.6)
#,(%) of SATD instances where A<B	2 (2.9)	14 (18.7)	12 (11.5)	10 (9.6)	38 (10.8)
#,(%) of SATD instances where A=B (A, B $\neq$ 0)	1 (1.4)	7 (9.3)	7 (6.7)	5 (4.8)	20 (5.6)
#,(%) of SATD instances where A, B=0)	55 (78.6)	6 (8)	47 (45.2)	72 (63.2)	180 (51)
Tot # of SATD instances retrieved per project	70	75	104	104	353

We can notice that, in terms of number of bugs within the SATD-phase or after it, most SATD instances we considered in all projects are related to more bugs between introduction and fixing, rather than after fixing till present time. Basically, before it was fixed, a SATD comment was connected to a larger amount of bugs on average.

This means that, for between 53% and 80% (excluding the SATDs that we found related to no bugs) of the SATD instances, it is true that the presence of a SATD increased the number of bugs, and paying back the technical debt generally helped make the code less error-prone. Additionally, by RQ1 the change impact is typically larger for SATD-fixes. This implies that the ratio number of BRs over change impact is smaller for SATD-fixes than for SATD-introductions.

However, except for Hibernate, it is relevant to notice that the majority of the SATD instances we considered was not associated to any bug, neither before nor after the SATD-fixing (fourth row of the table). Also, in some cases the number of bugs we observed was equal in the two phases (e.g. for 27 SATD occurrences in JMeter).

Finally, excluding these latter cases, we observed that some SATD instances are related to a number of bugs that was reported after solving the SATD. For example, in Apache Ant only 2 SATD occurrences out of 70 are connected to more bugs reported after SATD-fixing, but almost 20% of the SATD occurrences found in Hibernate seem to have caused more bugs after fixing, meaning that the code would have probably been less "buggy" in this case, if the SATD had not been resolved.

At this point, we analysed the SATD instances where this happens and tried to understand what causes – if there is a cause for – a greater amount of bugs to be reported after a SATD-fixing. We were not able to find evidence that one or more characteristics of a SATD systematically cause an increase of errors after the debt has been solved. However, here below we report the patterns that we observed in the SATD occurrences where the "after-SATD phase" (*Group A*) includes more bugs than the "before-SATD phase" (*Group B*), meaning that these traits combined may be the reason of the bug increase.

- 92% of the SATD in Group A belong to the first case (I) according to the classification described in 2.6.1 (i.e. the comment was found inside a method block). The rest of them belongs to case III – comments referring to single statements. With respect to Group B, 75% belong to case I, 10% to case II and 15% to case III.
  It is interesting to notice this because, as we pointed out in 2.6.1, a comment found within a method block (case I) probably refers to a single operation rather than the whole method, whereas the external

comments of case II refer to the entire method, so in most cases, the reason for more faults to happen after the fixing is to be connected to few operations within a method.

- 90% of the SATD blocks in Group A had less than 100 lines of code when the SATD was introduced. 100% of the SATD blocks included less than 150 LOC. In general, the SATD blocks are of average size, considering the sizes of all SATD-methods analysed (see Figure 3.3). All the outliers we observed belong instead to Group B, which includes SATD-methods with almost 500 LOC.

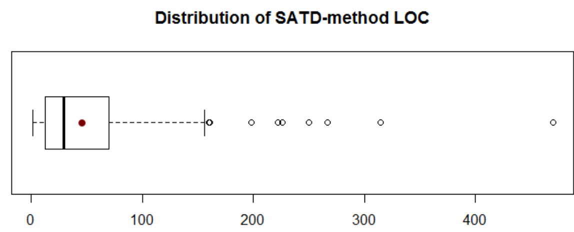


Figure 3.3: Distribution of SATD-method sizes (LOC) at the time of SATD-introduction in the four projects.

- About 70% of the SATD in Group A was subject to more changes at SATD-fixing than at introduction, as for RQ1. For Group B, the percentage is lower, 55%. Thus, most SATD instances with more bugs after fixing required greater effort to solve the SATD, indicating that such effort, expressed in terms of LOC changed, may have caused an increase of bugs.

- As for RQ3, which we discuss below, 25% of the SATD blocks in Group A grew at fixing time; 65% shrank or were completely removed; 10% remained unmodified. The results are about the same for Group B, except for a larger amount of blocks that were removed at fixing time (45%, against 15% of Group A).

It is important to mention that neither of these observations can be considered the direct cause of the bug increase on its own, since we did not observe a general, "absolute" reason for this phenomenon to happen. However, we were able to identify some common patterns that, taken together, may help classify the SATD occurrences at risk.

3.3 RQ3: Evolution of SATD-methods

In this last research question, we studied more deeply how technical debt affects source code, more specifically by analysing how an initial method M where a SATD is introduced evolves into M', the same method at the moment when the SATD is removed.

We used the tool described in 2.6.1 to parse the method a SATD comments refers to in the introductory version of the code. In order to obtain the code block of the same method in the SATD-fixing version, we applied a similar technique, only we searched for the method header we have found previously and parsed the block starting from it – since the fixing version of code does not contain the SATD comment any more.

In the projects we considered, there are a few cases of methods that where completely removed in the fixing commit, together with the code: when this happens, i.e. when the SATD-method is not found in the fixing version, it is sufficient to parse the final diff file instead (the file including the changes that led to the SATD-fixing version) and retrace the method from the changes. In the less common cases where also the method declaration changes (for example, if a new input parameter is introduced), again searching for the SATD comment in the diff file will work to detect the last available version of the method.

As we stated in the question formulation, we observed the following types of evolution:

- The method M changes during the time between introduction and fixing, and the final version has more lines of code than it had at the beginning: $M' > M$;

- On the contrary, M shrinks between SATD-introduction and fixing, and the final method M' is smaller than the first version: $M' < M$;

- The method is removed in the last version containing the SATD, for example as it was marked as *@Deprecated*, and a new method may have been introduced to replace it: $M' = 0$. This case is recognisable by examining the fixing diff file, which shows the whole method being completely cancelled. As sub-case of $M' = 0$, we hypothesised that some methods that seem removed may actually have been split into smaller methods ($M' + M'' + ... + M^N \simeq M$). In fact, we observed a case where this happens, but a manual inspection was necessary to find that, as the sub-methods usually change name and/or parameters.

- The SATD is removed without relevant changes to the method, so the method has not changed since the SATD was introduced: $M' = M$. Note that this case does not include possible methods that were changed and the final number of LOC casually remained equal to the initial one (i.e. the developers added as many lines as they removed, but the initial method M is different from the final one M').

We applied our tool to obtain the initial and final version of the SATD-methods in the four projects and analysed how many occurrences belong to each category. Table 3.5 summarises these values. The last row lists the number (and percentage of the total) of SATD comments that refers to methods, thus excludes the SATD occurrences denoting single statements – as we explained in RQ1, in section 3.1 – since we are interested in studying the evolution of code blocks.

Table 3.5: SATD instances where the original method M evolves into M' in different ways, by growing, shrinking, remaining the same or being removed.

	Ant	**Hibernate**	**JMeter**	**JRuby**	**General**
#,(%) of SATD instances where $M'>M$	10 (14.9)	8 (11.4)	19 (23.8)	14 (15.1)	51 (16.5)
#,(%) of SATD instances where $M'<M$	21 (31.3)	18 (25.7)	24 (30)	27 (29)	90 (29)
#,(%) of SATD instances where $M' = M$	7 (10.5)	9 (12.9)	3 (3.7)	8 (8.6)	27 (8.7)
#,(%) of SATD instances where $M' = 0$	29 (43.3)	35 (50)	34 (42.5)	44 (47.3)	142 (45.8)
Tot # of SATD instances referring to code blocks (tot # of SATD found)	67 (70)	70 (75)	80 (104)	93 (104)	310 (353)

We can notice that, for all the projects considered, between 25% and 31% of SATD-methods shrinks when the technical debt is resolved, while a smaller amount, between 11% and 24%, grows during the "SATD phase", until the technical debt is fixed. Also, between 42% and 50% of the methods is completely removed at fixing time. The smallest number on average is the amount of methods that 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 (in fact, several SATD had been marked as *@Deprecated* before being removed.

To better understand how a method M evolves into M', in the next pages we discuss some relevant examples taken from the projects we analysed. These examples aim to show the changes that affected the SATD-methods in their course and provide concrete chunks of code having the characteristics we observed in this section.

3.3.1 Methods that grow or shrink at SATD-fixing ($M' \neq M$)

We report two example where a method changes during the "SATD phase". The first was taken from JRuby, and we can notice that the final method is larger than the introductory one ($M'>M$), as it is visible from the second code chunk where a few more lines were added.

In these examples we omitted some code for readability purposes, but in this case the final method included 15 LOC, against the 4 when the SATD was introduced.

```
[JRuby, comment # 22]
// FIXME: Somehow I'd feel better if this could get the appropriate var index...
```

```
[commit 20225f17, Jan 13 2009]
    public void nextValue(BodyCompiler context, Object object, int index) {
      // FIXME: Somehow I'd feel better if this could get the appropriate var index...
      context.getVariableCompiler().assignLocalVariable(index, false);
    }

[commit 37355c58, Jan 5 2011]
    public void nextValue(BodyCompiler context, Object object, int index) {
-       // FIXME: Somehow I'd feel better if this could get the appropriate var index...
-       context.getVariableCompiler().assignLocalVariable(index, false);
+       ArrayNode arguments = (ArrayNode)object;
+       Node argNode = arguments.get(index);
+       switch (argNode.getNodeType()) {
+       ...
+       }
    }
```

The second, from Apache Ant, instead shows a method that was simplified, with respect to the original one of the introductory commit ($M'<M$): the `if` condition is replaced by a ternary conditional operator, which is directly inserted in the `return` statement, so now only one line is used, instead of a procedure that needed 5 lines.

`[Ant, comment # 7] // FIXME: Is "No Namespace is Empty Namespace" really OK?`

```
[commit cc6786e6, Dec 20 2005]
    private static String getNamespaceURI(Node n) {
       String uri = n.getNamespaceURI();
       if (uri == null) {
          // FIXME: Is "No Namespace is Empty Namespace" really OK?
          uri = "";
       }
       return uri;
    }

[commit b7d1e9bd, Apr 13 2017]
    private static String getNamespaceURI(Node n) {
       String uri = n.getNamespaceURI();
-      if (uri == null) {
-         // FIXME: Is "No Namespace is Empty Namespace" really OK?
-         uri = "";
-      }
-      return uri;
+      return uri == null ? "" : uri;
    }
```

3.3.2 Methods that did not change at SATD-fixing ($M' = M$)

In some cases we observed that, despite the SATD comment being removed, the corresponding method did not change when the SATD was removed (meaning that the design debt reported in the comment either did not subsist or was ignored and removed without changes to the code). The following example shows one of these methods:

`[Ant, comment # 49] // XXX - should throw an exception instead?`

```
[commit e4f0795f, Nov 18 2000]
    protected String replaceReferences(String source) {
       Vector v = reg.getGroups(source);
       result.setLength(0);
       for (int i=0; i<to.length; i++) {
          if (to[i] == '\\') {
             if (++i < to.length) {
               ...
             } else {
+               // XXX - should throw an exception instead?
                result.append('\\');
             }
          } else {
             result.append(to[i]);
          }
       }
       return result.toString();
    }

[commit 13000c1a, Jul 17 2013]
    protected String replaceReferences(String source) {
       Vector v = reg.getGroups(source);
       result.setLength(0);
       for (int i=0; i<to.length; i++) {
          if (to[i] == '\\') {
             if (++i < to.length) {
               ...
             } else {
-               // XXX - should throw an exception instead?
                result.append('\\');
             }
          } else {
             result.append(to[i]);
          }
       }
       return result.toString();
    }
```

We can observe that the SATD was indeed removed, but the code remained exactly the same and was not modified to address the design debt. In this case, the SATD comment pointed out that an exception should be thrown when the second `if` condition is not meet and the `else` part is executed. As we can see, though, the comment was only removed without the due correction.

3.3.3 Methods that were removed at SATD-fixing ($M' = 0$)

As Table 3.5 summarised, the group of methods that were cancelled at the time of SATD-fixing represents the majority of the SATD occurrences we analysed. This phenomenon can be explained by the nature of a *Design Debt*, since in order to fix design defects a code refactoring is often needed, and as we observed in RQ2, important changes to the code generally did not increase the amount of related bugs, in a slightly counter-intuitive way.

It is interesting to notice about this case that, as expected, all methods that were removed at SATD-fixing are not connected to any bug after fixing. This happens because, whether the method has been deleted or moved somewhere else with a different name, our tools are not able to track it; thus, the set C of LOC changed is empty (as it is the method) and no related bugs – if any – can be found.

Again, we observed this by manually inspecting the SATD-methods, and the results are in line with this fact: 51% of SATD instances have no bugs before or after SATD-fixing (Table 3.4), and we observed that between 63% and 75% of the SATD have no bugs after fixing (including both cases where *#BR[Intro-Fix]* > *#BR[Fix-Now]* and *#BR[Intro-Fix]* = *#BR[Fix-Now]* = 0).

We report below a couple of examples of such methods.

`[Hibernate, comment # 27] // todo :  potentially look at optimizing these two arrays`

```
[commit 5c4dacb8, Mar 20 2015]
+   public void serialize(ObjectOutputStream oos) throws IOException {
+       Status previousStatus = getPreviousStatus();
+       oos.writeObject( getEntityName() );
+       oos.writeObject( id );
+       oos.writeObject( getStatus().name() );
+       oos.writeObject( (previousStatus == null ? "" : previousStatus.name()) );
+       // todo : potentially look at optimizing these two arrays
+       oos.writeObject( loadedState );
+       oos.writeObject( getDeletedState() );
+       oos.writeObject( version );
+       oos.writeObject( getLockMode().toString() );
+       oos.writeBoolean( isExistsInDatabase() );
+       oos.writeBoolean( isBeingReplicated() );
+       oos.writeBoolean( isLoadedWithLazyPropertiesUnfetched() );
+   }

[commit 3e5a8b66, Apr 1 2015]
-   public void serialize(ObjectOutputStream oos) throws IOException {
-       Status previousStatus = getPreviousStatus();
-       oos.writeObject( getEntityName() );
-       oos.writeObject( id );
-       oos.writeObject( getStatus().name() );
-       oos.writeObject( (previousStatus == null ? "" : previousStatus.name()) );
-       // todo : potentially look at optimizing these two arrays
-       oos.writeObject( loadedState );
    ...
-       oos.writeBoolean( isLoadedWithLazyPropertiesUnfetched() );
-   }
```

`[JRuby, comment # 29] // FIXME: This isn't right for within ensured/rescued code`

```
[commit fc09ed42, Sep 26 2007]
    public void issueRedoEvent() {
+       // FIXME: This isn't right for within ensured/rescued code
        if (currentLoopLabels != null) {
            issueLoopRedo();
        } else {
            // jump back to the top of the main body of this closure
            method.go_to(scopeStart);
        }
    }
```

```
[commit 148883da, Sep 14 2008]
-   public void issueRedoEvent() {
-       // FIXME: This isn't right for within ensured/rescued code
-       if (currentLoopLabels != null) {
-           issueLoopRedo();
-       } else if (withinProtection) {
-           invokeUtilityMethod("redoJump", sig(IRubyObject.class));
-       } else {
-           // jump back to the top of the main body of this closure
-           method.go_to(scopeStart);
-       }
-   }
```

Both SATD comments were removed in the final commit. The difference between these two cases is that the former had also the introductory method inserted from scratch in the first commit, while in the latter only the comment was added in the introductory commit. This means that in the second example, the method existed even before the SATD was added, while the first was born together with the SATD.

3.3.4 Method marked as *@Deprecated* and removed at a later time

Below we report the interesting case of a SATD comment that was removed together with the related code, which can therefore be ascribed to the previous category ($M' = 0$). However, in this case the comment itself was affected by a relevant change before this happened.

```
[JMeter, comment # 87]
// 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. By further analysing the code, we observed that the last commit not only affected the SATD-method mentioned above: in fact, a general refactoring of the code was performed, which removed several other methods marked as `@deprecated`. The history of this SATD comment may be retraced as follows:

1. The //TODO was added to point out that a method was only called by a deprecated function (commit c932ee6a2 on May 16th 2012);

2. 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 18th 2016). Also the method referred to by the initial comment was marked as deprecated;

3. Finally, the whole comment, together with the now deprecated method, is suppressed (commit 3392c7314 on December 28th 2016), with a general refactoring that removed all deprecated methods.

From the case described above, we realised that not always a SATD comment is introduced and then removed as it is. In fact, 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.

3.3.5 Method split into two sub-methods at SATD-fixing

We report the only case we found where a method starts as a whole at SATD-introduction and has apparently split in two when the SATD is removed (this is commonly known as Facade pattern [44]).

[Ant, comment # 29] * two reasons are mentioned in the API docs for File.list...

```
[commit e2a75898]
    private void scandir(File dir, String vpath, boolean fast) {
       String[] newfiles = dir.list();
+      if (newfiles == null) {
+      /* two reasons are mentioned in the API docs for File.list
+      * (1) dir is not a directory. This is impossible as we wouldn't get here in case.
+      * (2) an IO error occurred (why doesn't it throw an exception then???) */
+          throw new BuildException("IO error scanning directory" + dir.getAbsolutePath());
+      }
...
    }
```

This was part of the starting method. Below, in the SATD-fixing commit, we can see the following:

```
[commit cafa34ca]
    protected void scandir(File dir, String vpath, boolean fast) {
        if (dir == null) {
            throw new BuildException("dir must not be null.");
-       } else if (!dir.exists()) {
-           throw new BuildException(dir + " doesn't exist.");
-       } else if (!dir.isDirectory()) {
-           throw new BuildException(dir + " is not a directory.");
        }
+       String[] newfiles = dir.list();
...
+       scandir(dir, vpath, fast, newfiles);
+   }
+   private void scandir(File dir, String vpath, boolean fast, String[] newfiles) {
        // avoid double scanning of directories, can only happen in fast mode
        if (fast && hasBeenScanned(vpath)) {
            return;
        }
-       String[] newfiles = dir.list();
-       if (newfiles == null) {
-       /* two reasons are mentioned in the API docs for File.list
-       * (1) dir is not a directory. This is impossible as we wouldn't get here in case.
-       * (2) an IO error occurred (why doesn't it throw an exception then???) */
-          throw new BuildException("IO error scanning directory" + dir.getAbsolutePath());
-       }
...
    }
```

The original SATD-method, with File, String and boolean as parameters, was not removed. However, a homonym method with a different modified (private) and an additional parameter (String[] newfiles, underlined in the code) was added right below the original one, which at the end invokes the second method.

This was an interesting example of how a method can evolve – not only by growing or shrinking in length, but also maintaining its functionality despite being divided in two parts.

3.3.6 SATD comments modified during their lifespan

As cases correlated to 3.3.4, we present some more examples of SATD comments that are changed during their lifespan. As we have already pointed out, not always a comment reporting a SATD is introduced and removed as it is, but there are cases where a comment is modified, often for reasons that do not depend on the SATD itself.

For example, a frequent change we observed is the shift from single-line to multi-line comments, or vice versa. In these cases, the text of the comment remains unvaried, while the portion that changes is the very beginning of the string.

`[JMeter, comment # 58] // TODO - perhaps save other items such as headers?`

```
[commit 59671c56]
    /**
-    * Save Result responseData to a set of files
-    * TODO - perhaps save other items such as headers?
     ...
+   // TODO - perhaps save other items such as headers?
    public class ResultSaver extends AbstractTestElement implements Serializable,
    SampleListener, Clearable {
     ...
```

Identifying such changes in comments is fairly easy – it is sufficient to run the `git` commands excluding the first characters (* or //, speaking of comment syntax in Java) and the blanks that follow. Also removing the `TODO` part (or `FIXME`, `XXX` or any other SATD identifier, if present) is useful to identify these cases: as we already observe, SATD comment are generally very specific to a code fragment, so skipping these portions will not compromise the results, i.e. `git log` will not be able to find very similar comments in other classes.

In other cases, however, cutting the initial part will not help: sometimes, a comment is modified to correct a typo. At this point, searching for the exact string will not allow to trace the history of the comment once it was changed.

`[Hibernate, comment # 30]`
`// todo :  remove this and just have subclasses use Isolater/IsolatedWork directly...`

```
[commit 08d9fe2117]
    public abstract class TransactionHelper {

-       // todo : remove this and just have subclasses use Isolater/IsolatedWork directly...
+       // todo : remove this and just have subclasses use IsolationDelegate directly...
...
```

`[JMeter, comment # 126] // TODO - can this eever happen?`

```
[commit 8cf39ed8]
    try {
        newNode.setEnabled(component.isEnabled());
-   } catch (Exception e) { // TODO - can this eever happen?
+   } catch (Exception e) { // TODO - can this ever happen
      newNode.setEnabled(true);
    }
```

In this case it was not difficult to find the line that fixed the typo, as it was right below the original comment. However, in our study we had to manually identify these changes, since our current tools are not able to recognise these changes automatically. In the following chapter we address this problem and give suggestions to solve it, as future improvement.

Chapter 4

Threats to Validity

4.1 Regular Expression

The tools we wrote to parse code blocks from the SATD comments (2.6.1) and to identify BRs related to SATD occurrences (2.6.3) are largely base on regular expression parsers. All regular expressions we built address code that follows Java code standards, taken from from mature codebase of open-source software projects featuring well-commented source code.

Expressions were built to be generally robust: for example, a RegEx for a method declaration works with multiple combinations of access control (`private, public, protected`) and non-access modifiers (`static, final, synchronized`...), with standard and custom return types and also with possible extra spaces between the words that compose the declaration. We tested the tool and repeatedly updated the cases that the regular expressions can match whenever we encountered a false positive, but our tests were limited to the four projects we considered. Thus, the main threat to external validity concerns the possibility that the results may not generalise: there may be cases we did not observed that the tools are not able to match, resulting in higher false positive rates.

Finally, as we wrote, the tools are currently working only for software projects written in Java, and of course should be extended in order to be applied to code developed in other languages.

4.2 Comment removal vs. SATD-fixing

We observed that SATD differs from unintentional Technical Debt, as developers document the debt they consciously introduce in form of comments.

In this work, we traced the history of a SATD by tracking the corresponding comment and localise the two versions of the code where the comment was added and then cancelled. The comments are the most evident indicator of the presence of SATD in code and we are obliged to refer to them when we search for SATD occurrences in code (since deliberate, non-documented SATD instances are only known by the developers). However, we cannot be sure that the version where a SATD-comment was added is exactly the version when also the SATD was introduced too, and same for the fixing SATD.

For example, consider a technical debt that is introduced unintentionally and an explanatory comment is added only when a developer notices it, possibly after months or years. Another case might happen, for instance, if the comment is removed by mistake, but the technical debt persists.

In a few words, adding/removing SATD-comments might not necessarily mean that the corresponding SATD instances has been introduced/fixed at the same time. Nevertheless, as we saw, comments are currently the only sign of the presence of a Self-Admitted Technical Debt.

4.3 Identification of Bug Reports related to SATD

Our approach to recognise SATD-related Bugs covered the following steps: first, we located the introductory and fixing versions of the SATD, then we listed all the BRs whose date was between SATD-introduction and fixing and between fixing and present time. Finally, we extracted the change files corresponding to each bug report and searched for SATD-related changed lines within each bug's change file (see section 2.6.3 for more details).

This was the heuristic we developed to track changes related to SATD for every bug report found. However, even if a bug-fixing change file included lines belonging to the SATD-methods or generally changes connected to the SATD, it does not mean that *the bug was directly caused by the introduction of a SATD*. Were the bugs we classified as SATD-related really caused by SATD? Vice versa, are there related bugs that we could not observe with our tools?

Confirming whether a bug is really connected to a SATD would require a much deeper analysis of the code and actual tests on the project functionalities, as simply mining the code cannot give us such information.

In any case, at present we are not aware of any studies that practically connect SATD instances and code defects (Wehaibi and Shihab [34] did work on finding SATD-related defects, but also they considered the defects in the whole files containing some SATD instance): this work was our first attempt to do so with custom tools, which we will try to improve as future challenge.

Chapter 5

Conclusion and Future Work

In this work we investigated three main aspects of the phenomenon of Self-Admitted Technical Debt: the change impacts of introducing and fixing SATD, the amount of defects found before and after SATD-fixing and the evolution of methods including SATD instances over the SATD's lifespan.

In particular, we observed that between 43% and 67% of the SATD instances, in the four open-source projects we considered, required greater effort to pay back the technical debt in terms of lines of code changed.

Moreover, excluding about half of the SATD instances in the four projects for which we could not find any related bugs (180 SATDs out of 353), 53-80% of the SATDs were connected to more bugs in in the "SATD-phase" (i.e. between SATD-introduction and fixing) than in the "after-SATD-phase". Only 14-30% of the SATDs, instead, featured more bugs after the fixing, while in 7-15% of the cases the number of bugs remained unvaried.

Finally, we grouped all SATD-methods we found according to the type of their evolution, from the introduction in the code until the fixing: we observed that the majority of the methods were completely removed at SATD-fixing (43-47%), mostly due to code refactoring. The rest of the SATD grew (11-24%), shrank (26-31%) or remained the same (4-13%) when the SATD was fixed. These values include very few non-standard cases, e.g. the single instance of a SATD-method that was divided in two parts at SATD-fixing, keeping the original functionality.

5.1 Application of Results

Back to our initial idea, we structured the study in order to investigate if we can exploit technical debt as indicator of the quality of code: in particular, if SATDs and SATD-related changes can be used to identify potentially risky patterns that may result in code defects. Knowing that some later version is more at risk of errors, can we recommend customers to keep the current version until a fixing update is available? Or can we automatically revert code to a previous version – where a SATD instance is present, but the code has proven to be more stable and functioning – if we know that the latest release causes a bug as result of an emergent event?

The results of the research questions showed that it is on average more expensive to pay back a design debt than introducing it. It is necessary to evaluate the characteristics of a SATD instance from time to time, in order to decide whether to preserve the debt or to solve it. However, we observed that fixing design debts is generally more expensive: it is up to the developer to assess if it is worth fixing a debt, despite the cost in terms of lines modified, keeping in mind that the ratio number of bugs over change impact is smaller. Thus, it is a matter of trade-off between higher change and better quality.

Answering the second question, we observed that it is on average 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. We found, however, a few instances where later versions of the code, after the SATD was fixed, that were more error-prone, so we tried to classify those. We could not find a general common pattern that causes an increase of defects after SATD-fixing, but we believe these instances of SATD are a promising path to explore: if we are able to observe that some later release caused more bugs, we can program a downgrade to a previous more stable version, both in a context of emergence in Systems of Systems and of recommending code versions to the users of a software projects.

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 base on SATD presence is feasible, affordable and brings real benefits to code quality.

5.2 Application to other projects

Maldonado and Shihab [19] provided a large data set of SATD comments from 10 projects available online, and of course there are plenty of open-source projects that probably include instances of SATD. As future work we will expand the study to other projects to further validate the results we obtained. We will also work on generalising the work by improving our analysis tools to parse non-Java projects, in order to investigate the diffusion and the impact of SATD on different languages.

5.3 Evolution of comments

In section 3.3.6 we observed that SATD comments may change as well during the lifespan of a SATD. In this work, we traced the comments in the code by searching for the exact string in the project history and we detected changes in comments only by removing a few initial and final characters of the string (e.g. section 3.3.4) or by manual inspection.

Additionally, as future work it would be useful to develop a heuristic that retraces changes in comments, for example by considering different groups of words instead of the complete string. With such a tool, we might be able to identify more important changes in comments, which are difficult find manually, and in this way to investigate more deeply the evolution of SATD comments.

Acknowledgements

Thanks to Professor Russo, for believing in my abilities and stimulating me with her enthusiasm.

Thanks to the other professors I had the chance to meet, in particular: P. Abrahamsson, J. Bowring, G. Dodero, E. Franconi, S. Helmer, W. Nutt, F. Ricci, because Computer Science can be nasty, but with you it is impossible not to love it.

Thanks to the amazing staff at UniBz and CofC: Claudia, Federica, Gorana, Mario, Melissa, Rossella, for being available to answer my questions, because bureaucracy has no secrets for you.

Thanks to my fellows Computer Science students: Aurelia, Alessandro, Daniele G. (for showing me that it's always worth trying) and V. (that grades are not everything), Edona, Erdal (for his kind words improve the day), Filippo, Florian, Francesco, Giulia (because she exists), Luca, Matteo, Mercy, Michele, Richard (because he's the best), Romeo, Saifur, Stefan, Werner (for pushing me to give more) and all the others, just because they are there when you need them and they always understand what you are going through.

Thanks to mum and dad, for supporting and loving me even though lately I am never at home.

Finally, thanks to the people I forgot, because a list of acknowledgements is just not enough.

Bibliography

[1] J. Boardman and B. Sauser. System of Systems - the meaning of of. In *2006 IEEE/SMC International Conference on System of Systems Engineering*, pages 6 pp.–, April 2006.

[2] M. W. Maier. Architecting principles for Systems of Systems. pages 167–184, 1998.

[3] D. Weyns and J. Andersson. On the Challenges of Self-adaptation in Systems of Systems. In *Proceedings of the First International Workshop on Software Engineering for Systems-of-Systems*, SESoS '13, pages 47–51, New York, NY, USA, 2013. ACM.

[4] F. Cervantes, F. Ramos, L. F. Gutiérrez, M. Occello, and J. P. Jamont. A New Approach for the Composition of Adaptive Pervasive Systems. *IEEE Systems Journal*, PP(99):1–13, 2017.

[5] E. Alonso and M. Fairbank. Emergent and Adaptive Systems of Systems. In *2013 IEEE International Conference on Systems, Man, and Cybernetics*, pages 1721–1725, Oct 2013.

[6] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[7] N. Karcanias and A. G. Hessami. Complexity and the notion of system of systems: part (I): general systems and complexity. In *2010 World Automation Congress*, pages 1–7, Sept 2010.

[8] H Thorén and Philip Gerlee. Weak emergence and complexity. In *Artificial Life XII: Proceedings of the Twelfth International Conference on the Synthesis and Simulation of Living Systems*, Cambridge, MA, USA, 2010. The MIT Press.

[9] M. A. Bedau. Weak emergence. *Noûs*, 31:375–399, 1997.

[10] D. Wachholder and C. Stary. Enabling emergent behavior in systems-of-systems through bigraph-based modeling. In *2015 10th System of Systems Engineering Conference (SoSE)*, pages 334–339, May 2015.

[11] W. Cunningham. The WyCash Portfolio Management System. *SIGPLAN OOPS Mess.*, 4(2):29–30, December 1992.

[12] S. McConnell. Technical Debt, Blog. `http://www.construx.com/10x_Software_Development/Technical_Debt/`, 2007. [accessed 12-06-2017].

[13] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, R. Sangwan, C. Seaman, K. Sullivan, and N. Zazworka. Managing Technical Debt in Software-reliant Systems. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, pages 47–52, New York, NY, USA, 2010. ACM.

[14] R. E. Fairley and M. J. Willshire. Better Now Than Later: Managing Technical Debt in Systems Development. *Computer*, 50(5):80–87, May 2017.

[15] P. Kruchten, R. L. Nord, and I. Ozkaya. Technical Debt: From Metaphor to Theory and Practice. *IEEE Software*, 29(6):18–21, Nov 2012.

[16] A. Potdar and E. Shihab. An exploratory study on self-admitted technical debt. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 91–100. IEEE, 2014.

[17] M. Fowler. Technical Debt Quadrant, Blog. `https://martinfowler.com/bliki/TechnicalDebtQuadrant.html`, 2009. [accessed 13-06-2017].

[18] N. Alves, L. Ribeiro, V. Caires, T. Mendes, and R. Spínola. Towards an ontology of terms on technical debt. In *Managing Technical Debt (MTD), 2014 Sixth International Workshop on*, pages 1–7. IEEE, 2014.

[19] E. Maldonado and E. Shihab. Detecting and quantifying different types of self-admitted technical debt. In *Managing Technical Debt (MTD), 2015 IEEE 7th International Workshop on*, pages 9–15. IEEE, 2015.

[20] R. V. Binder. *Testing Object-oriented Systems: Models, Patterns, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[21] G. Bavota and B. Russo. A large-scale empirical study on self-admitted technical debt. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 315–326. ACM, 2016.

[22] Z. M. Jiang and A. E. Hassan. Examining the Evolution of Code Comments in PostgreSQL. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, pages 179–180, New York, NY, USA, 2006. ACM.

[23] Y. Padioleau, Lin T., and Y. Zhou. Listening to Programmers Taxonomies and Characteristics of Comments in Operating System Code. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 331–341, Washington, DC, USA, 2009. IEEE Computer Society.

[24] H. Malik, I. Chowdhury, H. Tsou, Z. M. Jiang, and A. E. Hassan. Understanding the rationale for updating a function's comment. In *2008 IEEE International Conference on Software Maintenance*, pages 167–176, 2008.

[25] B. Fluri, M. Wursch, and H. C. Gall. Do Code and Comments Co-Evolve? On the Relation Between Source Code and Comment Changes. In *Proceedings of the 14th Working Conference on Reverse Engineering*, WCRE '07, pages 70–79, Washington, DC, USA, 2007. IEEE Computer Society.

[26] L. Tan, D. Yuan, G. Krishna, and Y. Zhou. /*iComment: Bugs or Bad Comments?*/. *SIGOPS Oper. Syst. Rev.*, 41(6):145–158, October 2007.

[27] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens. @tComment: Testing Javadoc Comments to Detect Comment-Code Inconsistencies. In *5th International Conference on Software Testing, Verification, and Validation*, ICST 2012, pages 260–269, Montreal, Canada, 2012.

[28] P. Kruchten, R. L. Nord, I. Ozkaya, and D. Falessi. Technical Debt: Towards a Crisper Definition. *SIGSOFT Softw. Eng. Notes*, 38(5):51–54, August 2013.

[29] N. Zazworka, R. O. Spínola, A. Vetro', F. Shull, and C. Seaman. A Case Study on Effectively Identifying Technical Debt. In *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*, EASE '13, pages 42–47, New York, NY, USA, 2013. ACM.

[30] Y. Guo, C. Seaman, R. Gomes, A. Cavalcanti, G. Tonin, F. Q. B. Da Silva, A. L. M. Santos, and C. Siebra. Tracking technical debt; an exploratory case study. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 528–531, Sept 2011.

[31] E. Lim, N. Taksande, and C. Seaman. A Balancing Act: What Software Practitioners Have to Say About Technical Debt. *IEEE Softw.*, 29(6):22–27, November 2012.

[32] Everton Maldonado, Emad Shihab, and Nikolaos Tsantalis. Using natural language processing to automatically detect self-admitted technical debt. *IEEE Transactions on Software Engineering*, 2017.

[33] Q. Huang, E. Shihab, X. Xia, D. Lo, and S. Li. Identifying self-admitted technical debt in open source projects using text mining. *Empirical Software Engineering*, pages 1–34, 2017.

[34] S. Wehaibi, E. Shihab, and L. Guerrouj. Examining the impact of self-admitted technical debt on software quality. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 179–188, March 2016.

[35] M. Linares-Vásquez, L. Cortés-Coy, J. Aponte, and D. Poshyvanyk. Changescribe: A tool for automatically generating commit messages. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 2, pages 709–712. IEEE, 2015.

[36] E. Maldonado, E. Shihab, and N. Tsantalis. Replication package for using natural language processing to automatically detect self-admitted technical debt. `https://github.com/maldonado/tse_satd_data`. [accessed 12-06-2017].

[37] D. A. Wheeler. SLOC count users guide. `https://www.dwheeler.com/sloccount/sloccount.html`, 2004. [accessed 12-06-2017].

[38] AlDanial. CLOC Git. `https://github.com/AlDanial/cloc`. [accessed 12-06-2017].

[39] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou. JDeodorant: Identification and Removal of Type-Checking Bad Smells. In *Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering*, CSMR '08, pages 329–331. IEEE Computer Society, 2008.

[40] OpenHub Homepage. `https://www.openhub.net/`. [accessed 12-06-2017].

[41] Git Commands Reference. `https://git-scm.com/docs`. [accessed 12-06-2017].

[42] Java Reflection. `https://docs.oracle.com/javase/tutorial/reflect`. [accessed 12-06-2017].

[43] M. Pancaldi and B. Russo. Replication Package. `https://github.com/martapanc/SATD-replication-package`, 2017.

[44] Facade Pattern. `https://en.wikipedia.org/wiki/Facade_pattern`. [accessed 5-07-2017].