

Discovering the Undercurrents: A Glimpse at Technical Debt in Self-Adaptive Systems



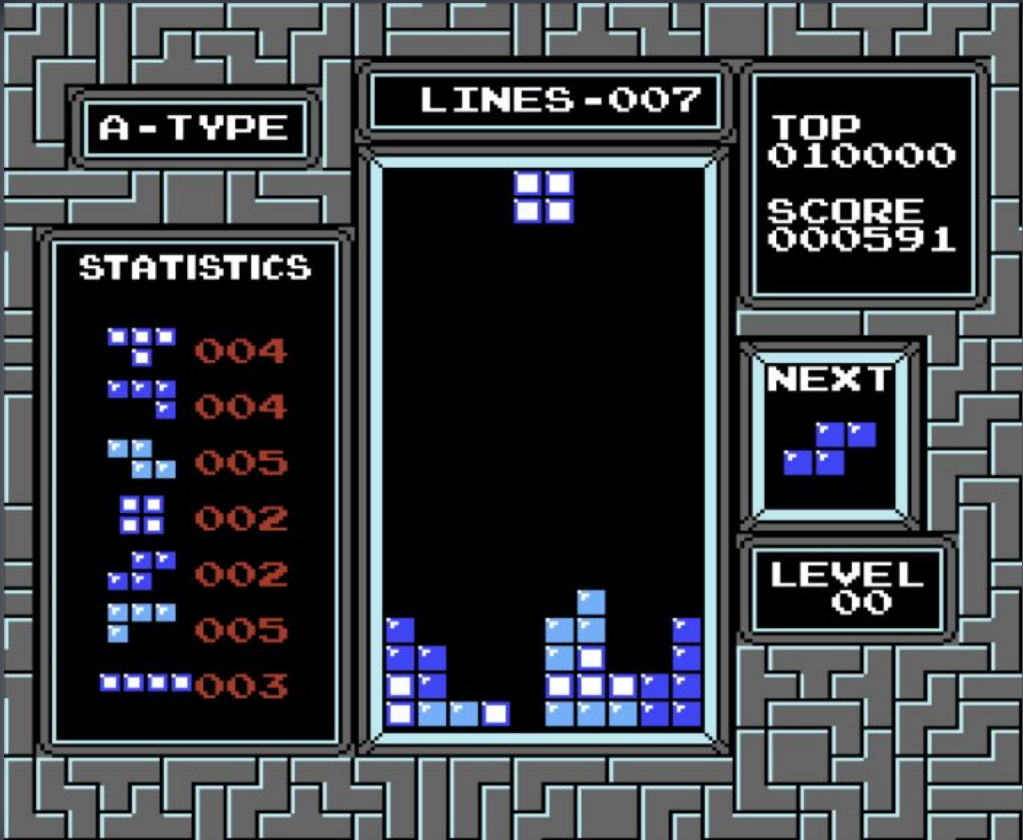
Roberto Verdecchia
University of Florence

Technical Debt is a game of Tetris™

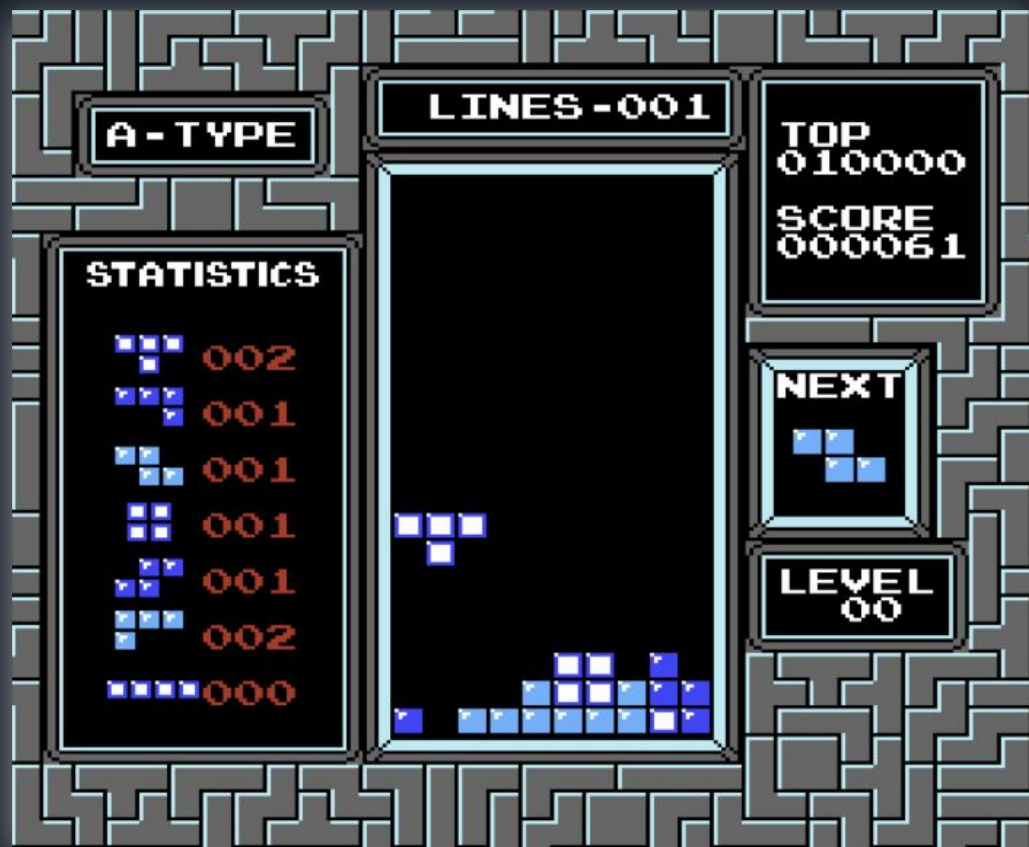


<https://medium.com/s/story/technical-debt-is-like-tetris-168f64d8b700>

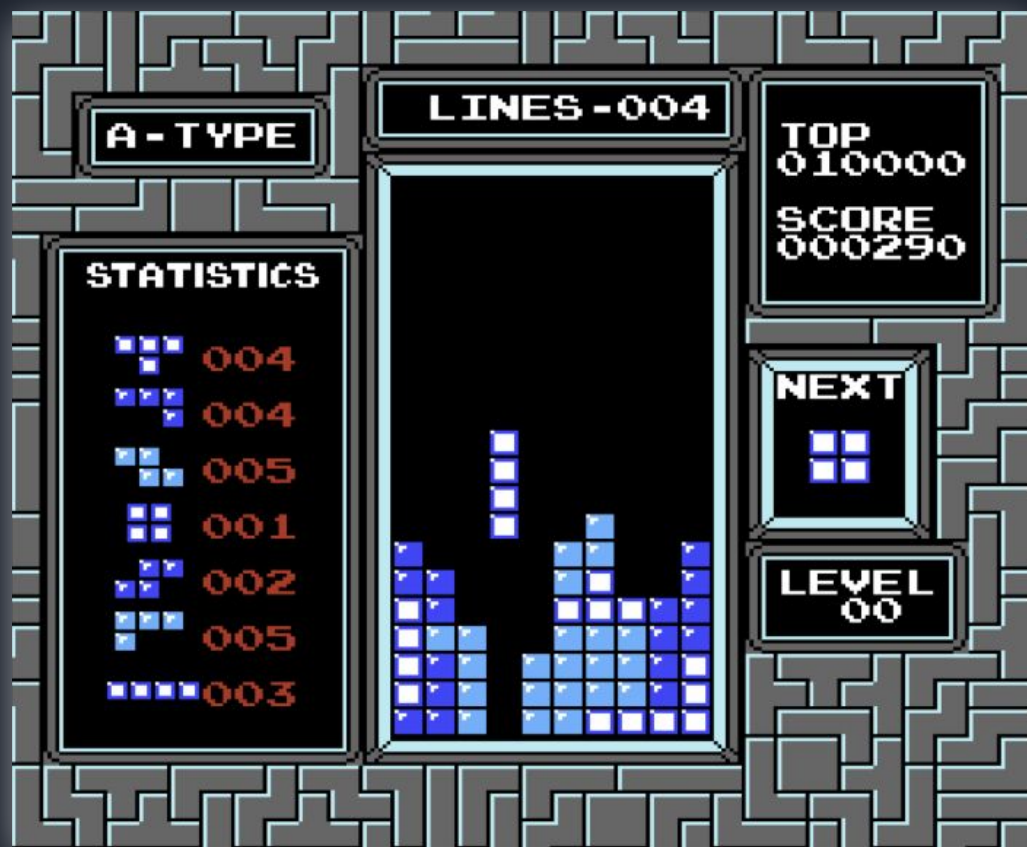
Technical Debt is a game of Tetris™



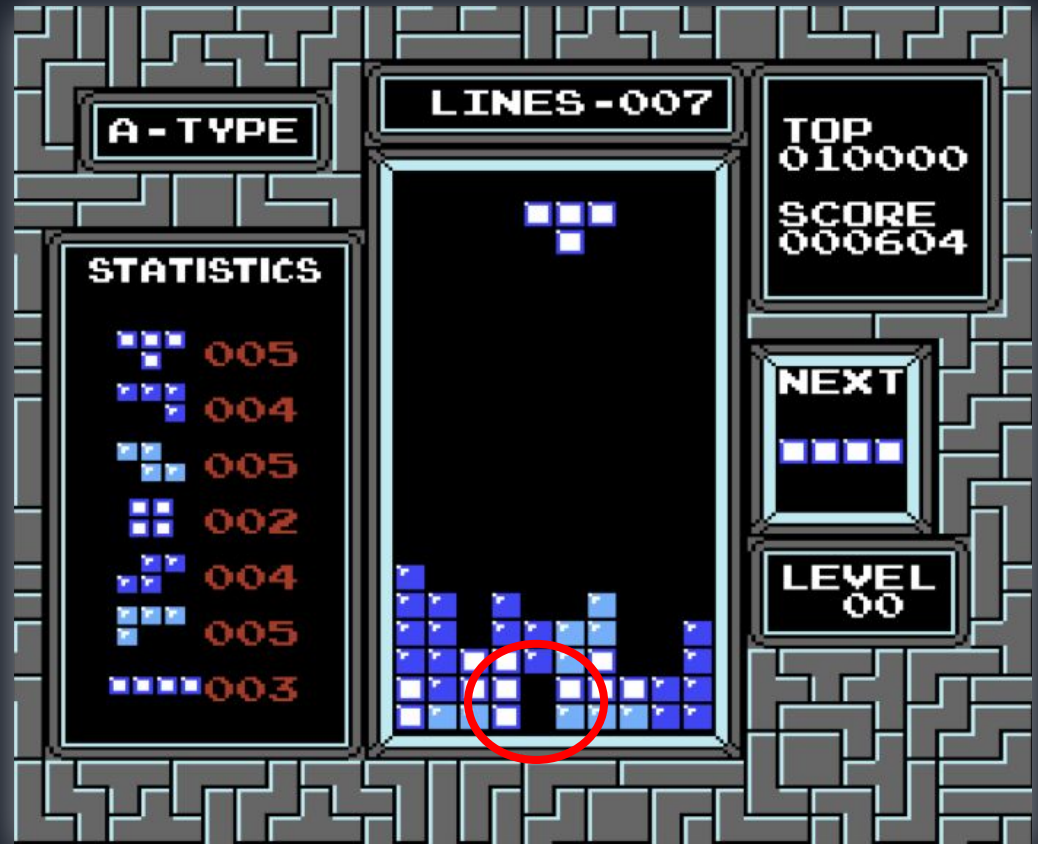
At the start
complexity is low...



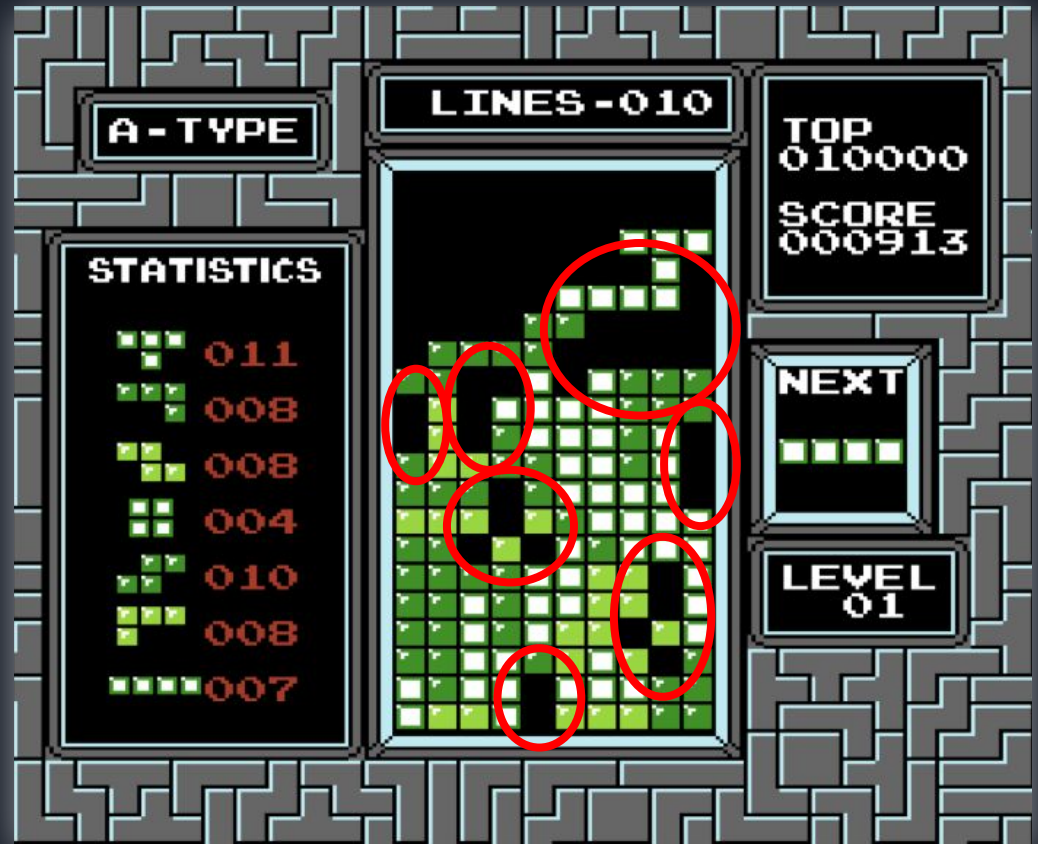
Building on the side
might be a sound
strategy



Some **gaps** make the game more difficult, but are acceptable

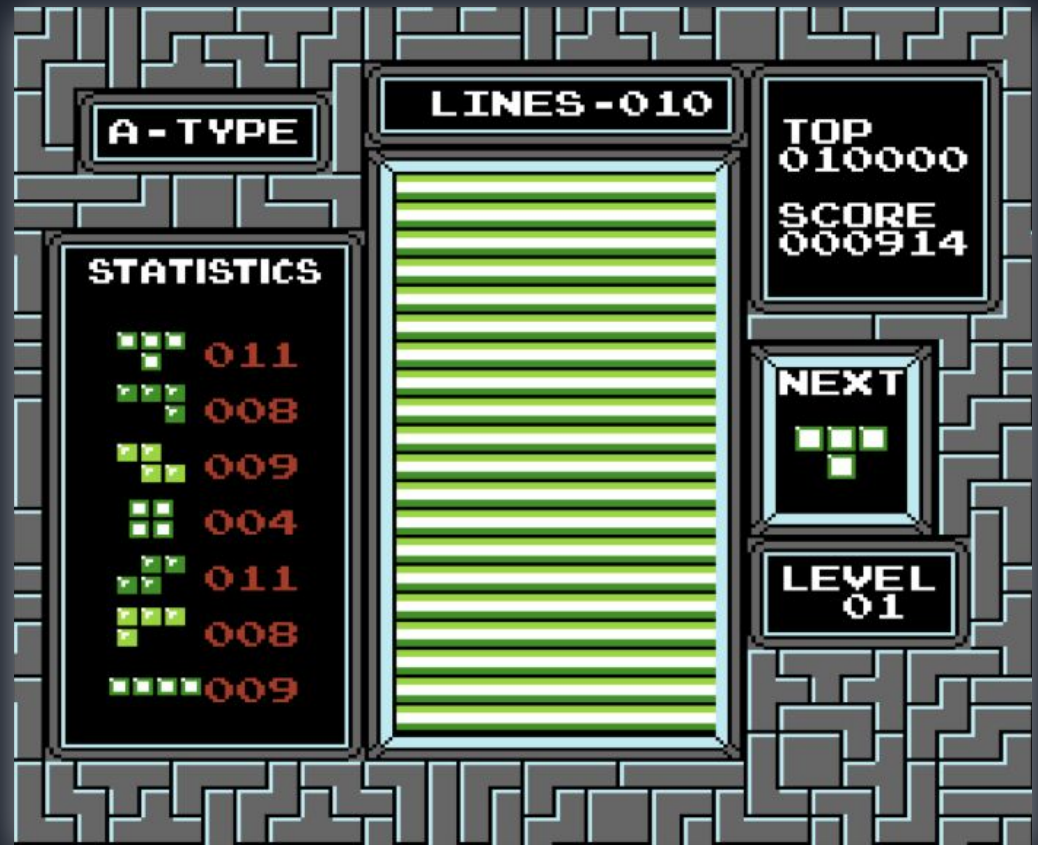


Too many **gaps** hinder positioning new blocks



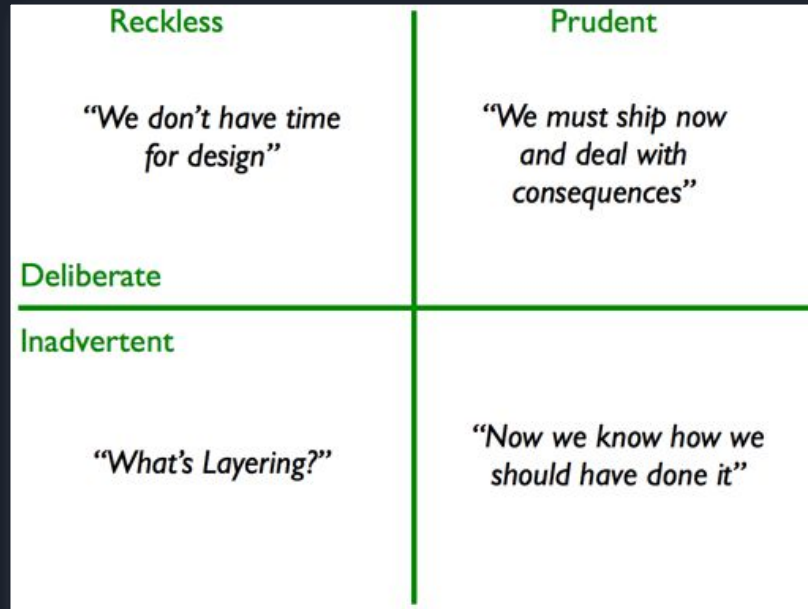
Unable to place new
blocks?

Game Over!



Technical debt core concepts

- **Technical debt principal:** Cost of fixing the short-term expedient
- **Technical debt interest:** Extra effort needed to maintain the system



Technical debt types

The Journal of Systems and Software 101 (2015) 193–220



The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss



A systematic mapping study on technical debt and its management



Zengyang Li^{a,*}, Paris Avgeriou^a, Peng Liang^{b,c}

^a Department of Mathematics and Computing Science, University of Groningen, Nijenborgh 9, 9747 AG Groningen, The Netherlands
^b State Key Lab of Software Engineering, School of Computer, Wuhan University, Luojiazuihan, 430072 Wuhan, China
^c Department of Computer Science, VU University Amsterdam, De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands

ARTICLE INFO

Article history:
 Received 11 July 2014
 Revised 10 December 2014
 Accepted 10 December 2014
 Available online 16 December 2014

Keywords:
 Systematic mapping study
 Technical debt
 Technical debt management

ABSTRACT

Context: Technical debt (TD) is a metaphor reflecting technical compromises that can yield short-term benefit but may hurt the long-term health of a software system.
Objective: This work aims at collecting studies on TD and TD management (TDM), and making a classification and thematic analysis on these studies, to obtain a comprehensive understanding on the TD concept and an overview on the current state of research on TDM.
Method: A systematic mapping study was performed to identify and analyze research on TD and its management, covering publications between 1992 and 2013.
Results: Ninety-four studies were finally selected. TD was classified into 10 types, 8 TDM activities were identified, and 29 tools for TDM were collected.
Conclusions: The term “debt” has been used in different ways by different people, which leads to ambiguous interpretation of the term. Code-related TD and its management have gained the most attention. There is a need for more empirical studies with high-quality evidence on the whole TDM process and on the application of specific TDM approaches in industrial settings. Moreover, dedicated TDM tools are needed for managing various types of TD in the whole TDM process.

© 2014 Elsevier Inc. All rights reserved.

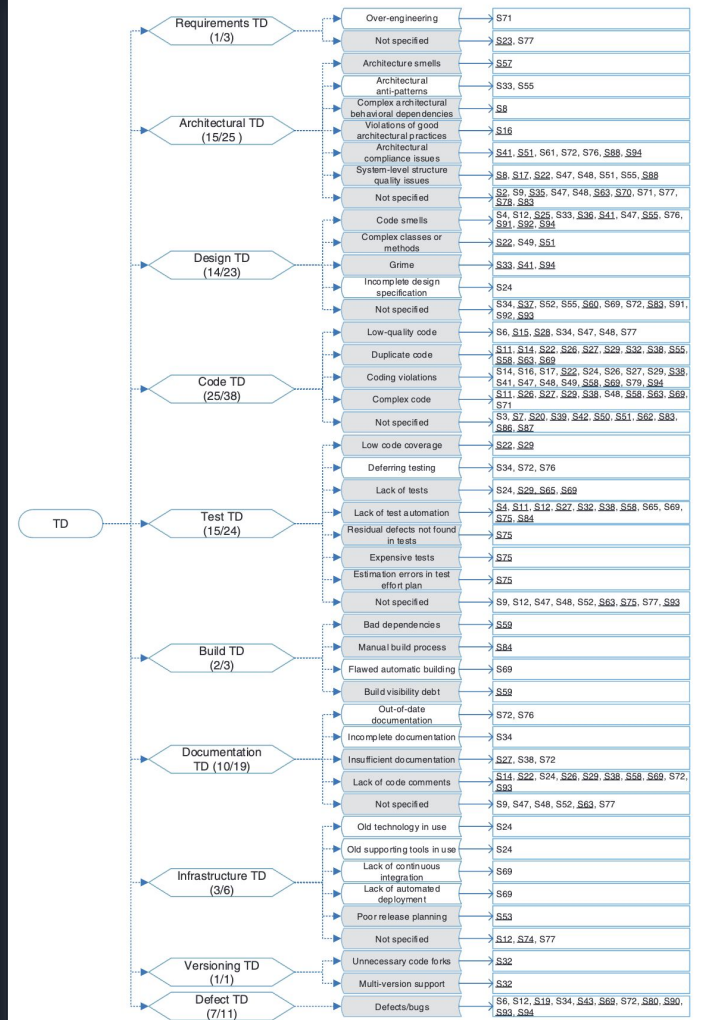
1. Introduction

Technical debt (TD) is a metaphor reflecting technical compromises that can yield short-term benefit but may hurt the long-term health of a software system. This metaphor was initially concerned with software implementation (i.e., at code level), but it has been gradually extended to software architecture, detailed design, and even documentation, requirements, and testing (Brown et al., 2010). Although the technical debt metaphor was proposed two decades ago, it has only received significant attention from researchers in the past few years.

accumulated incrementally, which in turn results in challenges for maintenance and evolution tasks.

Both intentional and unintentional TD (McConnell, 2008) should be managed in order to keep the accumulated TD under control (Lim et al., 2012). TD management (TDM) includes activities that prevent potential TD (both intentional and unintentional) from being incurred, as well as those activities that deal with the accumulated TD to make it visible and controllable, and to keep a balance between cost and value of the software project.

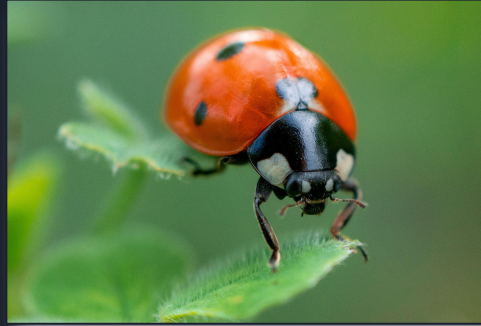
In order to systematically manage TD, it is necessary to have a clear and thorough understanding on the state of the art of TDM. Different methods and tools have been used, proposed, and developed for TDM.



Technical debt types



Code TD: complex code, code violations, dead code, ...



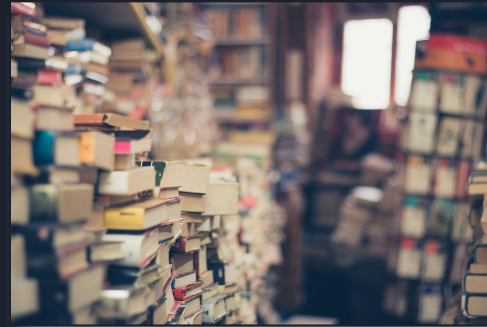
Test TD: low coverage, flaky tests, underperforming tests, ...



Architecture TD: architectural smells, technology lock-in, stuck on POF, ...



Requirements TD, e.g., ill-defined requirements, simplistic context definition...



Requirements TD, e.g., unclear requirements, simplistic context definition...

Technical debt analysis approaches: Quantitative analyses

```
...data type: "script", cache: 0, async: 1, global: 1, "throws": 0, "return": 1, ...
...insertBefore(this(0)), b.map(function({var
...each function(b)(n(this, wrapInner(a.call(this, b)))}; this.each(function
...call this, c))a));}, unwrap: function() {return this.parent().each(function
...if ("none" === Xb(a) || "hidden" === a.type) return 0; a = a.parentNode; return
...function(a) {return !n.expr.filters.hidden(a)}; var Zb = A20/g, Sb =
...test(a)(b, e); cc(a) * 1 + "object" === typeOf e && null != e ? b : "" + a;}, e, c, d));
...length - encodeURIComponent(a) + " = " + encodeURIComponent(b); if (void
...in a) cc(a, a | c, b, e); return d.join("&").replace(Zb, "+"), n.fn.extend({
...filterFunc
...array(c) ? n.
...function(a, d)
...catch(i)
...function pc()
...cache(a), a.cache
...filter(a).scr
...set(a, scr
...filter(a).scr
...filter(a).scr
```



On the diffuseness of technical debt items and accuracy of remediation time when using SonarQube

Maria Teresa Baldassarre¹, Valentina Lenarduzzi², Simone Romano³, Nyuti Saarimäki^{4*}

¹University of Bari, Italy
²LUU, University, Poland
³University, Poland

ARTICLE INFO

Keywords:
Technical debt
Remediation time
Effort estimation
SonarQube
Case study

ABSTRACT

Context. Among the static analysis tools available, SonarQube is one of the most used technical Debt (TD) items—i.e., violations of coding rules—and then estimates TD as the items. However, practitioners are still skeptical about the accuracy of remediation Objective. In this paper, we analyze both diffuseness of TD items and accuracy of rem. SonarQube, to fix TD items on a set of 21 open-source Java projects. Method. We did study where we asked 81 junior developers to fix TD items and reduce the TD of 21 pr that TD items are diffused in the analyzed projects and most items are code smells; out that the remediation time estimated by SonarQube is inaccurate and, as compares fix TD items, it is most cases overestimated. Conclusions. The results of our study are and researchers. The former can make more aware decisions during project execution the latter can use this study as a starting point for improving TD estimation models.

1. Introduction

Improving software quality requires a lot of effort, and software companies have been investing in reworking activities to remove everything that can impact the quality of their products, including technical issues [1, 2] like non-compliance with specific coding rules or with documentation conventions. Neglecting such issues can reduce the overall quality and consequently increase the Technical Debt (TD) of the entire system over time.

TD has been defined as “making technical compromises that are expedient in the short term, but that create a technical context that increases complexity and cost in the long term” [3]. Software companies usually adopt static analysis tools to measure software quality and TD [4, 5]. Among the static analysis tools available, SonarQube¹ is one of the most used—e.g., it has been adopted by more than 100K organizations including nearly 15K public open-source projects [2]. SonarQube checks for code compliance against a set of coding rules (i.e., technical issues). If the code violates any of the classified rules, SonarQube considers it a violation or a TD item. Moreover, it defines TD as the time needed (i.e., effort) to refactor the violated rules.

The diffuseness of TD items in software system TD items are present in software system in previous work [8, 9] whereas the overarch on software quality needs further attention [2]. study [12] proposed a “surgically-precise” TD enable a more precise and fine-grained lens of TD items. The results highlighted the need to keep track of the actual remediation time to fix TD items, in order to assess the accuracy of the estimated remediation time in TD tools.

In our previous work [10], we investigated the accuracy of the remediation time suggested by SonarQube to fix TD items and the diffuseness of TD items. To assess the accuracy of SonarQube’s remediation time we needed to compare the actual time with the estimated one. As so, we conducted a case study where we asked 65 junior developers to remove TD items from 15 open-source Java projects. We then compared the effort (i.e., time) developers spent to remedy TD items against the estimation proposed by SonarQube.

This paper extends the previous one [10] as follows:

- We increased the number of participants (from 65 to 81) and number

Abstract—Throughout a software development life cycle, developers knowingly commit code that is either incomplete, requires rework, produces errors, or is a temporary workaround. Such incomplete or temporary workarounds are commonly referred to as “technical debt”. Our experience indicates that self-admitted technical debt is common in software projects and may negatively impact software maintenance, however, to date very little is known about them.

Therefore, in this paper, we use source-code comments in four large open source software projects - Eclipse, Chromium OS, Apache HTTP Server, and Argo4J, to identify self-admitted technical debt. Using the identified technical debt, we study (1) the amount of self-admitted technical debt found in these projects, 2) why this self-admitted technical debt was introduced into the software projects and 3) how likely is the self-admitted technical debt to be removed after their introduction. We find that the amount of self-admitted technical debt exists in 2.4% - 31% of the files. Furthermore, we find that developers with higher experience tend to introduce most of the self-admitted technical debt and that time pressures and complexity of the code do not correlate with the amount of self-admitted technical debt. Lastly, although self-admitted technical debt is meant to be addressed or removed in the future, only between 26.3% - 63.5% of self-admitted technical debt gets removed from projects after introduction.

1. INTRODUCTION

Delivering high quality, defect-free software is the goal of all software projects. To ensure the delivery of high quality software, software project often plan their development and maintenance efforts. However, in many cases, developers are rushed into completing tasks for various reasons. A few of these reasons mentioned in prior work include, cost reduction, satisfying customers and market pressure from competition [11]. Intuition and general belief indicate that such rushed development tasks (also known as technical debt) negatively impact software maintenance and overall quality [2].

An Exploratory Study on Self-Admitted Technical Debt

Aniket Potdar
Department of Software Engineering
Rochester Institute of Technology
Rochester, NY, USA
Email: asp6719@rit.edu

Emad Sh
Department of Computer Science
Concordia U
Montreal, QC
Email: eshahib@cs

areas of the software development natural language processing inconsistencies. G relationship between s (e.g., [7], [8]) and use ity [9].

The majority of the that are due to *unintended* errors introduced by the. However, to the best of examined the impact of *intentional* (i.e., self a technical debt). Study important since they we show later in this they negatively impact. Therefore, in this p to better understand by prior work (e.g., code comments to d perform our study on Eclipse, Chromium Q focus on quantifying debt (RO2), on determ is introduced (RO2) a debt is actually remon We make the follow

- Identify common technical debt, code comments to admitted technical debt, different comment technical debt.

Arcan: a Tool for Architectural Smells Detection

Francesca Arcelli Fontana¹, Ilaria Pigazzini¹, Riccardo Roveda², Damiano Tamburini¹, Marco Zanoni¹, Elisabetta Di Nitto¹
¹Università degli Studi di Milano - Bicocca, Milan, Italy
Email: {arcelli,riccardo.oveda,marco.zanoni}@disco.unimib.it, {pigazzini@campus.unimib.it} ²Politecnico di Milano Department of Electronics, Informatica e Bioingegneria, Milan, Italy
Email: {damianoandrew.tamburini,elisabetta.dinitto}@polimi.it

Abstract—Code smells are sub-optimal coding circumstances such as blob classes or spaghetti code - they have received much attention and led to in recent software engineering research. Higher-up in the abstraction level, architectural smells are problems or sub-optimal architectural patterns or other design-level characteristics. These have received significantly less attention even though they are usually considered more critical than code smells, and harder to detect, remove, and refactor. This paper describes an open-source tool called Arcan developed for the detection of architectural smells through an evaluation of several architecture dependency issues. The detection techniques inside Arcan exploit graph database technology, allowing for high scalability in smells detection and better management of large amounts of dependencies of multiple kinds. In the scope of this paper, we focus on the evaluation of the results carried out with real-life software developers to check if the architectural smells detected by Arcan are really perceived as problems and to get an overall usefulness evaluation of the tool.

1. INTRODUCTION

It is an established fact that good software architecture and design lead to better evolvability, maintainability, availability, security, software cost reduction and more [1]. Conversely, when that architecture and design process are compromised by poor or hasted design choices, the architecture is often subject to different architectural problems or anomalies, that can lead to software faults, failures or quality downsfalls such as a progressive architecture erosion [2], [3]. A category of these anomalies is represented by architectural smells, that are caused by a violation of recognized design principles with a negative impact on internal system quality [4].

To aid the detection and removal of architecture smells (AS), this paper introduces Arcan, a static-analysis software useful to support software developers and designers during the development, maintenance and evolution of Java applications. Arcan is able to detect 3 architectural smells (i.e., Cycle Dependency, Unusable Dependency and Hub-Like Dependency). We focused our attention on AS related to dependency issues; we will consider other AS in the next future. The design of our tool relies on recent advances in graph database technology and graph computing: once a Java project has been analyzed by Arcan, a new graph database is created containing the structural dependencies of the system. Thanks to graph computing and connected big data processing technology [5], it is then possible to run detection algorithms on this graph to extract information about the analyzed project (package/class metrics, architectural issues).

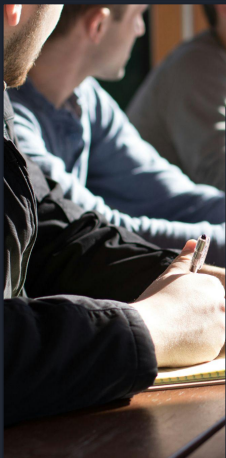
In a previous work [6], we outlined the detection algorithms hard-coded within Arcan. The original contribution of this paper is to focus on: (a) the tool’s architecture and inner workings; (b) on the improvements of its detection strategies for the Cycle Dependency smell, and (c) on the manual validation of the detection results of Arcan done by real-life software developers. The latter exercise enabled us to evaluate whether the detected architectural smells are indeed perceived as problems by the software developers responsible for them. By means of this evaluation exercise, we were able to provide a first evaluation on how the tool works and on its results in terms of precision and recall. According to the feedback of this validation, we outline and discuss future directions for research and extensions of the tool. More in particular, we learned the need to a Severity Index to identify the most critical smells to be analyzed and removed first. This Index would help developers to identify and estimate refactoring needs and their rough cost.

II. RELATED WORK

As previously stated, many tools have been developed for code smells detection but only few tools are currently available for architectural smells detection. The following briefly reports on some of them. First, the commercial tool InSpire² and its evolution in ARiverceer³ support the detection of both code smells and some design or architectural smells. Another commercial tool is Designite⁴ that detects several design smells in C# projects. The Hosted Detector [7] tool detects five architectural smells, called HotSpot Patterns, four patterns defined at file level and one at package level. Other tool prototypes have been proposed, e.g., SCODP [8], and one from Garcia et al. [9]. We outline that ARiverceer and Designite are commercial tools and according to our knowledge the other tools are not yet publicly available. Moreover, there are different commercial tools as for example Sotograph, Sonargraph, Structure 101 and Cast which are able to detect different kinds of architectural violations, as dependency cycles. Our tool (available at <http://essee.disco.unimib.it/wiki/arcan>) by analyzing compiled Java files, detects three AS. We compare the Cycle Dependency AS among classes and packages, and we detect it according to different shapes. We exploit different

¹InfoNotes, <http://www.infonotes.com/products/infofind>
²<http://www.inviser.com>
³<http://www.ariverceer.com>

Technical debt analysis approaches: Qualitative analyses



Information and Software Technology 139 (2021) 106669

Contents lists available at ScienceDirect

Information and Software Technology

ELSEVIER

journal homepage: www.elsevier.com/locate/infsof

Architectural design decisions that incur technical debt — An introductory study

Mohamed Soliman^a, Paris Avgeriou^a, Yikun Li^b
^aBernold Institute for Mathematics, Computer Science and Artificial Intelligence, University of Groningen, Groningen, The Netherlands

ARTICLE INFO

Keywords:
Technical debt
Architectural design decisions
Architectural knowledge
Architectural technical debt

ABSTRACT

Context: During software development, some architectural debt is incurred deliberately or inadvertently. These have serious impact on the significant time and effort to be changed. While current research architectural design decisions and technical debt separately, debt-incurring decisions have not been specifically explored in practice.

Objective: In this case study, we explore debt-incurring architectural decisions. Specifically, we explore the main types of DADDs, why and how do practitioners deal with these types of design decisions.

Method: We performed interviews and a focus group with practitioners from software companies, discussing their concrete experience with such decisions.

Results: We provide the following contributions: 1) A categorized current ontology on architectural design decisions; 2) A process model; 3) A conceptual model which shows the relationships between the DADDs; 4) The main factors that influence the way of dealing with DADDs.

Conclusion: The results can support the development of new architectural debt management from the perspective of Design Decisions. More architectural knowledge related to DADDs.

1. Introduction

Architectural design decisions (ADDs) have the biggest impact on the quality of a software system, and they are hard to change after their implementation [1]. Some ADDs incur technical debt, i.e. they “set up a technical context that can make future changes more costly or impossible” [2]. We call these Debt-Incurring Architectural Design Decisions (DADDs), and their impact is well recognized by both practitioners and researchers [3,4]. DADDs can be either deliberate or inadvertent [5]. Deliberate DADDs are taken because of time pressure or lack of resources: a solution is chosen that is quicker and cheaper but compromises maintainability and evolvability. For example, instead of adhering to the layered structure of the architecture, shortcuts are created that bypass layers. This results in implementing the required features quicker, but those shortcuts create ripple effects when making changes.

Inadvertent DADDs, are decisions that, when taken, do not bear any

effort on maintenance, thus become obsolete after a few years. Inadvertent DADDs becomes obsolete after a few years because of an optimal decision in the past, and unnecessary complexity.

Related research work on an empirically explored different types of architectural design decisions (ADDs) [6], their causes, trends [2] and effects [2]. Moreover, methods have been proposed to identify ATD (e.g. through capturing architectural bad smells from existing systems) [10]. Nevertheless, current studies have not examined ATD from the perspective of the Architecture Design Decisions (ADDs) that incur it either deliberately or inadvertently. This perspective is of paramount importance to inform the development of approaches to manage ATD, as well as tools to support the decision making process.

In this paper, we aim at exploring the current state of practice in industry regarding DADDs: determining types of DADDs, we study the

The Journal of Systems & Software 176 (2021) 110925

Contents lists available at ScienceDirect

The Journal of Systems & Software

ELSEVIER

journal homepage: www.elsevier.com/locate/jss

Building and evaluating a theory of architectural technical debt in software-intensive systems^a

Roberto Verdecchia^a, Philippe Kruchten^b, Patricia Lago^{b,c}, Ivano Malavolta^{a,*}

^aVrije Universiteit Amsterdam, The Netherlands
^bUniversity of British Columbia, Vancouver, Canada
^cChalmers University of Technology, Gothenburg, Sweden

ARTICLE INFO

Article history:
Received 22 December 2020
Accepted 5 February 2021
Available online 27 February 2021

Keywords:
Software engineering
Software architecture
Technical debt
Software evolution
Grounded theory
Focus group

ABSTRACT

Architectural technical debt in software-intensive systems is a metaphor design decisions (e.g. choices regarding structure, frameworks, technologies being suitable or even optimal when made, significantly hinder progress types of debt, such as code-level technical debt, can be readily detected by refactored with minimal or only incremental efforts, architectural debt wide-ranging remediation cost, daunting, and often avoided.

In this study, we aim at developing a better understanding of how software systems conceptualize architectural debt, and how they deal with it. In order to we apply a mixed empirical method, constituted by a grounded theory study. With the grounded theory method we construct a theory on architectural qualitative data from software architects and senior technical staff from a software development organizations. We applied the focus group method theory and refine it according to the new data collected.

The result of the study, i.e. a theory emerging from the gathered data, passing conceptual model of architectural technical debt, identifying and its symptoms, causes, consequences, management strategies, and common conducted focus groups, we assessed that the theory adheres to the four of grounded theory, i.e. the theory fits its underlying data, is able to work as new data appears.

By grounding the findings in empirical evidence, the theory provides novelties with novel knowledge on the crucial factors of architectural technical debt.

© 2021 The Author(s). Published by Elsevier Inc. This is an open access article under the <http://creativecommons.org/licenses/by-nc-nd/4.0/>

1. Introduction

Technical Debt (TD) is a concept that has been with us for a long time, at least since 1992 when Cunningham crafted the phrase (Cunningham, 1992), but it only got some real attention from researchers in the last 10 years (Lidzinska et al., 2011). What is technical debt? In software-intensive systems, technical debt consists of design or implementation constructs that are expedient in the short term, but set up a technical context that can make a future change more costly or impossible. Technical debt is a contingent liability whose impact is limited to internal system

qualities, primarily maintainability and evolvability (Lidzinska et al., 2011).

Technical debt can take many different forms, and can be found in many different contexts (Lidzinska et al., 2011). While much of the literature able to address code-level technical debt (e.g., frameworks, packages, libraries, etc.), systems grow in size and their lifespan many of these original design choices limit future evolution or even prevent developers to find workarounds and offer

* Corresponding author.
E-mail address: verdecchia@vu.nl (R. Verdecchia), p.lago@chalmers.se (P. Kruchten).

Journal of Systems & Software

Contents lists available at ScienceDirect

Journal of Computer Programming

ELSEVIER

journal homepage: www.elsevier.com/locate/jscico

Technical Debt tracking: Current state of practice A survey and multiple case study in 15 large organizations

Antonio Martini^{a,b,*}, Terese Besker^c, Jan Bosch^c

^aCA Technologies Strategic Research Team Barcelona, Spain
^bUniversity of Oslo, Programming and Software Engineering, Oslo, Norway
^cComputer Science and Engineering, Chalmers University of Technology, Gothenburg, Sweden

ARTICLE INFO

Article history:
Received 5 November 2017
Received in revised form 20 March 2018
Accepted 25 March 2018
Available online 29 March 2018

Keywords:
Technical Debt
Change management
Software process improvement
Survey
Multiple case study

Large software companies need to support continuous and fast delivery of customer value both in the short and long term. However, this can be hindered if both the evolution and maintenance of existing systems are hampered by Technical Debt. Although a lot of theoretical work on Technical Debt has been produced recently, its practical management lacks empirical studies. In this paper, we investigate the state of practice in several companies to understand what the cost of managing TD is, what tools are used to track TD, and how a tracking process is introduced in practice. We combined two phases: a survey involving 226 respondents from 15 organizations and an in-depth multiple case study in three organizations including 15 interviews and 79 Technical Debt issues. We selected the organizations where Technical Debt was better tracked in order to distill best practices. We found that the development time dedicated to managing Technical Debt is substantial (an average of 25% of the overall development), but mostly not systematic: only a few participants (20%) use a tool, and only 72% methodically track Technical Debt. We found that the most used and effective tools are currently backlogs and static analyzers. By studying the approaches in the companies participating in the case study, we report how companies start tracking Technical Debt and what the initial benefits and challenges are. Finally, we propose a Strategic Adoption Model for the introduction of tracking Technical Debt in software organizations.

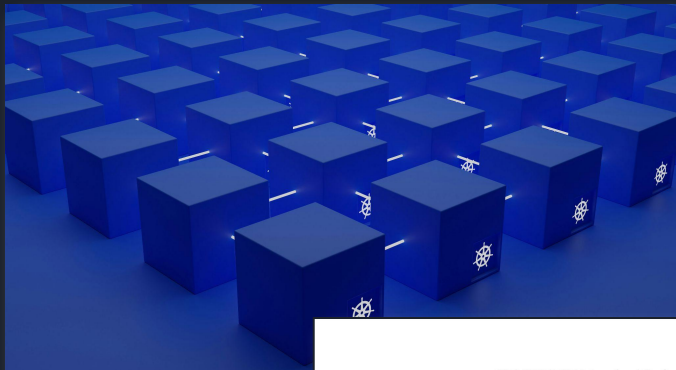
© 2018 The Authors. Published by Elsevier B.V. This is an open access article under the [CC BY-NC-ND license \(http://creativecommons.org/licenses/by-nc-nd/4.0/\)](http://creativecommons.org/licenses/by-nc-nd/4.0/).

1. Introduction

Large software companies need to support continuous and fast delivery of customer value both in the short and long term. However, this can be hindered if both the evolution and maintenance of the systems are hampered by Technical Debt.

Technical Debt (TD) has been studied recently in the software engineering literature [1–3]. TD is composed of a debt, which is a sub-optimal technical solution that leads to short-term benefits as well as to the future payment of interest, which is the extra cost due to the presence of TD (for example, slow feature development or low quality) [3]. The principal is regarded as the cost of refactoring TD. Although accumulating Technical Debt might prove useful in some cases, in others, the interest might largely surpass the short-term gain, for example, by causing development crises in the long term [3].

Technical debt is context specific



Architectural Technical Debt in Microservices A case study in a large company

Saulo S. de Toledo
Dept. of Informatics
University of Oslo
Oslo, Norway
Email: saulos@ifi.uio.no

Antonio Martini
Dept. of Informatics
University of Oslo
Oslo, Norway
Email: antonima@ifi.uio.no

Agata Przybyszewska
Dept. of Computer Science
IT University of Copenhagen
Copenhagen, Denmark
Email: agpr@itu.dk

Dag I.K. Sjovold
Dept. of Informatics
University of Oslo
Oslo, Norway
Email: dagsj@ifi.uio.no

Abstract—Introduction: Software companies aim to achieve continuous delivery to constantly provide value to their customers. A popular strategy is to use microservices architecture. However, such an architecture is also subject to debt, which hinders the continuous delivery process and thus negatively affects the software released to the customers.

Objectives: The aim of this study is to identify issues, solutions and risks related to Architecture Technical Debt in microservices.

Method: We conducted an exploratory case study of a real life project with about 1000 services in a large, international company. Through qualitative analysis of documents and interviews, we investigated Architecture Technical Debt in the communication layer of a system with microservices architecture.

Results: Our main contributions are a list of Architecture Technical Debt issues specific for the communication layer in a system with microservices architecture, as well as their associated negative impact (interest), a solution to repay the debt and its cost (principal). Among the found Architecture Technical Debt issues were the existence of business logic in the communication layer and a high amount of point-to-point connections between services. The studied solution consists of the implementation of different canonical models specific to different domains, the removal of business logic from the communication layer, and migration from services to use the communication layer correctly. We also contributed with a list of possible risks that can affect the payment of the debt, as lack of funding and inadequate prioritization.

Conclusion: We found issues, solutions and possible risks that are specific for microservices architectures not yet encountered in

on long-standing systems using such technology [1]. This study holds also with respect to what is defined for context, as a good microservices architecture and, consequently, sub-optimal solutions can lead to costly Architecture Technical Debt.

Architecture in microservices is in fact based on the combination of qualities and structural features that are different from traditional systems. An example of this is the use of a collection self-contained microservices connected by a messaging system usually called communication layer. However, knowledge about what is either a virtuous or a harmful design of such architectures is still missing [2], especially for evidence collected in a systematic research fashion and in the context of well-established industrial systems.

Understanding more about the negative effect (interest) of Architectural Technical Debt (ATD), about possible solutions and their cost (principal of ATD) would be useful for organizations and development teams that adopt microservices. Consequently, our research questions are:

- RQ1:** What is ATD in microservices?
- RQ1.1:** What is the negative impact (interest) generated by ATD in microservices?
- RQ1.2:** What is a solution for the identified ATD in microservices and its associated refactoring cost (principal)? We have investigated a large company developing financial

The Journal of Systems & Software 169 (2020) 110710

Contents lists available at ScienceDirect

ELSEVIER

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

NEW TRENDS AND IDEAS

Does migrating a monolithic system to microservices decrease the technical debt?

Valentina Lenaruzzi^a, Francesco Lorenzini^b

^a LUT University, Finland
^b Tampere University, Finland

ARTICLE INFO

Article history:
Received 20 February 2019
Received in revised form 7 May 2020
Accepted 26 June 2020
Available online 3 July 2020

Keywords:
Technical debt
Architectural debt
Code quality
Microservices
Refactoring

ABSTRACT

Background:
of the system before a Method project system perform an initial short paper

Technical Debt in Microservices:
A Mixed-Method Case Study

Roberto Verdecchia^a, Kevin Maggi^a,
Leonardo Sommezzano^b, and Enrico Vicario^b

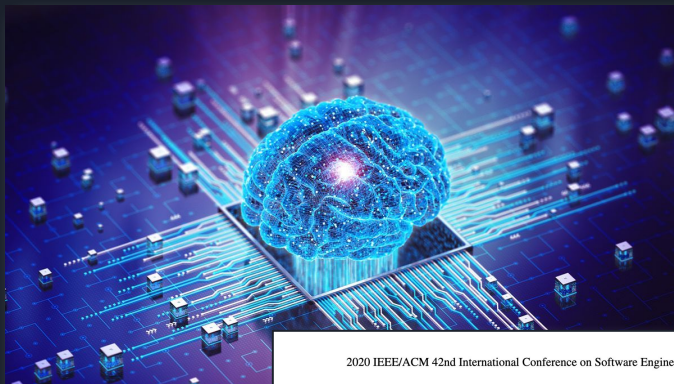
Department of Information Engineering, University of Florence, Italy
roberto.verdecchia@unifi.it,
kevin.maggi@edu.unifi.it, leonardo.sommezzano@unifi.it, enrico.vicario@unifi.it

Abstract. *Background:* Despite the rising interest of both academia and industry in microservice-based architectures and technical debt, the landscape remains uncharted when it comes to exploring the technical debt evolution in software systems built on this architecture. *Aims:* This study aims to unravel how technical debt evolves in software-intensive systems that utilize microservice architecture, focusing on (i) the patterns of its evolution, and (ii) the correlation between technical debt and the number of microservices. *Method:* We employ a mixed-method case study on an application with 13 microservices, 977 commits, and 38k lines of code. Our approach combines repository mining, automated code analysis, and manual inspection. The findings are discussed with the lead developer in a semi-structured interview, followed by a reflexive thematic analysis. *Results:* Despite periods of no TD growth, TD generally increases over time. TD variations can occur irrespective of microservice count or commit activity. TD and microservice numbers are often correlated. Adding or removing a microservice impacts TD similarly, regardless of existing microservice count. *Conclusions:* Developers must be cautious about the potential technical debt they might introduce, irrespective of the development activity conducted or the number of microservices involved. Maintaining steady technical debt during prolonged periods of time is possible, but growth, particularly during innovative phases, may be unavoidable. While monitoring technical debt is the key to start managing it, technical debt code analysis tools must be used wisely, as their output always necessitates also a qualitative system understanding to gain the complete picture.

Keywords: Technical Debt · Microservices · Software Evolution

1. Introduction.....
2. Background.....
2.1. Microservices.....
2.2. Technical debt.....

Technical debt is context specific



2020 IEEE/ACM 42nd International Conference on Software Engineering: Software E



Is Using Deep Learning Frameworks Characterizing Technical Debt in Deep Learning

Jiakun Liu[†]
Zhejiang University
College of Computer Science and
Technology
Hangzhou, Zhejiang, China
jkliau@zju.edu.cn

Qiao Huang
Zhejiang University
College of Computer Science and
Technology
Hangzhou, Zhejiang, China
tkdshoop@zju.edu.cn

Emad Shihab
Concordia University
Department of Computer Science and
Software Engineering
Montreal, Canada
eshihab@encs.concordia.ca

David Lo
Singapore Management University
School of Information Systems
Singapore
davidlo@smu.edu.sg

ABSTRACT

Developers of deep learning applications (shortened as application developers) commonly use deep learning frameworks in their projects. However, due to time pressure, market competition, and cost reduction, developers of deep learning frameworks (shortened as framework developers) often have to sacrifice software quality to satisfy a shorter completion time. This practice leads to technical debt in deep learning frameworks, which results in the increasing burden to both the application developers and the framework developers in future development.

In this paper, we analyze the comments indicating technical debt (self-admitted technical debt) in 7 of the most popular open-source deep learning frameworks. Although framework developers are aware of such technical debt, typically the application developers are not. We find that: 1) there is a significant number of technical debt in all the studied deep learning frameworks. 2) there is design debt, defect debt, documentation debt, test debt, requirement debt,

applications. Based on our findings, we highlight future research directions and provide recommendations for practitioners.

CCS CONCEPTS

• Software and its engineering → Software evolution: Maintaining software.

KEYWORDS

Self-admitted Technical Debt, Deep Learning, Categorization, Empirical Study

ACM Reference Format:

Jiakun Liu, Qiao Huang, Xin Xia, Emad Shihab, David Lo, and Shanning Li. 2020. Is Using Deep Learning Frameworks Free? Characterizing Technical Debt in Deep Learning Frameworks. In *Software Engineering in Society (ICSE-SEIS'20)*, May 23–29, 2020, Seoul, Republic of Korea. ACM, New York, NY, USA, 10 pages. <https://doi.org/10.1145/3377815.3381377>

1 INTRODUCTION

2021 IEEE/ACM International Conference on Technical Debt (TechDebt)

Characterizing Technical Debt and Antipatterns in AI-Based Systems: A Systematic Mapping Study

Justus Bogner*
University of Stuttgart
Institute of Software Engineering
Stuttgart, Germany
justus.bogner@iste.uni-stuttgart.de

Roberto Verdecchia*
Vrije Universiteit Amsterdam
Department of Computer Science
Amsterdam, The Netherlands
r.verdecchia@vu.nl

Ilias Gerostathopoulos*
Vrije Universiteit Amsterdam
Department of Computer Science
Amsterdam, The Netherlands
i.gerostathopoulos@vu.nl

Abstract—Background: With the rising popularity of Intelligence (AI), there is a growing need to build complex AI-based systems in a cost-effective and in way. Like with traditional software, Technical Debt emerge naturally over time in these systems, therefore challenges and risks if not managed appropriately. The of data science and the stochastic nature of AI-based may also lead to new types of TD or antipatterns, which yet fully understood by researchers and practitioners. The goal of our study is to provide a clear overview characterization of the types of TD (both established ones) that appear in AI-based systems, as well as the and related solutions that have been proposed. **Method:** the process of a systematic mapping study. 21 prima are identified and analyzed. **Results:** Our results show established TD types, variations of them, and four types (data, model, configuration, and ethics debt) as in AI-based systems. (ii) 72 antipatterns are discussed literature, the majority related to data and model debt (iii) 46 solutions have been proposed, either specific TD types, antipatterns, or TD in general. **Conclusions:** Our results can support AI professionals with reason and communicating aspects of TD present in their systems. Additionally, they can serve as a foundation for future research to further our understanding of TD in AI-based systems. **Index Terms—Artificial Intelligence, Machine Learning, Systematic Mapping Study.**

Hangzhou, Zhejiang, China
shan@zju.edu.cn

shanning@zju.edu.cn



ELSEVIER

The Journal of Systems and Software 216 (2024) 112151

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss



Technical debt in AI-enabled systems: On the prevalence, severity, impact, and management strategies for code and architecture*

Gilberto Recupito^{*,†}, Fabio Pecorelli[‡], Gemma Catolino[‡], Valentina Lenarduzzi[§], Davide Taibi[§], Dario Di Nucci[‡], Fabio Palomba[‡]

^{*} Software Engineering (SeSE) Lab — University of Salerno, Salerno, Italy

[†] University of Oulu, Oulu, Finland

ARTICLE INFO

Keywords:
AI technical debt
Software quality
Survey studies
Software engineering for artificial intelligence
Empirical software engineering

ABSTRACT

Context: Artificial Intelligence (AI) is pervasive in several application domains and promises to be even more diffused in the next decades. Developing high-quality AI-enabled systems — software systems embedding one or multiple AI components, algorithms, and models — could introduce critical challenges for mitigating specific risks related to the systems' quality. Such development alone is insufficient to fully address socio-technical consequences and the need for rapid adaptation to evolutionary changes. Recent work proposed the concept of AI technical debt, a potential liability concerned with developing AI-enabled systems whose impact can affect the overall systems' quality. While the problem of AI technical debt is rapidly gaining the attention of the software engineering research community, scientific knowledge that contributes to understanding and managing the matter is still limited.

Objective: In this paper, we leverage the expertise of practitioners to offer useful insights to the research community, aiming to enhance researchers' awareness about the detection and mitigation of AI technical debt. Our ultimate goal is to empower practitioners by providing them with tools and methods. Additionally, our study sheds light on novel aspects that practitioners might not be fully acquainted with, contributing to a deeper understanding of the subject.

Method: We develop a survey study featuring 53 AI practitioners, in which we collect information on the practical prevalence, severity, and impact of AI technical debt issues affecting the code and the architecture other than the strategies applied by practitioners to identify and mitigate them.

Results: The key findings of the study reveal the multiple impacts that AI technical debt issues may have on the quality of AI-enabled systems (e.g., the high negative impact that *Undeclared consumers* has on security, whereas *Jumped Model Architecture* can induce the code to be hard to maintain) and the little support practitioners have to deal with them, limited to apply manual effort for identification and refactoring.

Technical debt is context specific

Runtime Adaptation

ICPE'18, April 9–13, 2018, Berlin, Germany



To Adapt or Not to Adapt? Technical Debt and Learning Driven Self-Adaptation for Managing Runtime Performance

Tao Chen
Department of Computing and
Technology, Nottingham Trent
University, UK;
CERCIA, School of Computer Science,
University of Birmingham, UK
t.chen@cs.bham.ac.uk

Rami Bahsoon, Shuo Wang
CERCIA, School of Computer Science,
University of Birmingham, UK
r.bahsoon,s.wang@cs.bham.ac.uk

Xin Yao
Department of Computer Science and
Engineering, Southern University of
Science and Technology, China;
CERCIA, School of Computer Science,
University of Birmingham, UK
x.yao@cs.bham.ac.uk

ABSTRACT

Self-adaptive system (SAS) can adapt itself to optimize various key performance indicators in response to the dynamics and uncertainty in environment. In this paper, we present Debt Learning Driven Adaptation (DLDA), an framework that dynamically determines when and whether to adapt the SAS at runtime. DLDA leverages the temporal adaptation debt, a notion derived from the technical debt metaphor, to quantify the time-varying money that the SAS carries in relation to its performance and Service Level Agreements. We designed a temporal net debt driven labeling to label whether it is economically healthier to adapt the SAS (or not) in a circumstance, based on which an online machine learning classifier learns the correlation, and then predicts whether to adapt under the future circumstances. We conducted comprehensive experiments to evaluate DLDA with two different planners, using 5 online machine learning classifiers, and in comparison to 4 state-of-the-art debt-oblivious triggering approaches. The results reveal the effectiveness and superiority of DLDA according to different metrics.

CCS CONCEPTS

• Software and its engineering → Software performance;

thread pool size and cache size, etc), to continually optimize for different key performance indicators, e.g., response time and energy consumption, under changing environment such as dynamic workload [16] [31]. SAS often operate under formally negotiated legal binding [33][18], e.g., Service Level Agreements (SLA) [3], especially in paradigms such as services and cloud computing. This binding allows us to translate the performance of SAS into a more intuitive monetary way, e.g., instead of saying the SAS's response time is 2s in average, we are able to state the SAS creates a total of \$54 profit (or debt) for the owner. The real money that the SAS carries (either as profit or debt) determines its economic health.

While majority of SAS research has focused on the runtime *planning* phase of the SAS that determines *what and how to adapt* (e.g., rule-based [7], search-based [11][29][12] or control theoretic planners [32]), there is little research that explicitly tackles the challenge of *when and whether to adapt* the SAS, i.e., how to design the trigger [31]. We argue that deciding on when adaptation should be triggered is also non-trivial [31], because the effectiveness of the diverse planners can vary with the changing *circumstances*, i.e., SAS's status and environment conditions. Even if we assume perfect planning, it still comes with cost, e.g., planning delay and extra resource/energy consumptions, etc. The key problem, which



Technical Debt in SAS: The State of the art

- **When and whether to adapt a Self Adaptive System?**
- **TD modeling**
 - **Principal cost:** Cost of adaptation
 - **Interest:** Penalty due to inability to react to the changing environment
- **Goal:** Adapting the SAS if and only if it can make the SAS economically healthier than not adapting it
- **Approach:**
 - Train a binary classifier
 - Set of temporal debt labels
 - Decide whether it is economically healthier to adapt a SAS or not

Runtime Adaptation

ICPE'18, April 9–13, 2018, Berlin, Germany



To Adapt or Not to Adapt? Technical Debt and Learning Driven Self-Adaptation for Managing Runtime Performance

Tao Chen

Department of Computing and
Technology, Nottingham Trent
University, UK;

CERCIA, School of Computer Science,
University of Birmingham, UK
t.chen@cs.bham.ac.uk

Rami Bahsoon, Shuo Wang

CERCIA, School of Computer Science,
University of Birmingham, UK
{r.bahsoon,s.wang}@cs.bham.ac.uk

Xin Yao

Department of Computer Science and
Engineering, Southern University of
Science and Technology, China;

CERCIA, School of Computer Science,
University of Birmingham, UK
x.yao@cs.bham.ac.uk

ABSTRACT

Self-adaptive system (SAS) can adapt itself to optimize various key performance indicators in response to the dynamics and uncertainty in environment. In this paper, we present Debt Learning Driven Adaptation (DLDA), an framework that dynamically determines when and whether to adapt the SAS at runtime. DLDA leverages the temporal adaptation debt, a notion derived from the technical debt metaphor, to quantify the time-varying money that the SAS carries in relation to its performance and Service Level Agreements. We designed a temporal net debt driven labeling to label whether it is economically healthier to adapt the SAS (or not) in a circumstance, based on which an online machine learning classifier learns the correlation, and then predicts whether to adapt under the future circumstances. We conducted comprehensive experiments to evaluate DLDA with two different planners, using 5 online machine learning classifiers, and in comparison to 4 state-of-the-art debt-

thread pool size and cache size, etc), to continually optimize for different key performance indicators, e.g., response time and energy consumption, under changing environment such as dynamic workload [16] [31]. SAS often operate under formally negotiated legal binding [33][18], e.g., Service Level Agreements (SLA) [3], especially in paradigms such as services and cloud computing. This binding allows us to translate the performance of SAS into a more intuitive monetary way, e.g., instead of saying the SAS's response time is 2s in average, we are able to state the SAS creates a total of \$54 profit (or debt) for the owner. The real money that the SAS carries (either as profit or debt) determines its economic health.

While majority of SAS research has focused on the runtime **planning** phase of the SAS that determines *what and how to adapt* (e.g., rule-based [7], search-based [11][29][12] or control theoretic planners [32]), there is little research that explicitly tackles the challenge of *when and whether to adapt* the SAS, i.e., how to design the **trigger** [31]. We argue that deciding on when adaptation should

Just one piece of a much bigger puzzle...



Trivial low hanging fruits: “General” TD in SAS

- Gravitating around the generic question:
Do TD types showcase peculiarities in SAS?
- Code TD:
 - *Are some rule violations more recurrent in SAS?*
 - *Which SAS components are more affected by code TD?*
 - *What are the most recurrent causes of complex code in SAS?*
- Self-admitted TD (SATD) in SAS:
 - *What is the recurrence of SATD items in SAS?*
 - *To what extent are state of the art SATD tools effective in SAS?*



Trivial low hanging fruits: “General” TD in SAS

- Architecture TD:
 - *Which architectural smells are more recurrent in SAS?*
 - *Are there common technology lock-ins in SAS?*
- Infrastructure TD:
 - *Do some recurrent DevOps processes require manual intervention in SAS?*
 - *Are some deprecated technologies still widely used in SAS?*
- ...



Going beyond: TD of SAS

- Understand, identify, monitor, and manage TD *specific to SAS*
- First step: Characterizing TD of SAS
 - **Interest:** Penalty due to inability to react to the changing environment
 - **Principal:** Cost of adaptation... + ?
- Starts from personal knowledge, intuition, and systematic qualitative data collection



Going beyond: TD of SAS

- **TD specific to SAS environment:**
 - Ill-understood or rushed environment definition
 - Coarse environmental condition monitoring
 - Outdated environment modeling
 -



Going beyond: TD of SAS

- **TD specific to SAS adaptation:**
 - Rushed adaptation policy
 - Over adapting for the sake of simplicity
 - Choosing ill-suited adaptation strategy (periodic vs event-driven adaptation)
 - Simplistic event prediction
 -



Going beyond: TD of SAS

- **TD specific to *Service Agreement Levels and KPIs of SAS*:**
 - Misaligned KPIs and SLAs
 - Over-optimizing for SLA compliance
 - Overemphasis on easily measurable KPIs
 - Overly granular KPIs
 - Ignoring KPI interdependencies leading to conflicting adaptations
 - Inadequate KPI evolution w.r.t. changing business goals
 -



An aerial photograph of a river with white water rapids, showing turbulent water and foam. The image is dark and serves as a background for the text.

***How is technical debt
shaping current self-adaptive
systems, and how can we
manage it?***

Consider submitting to TechDebt'25















- **Flagship conference** on Technical Debt
- **Co-located** with ICSE
- **Proceedings paper types:**
 - **Research/experience** (up to 10 pages)
 - **Short** (up to 5 pages)
- **Deadline November 15**



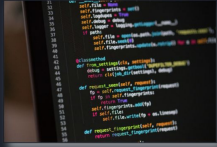
TechDebt 2025 Sat 26 April - Sun 4 May 2025 Ottawa, Ontario, Canada co-located with ICSE 2025

Attending - Call for Papers Organization - Search Series - Sign in Sign up

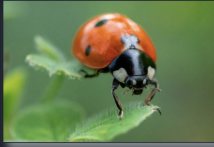
Organizing Committee TechDebt 2025

 Alexander Serebrenik General Chair Eindhoven University of Technology Netherlands	 Maria Teresa Baldassarre PC Co-Chair Department of Computer Science, University of Bari Italy
 Roberto Verdecchia PC Co-Chair University of Florence Italy	 Antonio Curci Web Chair University of Bari Italy
 Marco Tulio Valente Journal First Co-Chair Federal University of Minas Gerais, Brazil	 Maxime Lamothe Journal First Co-Chair Polytechnique Montreal Canada
 Simona Motogna Proceedings Chair Babes-Bolyai University, Cluj-Napoca Romania	 Nicolás E. Díaz Ferreyra Publicity Co-Chair Hamburg University of Technology Germany
 Neil Ernst MIP Co-Chair University of Victoria Canada	 Paris Avgeriou MIP Co-Chair University of Groningen, The Netherlands Netherlands
 Zadia Codabux Shadow PC Co-Chair University of Saskatchewan Canada	 Ipek Ozkaya Mentor, Coordinator of the Mentoring Program Carnegie Mellon University United States
 Carolyn Seaman Mentor University of Maryland Baltimore County United States	 Christoph Treude Mentor Singapore Management University Singapore

Technical debt types



Code TD: complex code, code violations, dead code, ...



Test TD: low coverage, flaky tests, underperformance tests, ...



Architecture TD: architectural smells, technology lock-in, stuck on POF, ...

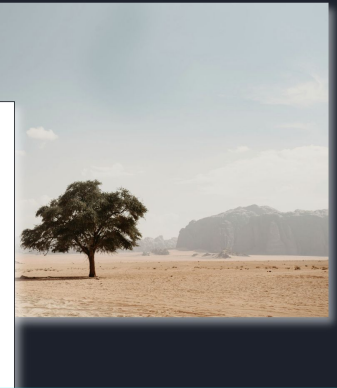
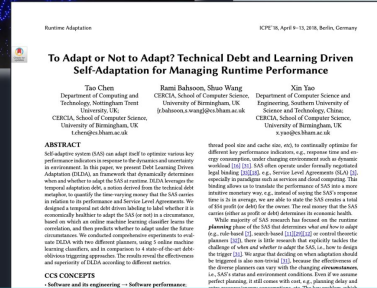
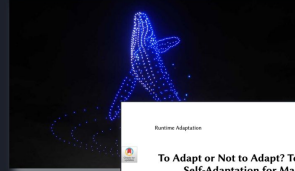


Requirements TD, e.g., ill-defined requirements, simplistic context definition...



Requirements TD, e.g., ill-defined requirements, simplistic context definition...

Technical debt is context specific



Trivial low hanging fruits: “General” TD in SAS

- Based on the generic question: *Do TD types showcase peculiarities in SAS?*
- Code TD:
 - Are some rule violations more recurrent in SAS?
 - Which SAS components are more affected by code TD?
 - What are the most recurrent causes of complex code in SAS?
- Self-admitted TD (SATD) in SAS:
 - What is the recurrence of SATD items in SAS?
 - To what extent are state of the art SATD tools effective in SAS?



Going beyond: TD of SAS

- TD specific to *Service Agreement Levels and KPIs of SAS*:
 - Misaligned KPIs and SLAs
 - Over-optimizing for SLA Compliance
 - Overemphasis on easily measurable KPIs
 - Overly Granular KPIs
 - Ignoring KPI Interdependencies leading to conflicting adaptations
 - Inadequate KPI evolution w.r.t. changing business goals
 -

