ScrumBut as an Indicator of Process Debt

Jacinto Ramirez Lahti
Solita Ltd.
Helsinki, Finland
jacinto.ramirez@solita.fi

Antti-Pekka Tuovinen
University of Helsinki
Helsinki, Finland
antti-pekka.tuovinen@helsinki.fi

Tommi Mikkonen
University of Jyväskylä
Jyväskylä, Finland
tommi.j.mikkonen@jyu.fi

Rafael Capilla
Rey Juan Carlos University
Madrid, Spain
rafael.capilla@urjc.es

***Abstract*—Technical debt analysis is used to detect problems in a codebase. Most technical debt indicators rely on measuring the quality of the code, as developers tend to induce recurring technical debt that emerges along with evolution cycles. This debt can emerge when project pressure leads to process deviations, for instance. In agile methods like Scrum, such deviations are commonly known as ScrumButs (like Scrum but ...), which can be considered as a form of process debt. In this paper, we investigate two recurring signs of process debt (i.e. code smells and anti-patterns) caused by Scrumbuts. Our contribution investigates typical ScrumBut practices found in agile projects in one company and we report the relationships found between problems in code and ScrumBut issues. Our findings identify three types of ScrumButs, their root causes, and how these relate to concrete code smells and anti-patterns.**

***Index Terms*—Technical debt, process debt, ScrumBut, code smells, anti-patterns.**

I. Introduction

Daily changes made under pressure to software code is one of the sources of the appearance of code issues. Hence, the need for adding new features and fixing bugs in the codebase multiple times per day [1] is a recurrent problem that often leads to the appearance of technical debt [2] in its various forms (e.g. architecture or code debt [3] [4]). The appearance of technical debt in some software development contexts can exacerbate the frequency of the debt caused by bad design and programming practices. This is known as process debt [5] [6], which can be understood as a sub-optimal activity that might have short-term benefits but that generates negative consequences in a software project in the medium and long term. An example of process debt in the agile context is the appearance of ScrumButs [7], or deviations from baseline Scrum practices, explained by developers as *'like Scrum, but [description of the deviation]'*.

Other works – in particular [8] – highlight the reasons for process debt and how to prioritize the improvements of processes. There the authors define process debt as a kind of sub-optimal development activity that may lead to the appearance of technical debt. In [8], the authors present a taxonomy of process debt, causes and consequences which could be somehow related to certain anti-patterns in software development processes. Consequently, it would be interesting to investigate which of these anti-patterns has more impact on process debt.

In this paper we study how deviations of the Scrum method lead to the appearance of anti-patterns and code smells as technical debt. We investigate this phenomenon in a software startup company and we report early results of the appearance of deviations in software development practices as examples of process debt. We first describe the background stemming from our previous work [9], where we identified code smells and anti-patterns related to software design in a product of the company. Second, we outline the case study we used to find out evidence of process debt in the case company that is using Scrum. Finally, we report the main findings and conclusions of our work by relating identified anti-patterns, code smells, and ScrumButs.

II. Background

The case company is a software startup, whose product is a customer satisfaction surveying and response analysis tool providing a dashboard for visualizing the responses and their analysis. At the time of the study in 2020, the product had been actively developed for five years and the team had grown from a single hired consultant to a team of six developers. The company followed the principles of lean startup [10] and the team used Scrum as the framework for its agile development approach.

The product was a typical web application, but internally it had some conspicuous features. The implementation was a mixture of Clojure and Java code and, consequently, a mixture of two different programming paradigms, which made the Java code interfacing with Clojure unidiomatic. Considering testing, the Clojure part had a small number of old tests that were run on every build, but their coverage was low. On the Java side there were more tests, but many of them were marked as ignored and the execution of the tests was not automated. Furthermore, the Clojure parts were not actively developed any more.

Our general approach to detect and remove the technical debt in the product was the following: (i) first, detecting common code smells with static analysis tools, (ii) then, by manually investigating the occurrences in order to discover possible underlying anti-patterns as the true cause of smells, and (iii) third, remedying the problems in code by following the guidance (refactored solution) given for each anti-pattern to remove them. The role of the static analysis was to pin-point potentially problematic areas of the code that warranted detailed scrutiny by the researchers and the team members in order to confirm the smells and to analyze and remove their

underlying causes. The details of this process and the steps taken are explained in [9].

In the end, code smells and anti-patterns proved to be useful concepts for recognizing and rectifying questionable design. However, although problems were corrected, the same anti-patterns (e.g. Reinvent the Wheel, Spaghetti Code) and code smells (e.g. Long Function, Large Class) kept re-appearing in the code. This motivated us to look deeper in how the team did its work focusing on the idiosyncrasies of the process and their causes. This lead to the work reported in this paper.

III. Research Approach

In this research, we are interested in the relationship between the code smells and anti-patterns, and the appearance of ScrumBut practices. Therefore, we conducted an exploratory case study [11] to investigate the effects in process debt when anti-patterns appear as consequence of Scrum deviations and process debt and how they can be detected. We came up with the following research question:

RQ: *How are code smells and anti-patterns related to the development team's deviations from Scrum practices, so-called ScrumBut?*

Rationale: We aim at understanding why certain code smells and anti-patterns kept reappearing after they were fixed, due to the processes and practices of the development team.

Data collection and analysis: To collect and analyse data, we performed the following tasks:

(i) We used two static code analysis tools to sniff code smells, that is CodeMR[1] and IntelliJ IDEA's code inspection tool[2].
(ii) We manually analyzed the confirmed code smells.
(iii) We identified the possible anti-patterns from the smells detected.
(iv) We observed the team performing post-mortem analysis to understand why certain anti-patterns kept reappearing after their removal.

Based on the post-mortem findings and observations of the team's behavior during the project, we then identified the ScrumButs. The lead author was employed as part of the development team at the case company. He did the practical work on the codebase and observed the team and its ways of working.

IV. Preliminary Results

Upon identifying the code smells and anti-patterns, the team started to fix the identified problems. This turned out to be difficult. In numerous cases, when a fix was introduced, some of the smells and anti-patterns quickly crawled back into the codebase. In particular, these include the following cases:

- Reintroducing external libraries for purposes that were covered by other libraries (Reinvent the Wheel anti-pattern).
- Adding unnecessary complexity into areas of code that were previously cleaned up (Spaghetti Code anti-pattern).
- Continued use of long functions and large classes where more modularized designs would be more suitable (Long Function and Large Class code smells).
- Usage of (partly) duplicate code that performs a certain task in multiple places instead of extracting the functionality for common use (Divergent Change code smell).

To combat the recurring technical debt issues, the team reflected on their working practices in order to understand the relation between technical debt and the team's activities. This was done using a postmortem that produced several findings. Because the team did not follow the recommended Scrum practices, they couldn't reflect their own findings from the postmortem to their usage of Scrum, so based on the initial analysis of the team we interpret their findings against the backdrop of Scrum recommendations.

Since Scrum is a comprehensive framework rather than a collection of disconnected best practices, deviations from the Scrum recommendations are often considered potentially harmful. The term ScrumBut is commonly used to refer to the phrase "we use Scrum, but..." [7]. Hence, as a custom variant of Scrum was the baseline for the team's behavior, the deviations can be considered as the possible causes for the problems noted in the postmortem analysis. To identify possible ScrumButs, we thoroughly examined the list of ScrumButs compiled in [7] in the context of the behaviour followed by the team and we identified three key ScrumButs in their practices, corroborated by the team's own findings, as possible causes of a continued accumulation of new technical debt.

A representation of what the team's Scrum process flow looked like in a span of 8 weeks is shown in Figure 1. The team did not use Scrum Sprints but used all the Scrum events from a typical Sprint cycle in a rather inconsistent way, with only the daily remaining clearly consistent. Compared to typical agile projects, this lack of structure led to other related problems, such as neglecting testing requirements and the possibility of external interference from customers.

Upon studying the behavior of the team and customers, it turned out that there were three bad habits, formalized in the following as ScrumButs:

- No Sprints but rather development that advances in under-defined increments (No Sprints);
- The lack of a thorough testing requirement (Testing Is Not Required);
- The team would react immediately whenever the customers would request something (Customer Intervention).

These can be considered as variants of the ScrumButs *Varying Sprint Length*, *Testing in Next Sprint*, and *Direct Customer Involvement* in [7]. As a remark, we have retained the original terminology of the development team to highlight the relation to the case study.

In Table I, we summarize all the identified ScrumButs. In addition to describing the ScrumButs, the table also presents their root causes, and the relation to code smells and anti-patterns. This relation between ScrumButs and code smells

[1]https://www.codemr.co.uk/
[2]https://www.jetbrains.com/help/idea/code-inspection.html

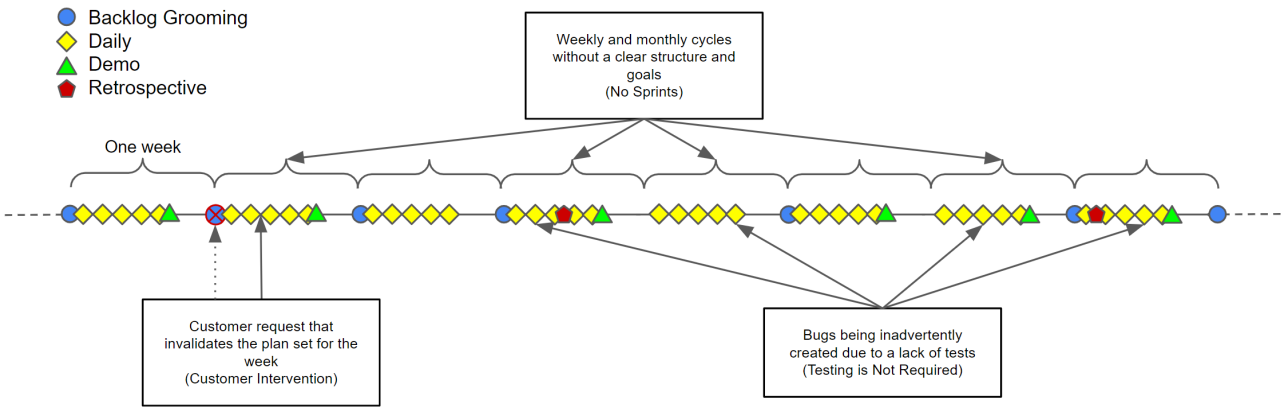

Fig. 1. An 8 week example of ScrumButs in the process of the case project. The diagram shows the lack of definition in the process being followed with no Sprints, enabling customers to disrupt the team and not requiring tests.

and anti-patterns is specific to this case and with just this data available we cannot ensure causality in either direction.

No Sprints ScrumBut: In the beginning, the case project did not use the concept of Sprints as defined by Scrum. A backlog grooming and week – instead of true sprint – planning session was held every Monday morning, and a demo was completed by Friday evenings. Daily meetings were held every morning, and a retrospective session was held once a month.

The lack of clear Sprints had an effect on the quality of the user stories in the backlog. In many cases, the stories were too big to fit in a week, but they were not divided into smaller pieces, simply because there was no burning need to do so. This meant that task allocation had to take this aspect into account. The situation was slightly improved with the introduction of a backlog management software that replaced the old spreadsheet-based backlog. This made it easier to discuss the tasks at hand and their splitting into smaller tasks. In hindsight, two-week Sprints should probably be at least tested to see if this would improve predicting when tasks would be completed.

Testing is Not Required ScrumBut: This ScrumBut, a variant of Testing in Next Sprint [7], considers the case when tests are not really required at all. This also happened in the case project; testing was never a priority during the early development phases, which led it to become the permanent state of affairs.

Little by little, the project grew to such an extent that going back that creating tests for everything would not be feasible. Introducing Java in the project added another layer of complexity, which would have required modifying test scripts to run Java tests too. As this was never done, most of the codebase had no tests at all or only manual tests, which increased the difficulty to test the code with its architecture.

Customer Intervention ScrumBut: As the company is a small startup growing rapidly, the team was very responsive for whatever demands users introduced. This service level was quickly taken as the norm by the customers, who learned that the company reacted quickly to accommodate the new needs in the system. Consequently, the development team suffered a considerable pressure from a few customers. This fact was not handled in the correct way, as these demands could put on hold other planned work on very short notice.

Not being able to follow the plan and not introducing new requirements in a managed way only harmed the development of new features that were planned for the wider customer base. Furthermore, they also affected the efforts to take care of the accrued technical debt that had been planned.

In summary, according to the technical debt quadrant[3], the majority of the debt was introduced deliberately because the team wanted to avoid the imminent cost (as time and effort) of fixing things and because of the pressure of demands from certain customers. Only the No Sprints Scrumbut happened inadvertently at first.

Limitations: To the best of our knowledge this is one of the first experiences showing the appearance of process debt caused by wrong software development practices. Although we show evidence of the connection between technical debt indicators and process debt, we reckon some limitations in our work. First, we only analyze one company so we can't generalize our results to other organizations. Second, we only investigate possible deviations the Scrum framework but there are other software development approaches that can also be analyzed. Third, we report our results on a number of code smells and anti-patterns but a deeper study should investigate how other sources of technical debt can be considered as the origin of process debt. Finally, it is needed to investigate more forms of process debt and how customer intervention or budget resources impact the occurrence of process debt.

V. Findings and Conclusion

In this work, we have studied the connection between recurring technical debt in a codebase and its relation to potential process debt. As concrete artifacts in the study, we used code smells, anti-patterns, and ScrumButs, all of which are established concepts in the field of technical debt.

[3]https://martinfowler.com/bliki/TechnicalDebtQuadrant.html

TABLE I
SCRUMBUTS IN THE CASE PROJECT, THEIR DESCRIPTIONS, ROOT CAUSES, AND THE RELATION TO CODE SMELLS AND ANTI-PATTERNS.

ScrumBut	Scrum Recommendation	Description	Root Causes	Relationship to code smells (CS) and anti-patterns (AP)
No Sprints	Sprints of fixed length and scope are used. The length of a sprint usually varies between 1-4 weeks.	The development team does not follow the Sprint model as defined by Scrum, making it more difficult for the team to focus on the work to be done in any given moment. This can also lead to tasks not being split into smaller pieces as they do not need to fit into a given sprint, making it more difficult to share the workload of a given task.	The team had rapidly grown from just a single developer to a team of six in the span of half a year. The team never considered that this huge change would require the introduction of Sprints. For convenience, they continued in their old way.	Feature Envy (CS) Temporary Field (CS) Large Class (CS) Spaghetti Code (AP)
Testing is Not Required	Testing should be done in conjunction with the implementation of the code. Each sprint produces a product that can be potentially delivered.	Testing is a fundamental part of software development in general and agile software development in particular. Unfortunately, sometimes it is ignored and seen as a nuisance, because it can introduce delays in software delivery. Not having tests will make the code more brittle and prone to future failure. Writing tests later is more difficult, which emphasises the importance of writing tests in parallel with the code they test.	The shift from the original Clojure back-end to the Clojure/Java hybrid ignored the need of configuring automated testing of the Java code. Once there was a considerable amount of code with no tests, this just became the norm. Although they realized the situation, the team still chose not to invest the time and effort required to rectify the situation.	Large Class (CS) Long Function (CS) Functional Decomposition (AP) Spaghetti Code (AP)
Customer Intervention	The team is responsible for running the Sprints, and the communication with customers happens through the backlog.	As customers can interfere with the scrum team adding not planned requirements, those in the backlog might not be properly prioritized. As these new requirements are then often implemented in an ad hoc manner, due to time pressure to go back to the original plan, these interruptions not only affect the efficacy of the team but can bring unexpected technical debt due to insufficient planning in the context of the larger plan.	Customers that had been onboard from the beginning were used to being able to influence the direction of the product very directly. Although the product and customer portfolio grew considerably, some of the customers continued to have special treatment when considering their requests. The team chose to offer their most valued customers this privilege to keep them satisfied.	Feature Envy (CS) Temporary Field (CS) Lava Flow (AP) Spaghetti Code (AP)

The main finding was that there is a clear relation between them – the recurrence of code smells and anti-patterns could be directly associated with ScrumBut. However, during the mainstream development, this connection was never realized by the team. Instead, the developers kept fixing the code without realizing that the process debt, which was a result of the deviations from Scrum that they had agreed on, was really the root cause for the need to repeatedly perform the same fixes.

The customer intervention ScrumBut was the most frequently detected, because the team was dependant on the customer. Two others were easier to eliminate, as they were caused by team's own actions, with no demand from outside. After the findings, the team made changes to try to get rid of these ScrumButs and their consequences. The team was sheltered more carefully from sudden new requirements and there was more thought put into user store creation.

As our work is still at an early phase, we seek to continue the research by studying more cases to investigate the generalizability of the results in other contexts. At the same time we seek to understand if any ScrumBut is prone to lead to technical debt, or if only some deviations from Scrum lead to such. This in turn may give an indicator whether or not process debt is intimately connected to other forms of debt in software engineering. Furthermore, it would be interesting to study the relationships and interplay of ScrumButs more thoroughly. For example, does having No Sprints really make it more likely to have Customer Intervention, or, are there No Sprints because of allowing Customer Intervention to happen are some of the questions that emerge.

REFERENCES

[1] D.G. Feitelson, E.Frachtenberg, K.L. Beck, Development and Deployment at Facebook. IEEE Internet Comput. 17(4): 8-17, 2013.

[2] W. Cunningham, The WyCash Portfolio Management System, OOPSLA'92 Experience Report, 1992.

[3] I. Pigazzini, F. Arcelli Fontana, B. Walter,A study on correlations between architectural smells and design patterns. J. Syst. Softw. 178: 110984, 2021.

[4] M. Fowler, Refactoring: improving the design of existing code, Addison-Wesley Professional, 2018.

[5] Z. Li, P. Avgeriou, P.Liang,A systematic mapping study on technical debt and its management. Journal of Systems and Software, 101: 193, 2015.

[6] A. Martini, V. Stray, N. Brede Moe, Technical-, Social- and Process Debt in Large-Scale Agile: An Exploratory Case-Study. XP Workshops, Springer LNBIP 364, 112-119, 2019.

[7] V.-P. Eloranta, K. Koskimies, T. Mikkonen, Exploring ScrumBut – an empirical study of scrum anti-patterns, Information and Software Technology 74, 194–203, 2016.

[8] A. Martini, T. Besker, J. Bosch, Process Debt: a First Exploration. 27th Asia-Pacific Software Engineering Conference (APSEC), IEEE DL, 316-325, 2020.

[9] J. Ramirez Lahti, A.-P. Tuovinen, T. Mikkonen, Experiences on managing technical debt with code smells and antipatterns. In Proceedings of TechDebt'21, IEEE, 2021.

[10] E. Ries, The Lean Startup: How today's entrepreneurs use continuous innovation to create radically successful businesses, Currency, 2011.

[11] R. K. Yin, Case Study Research Design and Methods (5th ed.), Sage, 2014.