The Hidden Cost of Backward Compatibility: When Deprecation Turns into Technical Debt - An Experience Report

Anders Sundelin
anders.sundelin@bth.se
Blekinge Institute of Technology
Karlskrona, Sweden

Javier Gonzalez-Huerta
javier.gonzalez.huerta@bth.se
Blekinge Institute of Technology
Karlskrona, Sweden

Krzysztof Wnuk
krzysztof.wnuk@bth.se
Blekinge Institute of Technology
Karlskrona, Sweden

ABSTRACT

Context The micro-services architectural pattern advocates for the partitioning of functionality into loosely coupled services, which should be backward compatible, to enable independent upgrades. Deprecation is commonly used as a tool to manage multiple versions of methods or services. However, deprecation carries a cost in that tests might be duplicated and might rely on services that have become deprecated over time.

Objective Using the terms of the Technical Debt metaphor, we explore the consequences of deprecation, and how it has affected the test base during seven years.

Method We take an exploratory approach, reporting on experiences found before and after servicing parts of the incurred Technical Debt. We mine code repositories and validate our findings with experienced developers.

Results We found that the growth of deprecation debt varied a lot. Some services experienced substantial growth, but most did not. Unit tests, where deprecation is visible in the developers' tools, were much less affected than integration tests, which lack such visualization mechanisms. While servicing debt of 121 out of 285 deprecated services, we discovered that up to 29% of the spent effort could be attributed to accrued interest. However, this is an upper bound; there could be less impact, depending on whether scripting could be used to service the debt or not.

Conclusion This paper illustrates that integration tests can be viewed as a debt from the perspective of deprecated services. While the pattern was that deprecated services (debt principal) experienced no or little accrued interest, some, highly used, services experienced a lot, particularly during stressful times. Java-based tests, where deprecation is visible in the IDE, did not experience a similar pattern of increasing debt. We postulate that deprecation debt should be kept visible, either using developer tools or statistical reports.

CCS CONCEPTS

• **Software and its engineering** → **Software evolution**; *Cloud computing*; *Software verification*; Agile software development.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

TechDebt '20, October 8–9, 2020, Seoul, Republic of Korea

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-7960-1/20/05...\$15.00

<https://doi.org/10.1145/3387906.3388629>

KEYWORDS

backwards compatibility, deprecation, technical debt, automated test base

ACM Reference Format:

Anders Sundelin, Javier Gonzalez-Huerta, and Krzysztof Wnuk. 2020. The Hidden Cost of Backward Compatibility: When Deprecation Turns into Technical Debt - An Experience Report. In *International Conference on Technical Debt (TechDebt '20)*, October 8–9, 2020, Seoul, Republic of Korea. ACM, New York, NY, USA, 10 pages. <https://doi.org/10.1145/3387906.3388629>

1 INTRODUCTION

The current trend in software engineering is to split functionality into small, focused, loosely coupled services, often called microservices [6], using the DevOps moniker [3]. To achieve loose coupling, services should be independently developed. Zimmermann et al. [13] highlight that continuous upgrade-ability and backward compatibility are necessary for achieving loose coupling because they allow clients to upgrade to newer available services at will, irrespective of how the service is upgraded.

However, we have not found any study that explores the converse problem — the importance of removing unused services, and how to visualize whether or not the clients use “the most current” version of a service.

Exposing several versions of the same service carries a cost, both from a development and verification point of view. The test base can be seen as “the first client” of a service, as it is the first code that exercises the service in order to verify the expected behavior.

This paper describes the impact of service deprecation on an existing test base, and the effect that this has had on a product over the course of seven years. We take an exploratory approach and mine the Git repository of the studied product for data related to the exposed services.

A core principle used when developing new versions of the services in the studied system has been backward compatibility. A new version of service should behave identical to the currently existing version, though it might accept other types of data, or return more or less data, based on the given input. Deprecation markers have been used to flag which versions to avoid. The studied system contains one non-deprecated version and N (≥ 0) other deprecated versions of any published service.

The paper is structured as follows: In chapter 2, we cover the related work in the area. In chapter 3, we report on the research methodology, and relate to the background of the studied system. In chapter 4, we report the results found during the study. In chapter 5, we discuss the threats to validity, and in chapter 6, we draw conclusions and elaborate on the implications for the research area and industry in general.

2 RELATED WORK

Deprecation is a way to signal to API consumers that there are other, more up-to-date ways to perform a given task. Morgenthaler [2], views deprecation as software aging made manifest, and maps this directly to technical debt.

A complication is that very often, deprecated methods or services remain for very long times in APIs. As an example, the Java Standard Library class `java.util.Date` has contained four deprecated constructors and 18 deprecated methods since the release of Java 1.1 in 1997. Today (in Java 12¹), these deprecated methods and constructors are still present, together with two non-deprecated constructors and 11 non-deprecated public methods.

Sawant et al. [9] studied how developers (API consumers) in open-source projects react to deprecation events in popular Java APIs. The overall conclusion is that developers seldom react at all, and rarely keep up with API evolution. A weakness of this study is that it focuses only on open-source projects. In [10], the same authors, apart from increasing the number of studied projects, also conducted a survey, trying to reach out to developers on proprietary projects. The conclusion in this paper is similar to the prior study, and respondents point to the perceived complexity and time consumption as the main reasons for not upgrading to the latest version of the APIs and keep using deprecated versions.

The same finding was made in a study by Kula et al. [8], who investigate the efforts of library migration covering over 4600 GitHub software projects and 2700 library dependencies. Results from this study show that 81.5% of the projects kept their outdated dependencies, citing unawareness, extra effort, and added responsibility.

As of Java 9, the deprecated annotation has been enhanced² with an attribute indicating that a construct is deprecated with the intent to remove it in a future version. The motivation for this is that the deprecation marker ended up being used with different purposes, and the theory is that if developers realize that a construct is deprecated with the intent to be removed in a future release, they are more likely to update. As part of this work, there was also additional tools developed, available as part of the Java ecosystem.

Snipes and Ramaswamy [11] proposed a sizing model for managing technical debt related to the deprecation of third-party software components. The model, based on a sigmoid curve, takes into account the lines of code affected, how much (as a percentage) of the API that is deprecated, and the age (in years, with a cutoff proposed to 5 years) of the software component. The authors illustrate the model using Monte-Carlo simulation.

Curtis et al. [5], proposed estimating the principal amount of technical debt based on static code analysis tools and a parameterized formula. The authors illustrate the model by using the tool and formula on 700 applications, comprising 357 MLOC in various languages, from 158 vendors. The article does not state whether only production code was analyzed, or if test code also was included in the metric. A conclusion is that the proposed formula is highly sensitive to the given parameters. The authors illustrate this with three sets of parameter values.

Codabux and Williams [4], studied an organization adopting Agile practices and focused partially on the management of technical

debt. Research questions center on characterizations, consequences on the development process, addressing, and prioritization of technical debt. Test debt in the study relates mostly to missing tests and lacking automation of tests.

Kruchten et al. [7], relate to *Testing Debt* in three different ways: “*Imperfection or suboptimal design and coding of tests*”, “*Misalignment between the tests and the actual code*” and “*Challenges of SaaS contexts*”. Tests, especially when automated, are also code and need to be designed just like production code. When the purpose is unclear, time is spent trying to figure out whether failures originate from the tests or the production code.

3 RESEARCH METHODOLOGY

We take an exploratory approach, utilizing mixed methods and triangulation to elicit experiences from the studied organization. As the studied product has been developed for nine years (of which seven in live operation), a complication is that many developers are no longer available for feedback on conclusions. When possible, we try to confirm and contrast our findings with experienced developers, with a long track record in the product (a handful of which has been with the product since the start). We mined the version control system (Git logs), where all changes to files are stored.

The developers of the studied system took principled decisions related to the support of multiple versions of exposed services:

- New versions are introduced when the existing service cannot accommodate newly requested features, but the core function is the same.
- When introducing a new service version, all old service versions are deprecated. This leads to each service having exactly one version that is “current.”
- When a version is deprecated, it remains in the product, and existing test cases continue to use it. New test cases are written for the new version, but old ones are not converted, even if they use the old, deprecated version of the service.
- Removing a service version is done as a planned task, based on extensive communication with impacted stakeholders and requires updates to all systems (including test cases that might use it for other purposes than explicitly testing the service version itself).

3.1 Research Questions

Based on the above principles, we formulate the following four research questions:

- **RQ1** How has the decision to avoid converting the existing test base when deprecating a service version contributed to the spread of Technical Debt?
- **RQ2** If there is a contribution to Technical Debt in the test code base, are there any differences between the Java-based files (where deprecation is visible in the IDE) and the non-Java-based text files (where no such feedback is visible)?
- **RQ3** How has the growth in deprecated service version usage contributed to the spread of Technical Debt?
- **RQ4** What is the likely cause of the spreading of deprecation debt?

¹<https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/util/Date.html>

²<https://openjdk.java.net/jeps/277>

3.2 Case Description

The system under study forms part of a FinTech global product that enables access to financial services via mobile phones and the Internet. It is typically installed in a high-availability configuration, with geographical redundancy, to meet service uptime requirements. Starting in 2013, the system today serves tens of millions of users in around 15 installations around the world, each integrating with several hundred different other systems. The system contains several transaction-intensive applications, with incoming and outgoing interfaces and persistence layers in the form of different databases. As it is a financial application, security plays a dominant role.

Figure 1: Schematic view of the studied system, exposing services in different versions. External client 2 uses service A in version v2.0 (request Req_{v2} and response $Resp_{v2}$), whereas client 1 has not yet upgraded, and is still using v1.0.

Figure 1 illustrates the system and the role of different services and how each service exposes different versions of requests and responses. Clients external to the system may choose to use any of the exposed service versions to invoke a particular service.

The studied system was developed with test automation in mind, using different test strategies, such as unit testing, integration testing using public APIs, and system testing by a different organization than the development teams. The development has been conducted in an Agile manner, with frequent releases, and incorporating core Agile practices, such as continuous integration and as short feedback loops as is practical. Continuous Integration tools such as Jenkins, giving fast feedback to developers on the quality of their product, has been used during the whole development period.

The system exposes a set of HTTP-based services to the surrounding systems, using XML for marshalling and unmarshalling. As the number of clients to the system is essentially unbounded (or at least unknown at design time), backward compatibility on the protocol level has been an important factor in the design of the system. For this purpose, protocol schemas were developed using XML Schema (XSD), and the system developers assumed that at least some clients, possibly also security firewalls, would enforce strict validation of the request and response messages against the corresponding schema. Due to the compatibility principle, the number of versions of the same service has grown over the years.

Another principle has been that when developing a new version of a service, all older versions are marked as deprecated, using the regular Java deprecation mechanism (annotation and Javadoc). This information was included in the product API documentation, which was generated from Javadoc.

As the system has been developed in Java, deprecation warnings appear integrated into the IDE, such as Eclipse and IntelliJ IDEA. However, integration test cases, interacting with the services via officially supported interfaces, such as XML requests over HTTP, has been developed in a text-based language, custom written for the application. This language is also specified in XML and is executed via a custom runner.

The use of XML text is similar to text-based BDD³ languages, such as Gherkin, but has proved hard to sustain for the 9000 unique test cases. Current developers are unwilling to modify existing test cases, and express frustration at the lack of IDE support for working with plain-text-based languages. This is in contrast to the comparatively well-developed search and refactoring support in IDEs such as Eclipse or IntelliJ IDEA. As the XML language lacks the deprecation mechanism that is available in plain Java, there is no warning when using a deprecated version of a service. Thus, the developer writing or maintaining test cases lack feedback on whether or not this should be changed.

Figure 2 shows the evolution of the number of files of production code (*prod*) and tests over the studied period. The combination of *int.tc* and *int.setup* files contains the integration test files, with *int.setup* referring to the code used for setup, tear-down, and utility functions used in integration test cases, whereas *int.tc* refers to the actual code of the integration test cases. *Unit* refers to the Java-based test code, typically in the form of unit tests.

Figure 2: Production and test files per quarter

The Java-based files, both production and unit tests, show steady growth over the studied quarters. Employing standard linear regression, we see that production files grow with 330 files/quarter, p-value less than 2×10^{-16} , and adjusted R^2 of 0.97. Likewise, the unit tests grow with 128 files/quarter, p-value less than 2×10^{-16} , and adjusted R^2 of 0.99.

For the integration test cases (*int.tc*), the picture is somewhat different. The period up to and including Q4 2015 exhibit a growth

³<https://dannorth.net/introducing-bdd/>

Table 1: Average lines per file across quarters

Type	$\bar{x}$	σ	$\hat{x}$	first	last	min	max
prod	73.4	3.6	73.8	74.2	88.4	66.8	88.4
unittest	172.3	24.4	181.8	139.6	234.5	133.1	234.5
int.tc	155.0	10.3	154.8	187.2	148.5	135.4	187.1
int.setup	83.4	23.1	82.8	117.8	61.5	61.5	135.6

Table 2: Average number of authors each quarter, per year

Year	$\bar{x}$	σ	$\hat{x}$
2011	26.00	7.9	28.0
2012	36.25	2.1	36.0
2013	42.25	6.2	40.5
2014	50.75	1.3	51.0
2015	58.00	3.7	58.5
2016	86.00	5.0	86.5
2017	25.00	10.7	20.0
2018	41.50	15.2	47.5
2019	46.75	3.9	46.5

rate of 300 files each quarter, with a p-value of less than 8×10^{-13} and an adjusted R^2 of 0.97. Then, the ratio increases, between Q4 2015 and Q4 2016, the growth rate is 801 test-case files, with a p-value of less than 0.010 and adjusted R^2 of 0.89. As of Q1 2017, the growth rate has flattened, and is now 84 files per quarter, with a p-value of 2×10^{-5} and adjusted R^2 of 0.80. The last quarter of 2019 shows a decrease in test case files, which reflects the fact that one developer removed, in his opinion, unnecessary integration tests, by converting some to unit tests and removing others altogether, in a refactoring operation. For the integration test setup files and shared functions (*int.setup*), the picture is slightly different. Up until and including Q2 2016, the setup files were growing with 175 files/quarter, p-value less than 3×10^{-16} and adjusted R^2 of 0.98. As of Q2 2016, the growth stagnates to 41 files/quarter, p-value less than 4×10^{-8} and adjusted R^2 of 0.90.

When discussing efforts in terms of counts of files, it is important to note that each file can grow as well. Table 1 contains data regarding the average lines per file type, and its variation across the quarters. For all four file types, the mean ($\bar{x}$) and median ($\hat{x}$) are quite similar, and the standard deviation is also quite small. The *first* column refers to Q1 2012, and the *last* to Q4 2019 (before servicing the debt). Relative to their size, the setup files have the largest standard deviation. Both integration test cases and setup files have a downward trend in file size across the quarters, while the Java-based files (production code and unit tests) are slightly growing, unit tests more so than production code.

The number of authors per year has varied somewhat, as illustrated in table 2, where the number of unique authors, as identified by the Git *Author*: tag, each quarter is summarized (4 quarters per year). After steady growth until 2016, the last year explosively, there was a contraction during 2017, followed by slower growth until the end of the study period.

4 RESULTS

4.1 The origin of deprecation debt

When faced with changed or additional requirements related to an existing service, developers face a dilemma:

Update the existing service, to accommodate new requests or response parameters, or change existing parameters. An equivalent solution is to delete the old service, and introduce a new one with the same name. No Technical Debt would be associated with this option.

Keep the old service version and introduce a new version of the same service, where the required changes are made. The old service version is deprecated, and is associated with a Technical Debt principal, as it has to be maintained alongside the new version, which solves a similar business purpose (being the same service). The consequences of keeping the deprecated version are not only visible at code level, but might be even more severe at the testing level since the integration tests, as will be illustrated later, can keep using the old versions of the service. The propagation effects of deprecation to the test cases are one example of a code TD-item that appears as a TD-item in the testware [1].

Either of the two options can be exercised, and for the studied system, the favored solution was **Update**, unless this was prevented by protocol conventions. Reasons for being unable to choose **Update** were such that would cause existing clients to break, for instance:

- Changing the XML Schema type of an existing parameter (e.g. `xsd:integer` to `xsd:string`), in either the request or the response.
- Adding a new mandatory request parameter P , as this would break old clients who would not send P in their requests towards the system.
- Making an existing optional request parameter P mandatory, as existing clients would be unaware that P was required.
- Adding an optional or mandatory parameter in a response, as this would break those clients who would perform strict schema validation on received responses.

Some of the reasons for choosing option **Update** were:

- If the existing service had not yet been part of any product release, so external clients would not have had the opportunity to interact with it, or learn about the API.
- Adding an optional parameter P to a request, as existing clients could refrain from sending P .
- Refraining from returning an optional parameter P in a response, as existing clients would already have had to deal with the absence of P when interpreting the response.

Initially, the system had no monitoring of which version of a service is currently used by which customer. This monitoring was introduced in 2017. During late 2019, there were efforts made to clean up the unused, deprecated service versions. Based on usage data, 121 deprecated service versions were identified and removed. At the end of the studied period (before cleanup), the system comprised 632 unique services, which exposed 891 different versions of services. In other words, almost three out of ten (29%) of the service versions were deprecated at the end of the studied period.

Kruchten et al. [7] refer to debt principal as proportional to the effort that a development team would expend to eliminate it. The interest incurred by a technical debt item is the additional effort needed to eliminate the debt if the item is left in the system. This view is also shared by others [2]. For our deprecated service versions, the principal would include the effort needed to remove the integration test cases (*int.setup* and *int.tc*), plus the production code (*prod*) and unit tests (*unit*) maintenance between the introduction of the usage statistics and the pay-back in Q4 2019. We classify our deprecated service versions as two different types of Technical Debt Items (TD-items):

- **Type I TD-items:** deprecated versions of services that the usage metrics reported as not in use in any external customer installation.
- **Type II TD-items:** versions of services, that, although deprecated, were reported as in use in some external customer installations by the usage metrics.

While Type I TD-items require a relatively small effort to remove the service versions (relatively low principal), the second Type II TD-items requires more coordination with the customer adaptation teams, for them to adapt the customer installation to the ultimate versions of the service before the removal of the deprecated service versions both from the code and test base.

We treat production code and unit tests similarly, as they are both specified in Java, a statically typed language. Likewise, we count both integration test cases and their setup files together, as they both are specified in untyped XML.

$$\begin{aligned} \text{javacode} &= \text{prod} + \text{unittest} \\ \text{int.test} &= \text{int.tc} + \text{int.setup} \end{aligned}$$

We will use the following set of equations to calculate the usage of a particular service version v at quarter q in files of type t :

$$\text{FILES}(q, t) = \{f : \text{quarter} = q, \text{type}(f) = t\} \quad (1)$$

$$\text{PRESENT}(v, f) = \begin{cases} 1, & \text{if } v \text{ occurs in file } f \\ 0, & \text{otherwise} \end{cases} \quad (2)$$

$$\text{COUNT}(v, q, t) = \sum_{f \in \text{FILES}(q, t)} \text{PRESENT}(v, f) \quad (3)$$

We define $\text{COUNT}(v, q, t)$ as the number of files of type t in which service version v was present in quarter q . We choose the number of files rather than the number of occurrences with the hypothesis that the marginal cost of changing an additional occurrence, once a file has been found, is negligible.

To calculate the distribution of service versions throughout the different kinds of files, we select quarter Q4 2019 (before servicing the deprecation debt), and enumerate all the 891 service versions. For each service version, we calculate the $\text{COUNT}(v, 2019\text{Q4}, t)$, values, where t is either Java (*java*) or XML files (*int.test*). The result is plotted in figure 3, as a histogram with 30 logarithmic bins.

Note the heavily right-skewed distribution, where relatively few service versions are heavily used, while the majority of service versions are used in only a handful of files. Occurrences in Java-based files occur more seldom (closer to the y-axis) than integration tests, though 96 service versions are present in only one integration test file. This is also visible in table 3, where the Java-based code

Figure 3: Occurrences of services in Java or integration test code

Table 3: Occurrences in files

Type	N	$\bar{x}$	σ	$\hat{x}$	$Q_{90\%}$	$Q_{95\%}$	max
javacode	891	5.3	7.9	3	10	15	165
int.test	867	31.0	110.0	8	57.4	106	1529

have a considerably smaller mean ($\bar{x}$) and median ($\hat{x}$) than the integration tests. Table 3 also displays the number (N) of different service versions (noting that 24 service versions that are used in Java code are not used in integration tests), standard deviation (σ), 90th and 95th percentiles, and the max value.

Production code and Java-based tests, such as unit tests, typically use a broader spectrum of interfaces, such as internal domain-model interfaces, whereas the primary purpose of the integration tests is to exercise and validate the external application interface (that is, the exposed services, in their different versions). The most heavily used service version appears in 1529 integration test files (including setup files), out of around 12300 files.

4.2 Deprecated services

All analyzed services start as “non-deprecated,” in version 1.0, but some will evolve together with the product to later versions.

Result: Time series of usage data for each deprecated service version following deprecation

```

end_of_study ← Q4_2019;
for v ← all_deprecated_service_versions do
  when ← find_deprecation_timestamp(v);
  following_quarter ← next_quarter(when);
  for q ← following_quarter to end_of_study do
    count[v, q, Java] ← COUNT(v, q, Java);
    count[v, q, XML] ← COUNT(v, q, XML);
  end
end
return count

```

Algorithm 1: Calculating usage of deprecated service versions

Algorithm 1, illustrates the process to collect the usage data for every deprecated service version, starting at the quarter following the deprecation event as recorded in the Git version control log. Thus, the initial data point for a service version closely reflects the state at the time when it was deprecated. In the ideal world, each service version should show a flat or decreasing line, as its usage decreases as the product evolves. It is important to note that also deprecated service versions, which are still in use and delivered with the product, should have some usage in tests (the target is not zero files). However, typically this number should be relatively low (in the single digits), as the officially supported version should be used instead.

Figure 4 depicts the situation for the six most frequently used deprecated service versions⁴, and figure 5 for the following six. All of these occur most frequently in integration tests. A few service versions are fairly flat, a few show moderate growth, and a few show extreme growth up until Q2 2016. Following the end of 2016, most of the usage is flat, and there are few changes in usage.

Figure 4: Top 6 used deprecated service versions per quarter

Figure 5: Top 7-12 used deprecated service versions per quarter

⁴The names and nature of the services are obfuscated due to confidentiality and security reasons.

The situation in the six deprecated services most frequently used in Java files is depicted in figure 6. While there is a slight increase for some services, in general the situation is much more stable than for the integration tests, and the number of affected files are also much lower.

Figure 6: Top 6 used deprecated service versions in Java files

We note that some services (e.g., *Epsilon*, *Eta*, *Iota*) appear both among the top 12 integration tests and in the top six Java-based files. These are “core services,” central to the application at hand. Furthermore, we note that one service, *Iota*, in different versions, is both the sixth and the eighth most used in integration tests.

Most of the quarters, there is no change in the usage of the deprecated service versions. Out of the 7849 data points (one deprecated service version in one quarter is a data point), growth is flat in 6584, i.e. given a random service in a particular version, and a random quarter following the deprecation of that service version, the chance is nearly 84% that there has been no change in usage. The odds for decreasing usage is 5.1% (413 cases), and increasing is 10.9% (852 cases).

Table 4: Most growing deprecated service versions, per quarter. *Count* is the $COUNT(v, q, t)$ metric, and *Growth* is the change in *Count* since the prior quarter. *Dupl* is the number or duplicates among the *Growth*, and % *Dupl* is the relative number of duplicates. Type-I TD-items in bold.

Service	Quarter	Count	Growth	Dupl	% Dupl
*Alpha v1_0	Q1 2016	949	338	66	19.5
Beta v1_0	Q2 2016	1058	243	77	31.6
*Alpha v1_0	Q2 2016	1150	201	74	36.8
Beta v1_0	Q1 2016	815	185	65	35.1
Theta v1_1	Q2 2016	1019	175	65	37.1
Theta v1_1	Q1 2016	844	150	48	32.0
Delta v1_0	Q1 2016	666	100	0	0.0
Beta v1_0	Q4 2015	630	86	53	61.6
Theta v1_1	Q4 2015	694	80	42	52.5
*Zeta v1_0	Q1 2016	505	72	0	0.0

Looking at the quarters with the most growth in absolute terms, we see the same three quarters as in figure 4. The top 10 growth

operations and quarters are illustrated in table 4. In the table, the service versions *Alpha v1.0* and *Zeta v1.0* is marked in bold, as being Type I TD-items (deprecated and not being used in live operation). The column *Count* contains the $COUNT(v, q, int.test)$ metric for the specified service version and quarter, while the *Growth* column, contains the change since the previous quarter. The *Dupl* column is the absolute number of duplicated files of the *Growth*, and the % *Dupl* is the relative percentage of duplicates, compared to the *Growth*. We see that a small number of service versions cause most of the growth in debt. Furthermore, this debt is concentrated to three quarters, spanning Q4 2015 to Q2 2016. Inspecting the % *Dupl* column, we see that for some service versions (*Beta v1.0*, *Theta v1.0*), over half of the added files are duplicates of each other. This indicates that the developers were unaware of a feature that was added to the test execution runner in Q4 2015, where a “shared function repository” could be defined. All of the duplicated files except one are setup files or “utilities” for the test cases.

Figure 7: Percentage of identical integration test files and LOC

In figure 7, we plot the relative number of file duplicates across each quarter, and the lines of code in these files. We see that as of 2014, around 15% of the files are duplicated, though these only make up of around 5% of the total code base.

In table 5, we instead look at the service versions and quarters where the most deprecation debt has been repaid (in the form of files that no longer are used). We see that five out of the top 10 service versions were repaid during Q4 2019. Further analysis of the version history logs revealed that this is related to the prior mentioned refactoring illustrated in figure 2, where one developer removed many integration tests, converting some of them to unit tests.

In order to conclude as to how the duplicated files shown in table 4 were introduced, we analyzed the version control logs. The analysis indicated that the files, to a large extent, originated from experienced developers. All in all, 102 commits were affected, authored by 25 different developers, and the top 5 contributors caused 73.5% of the duplicates. These top 5 contributors all had several years of experience in the product at the time of introducing the duplication.

Table 5: Least growing deprecated services, per quarter. *Count* is the $COUNT(v, q, t)$ metric, and *Growth* is the change in *Count* since the prior quarter. Type-I TD-items in bold.

Service	Quarter	Count	Growth
Nu v1_0	Q4 2019	200	-28
Psi v2_0	Q4 2019	44	-28
Chi v1_0	Q4 2019	20	-28
*Zeta v1_0	Q1 2017	537	-29
Epsilon v1_0	Q1 2017	851	-31
Iota v1_0	Q1 2016	601	-40
Delta v1_0	Q1 2017	730	-51
Zeta v1_7	Q3 2018	10	-60
Phi v1_0	Q4 2019	21	-105
*Zeta v1_4	Q4 2019	36	-148

Figure 8: Schematic illustration of Principal, Interest, First and Last metric for integration tests of service Alpha v1.0

Table 6: Affected files for all deprecated services

Type	<i>N</i>	$\sum First$	$\sum Last$	$\sum Add$	$\sum Del$
javafiles	285	1620	1878	292	-34
int.test	282	10619	13526	3819	-912
Sum		12239	15404	4111	-946

Figure 8 illustrates the concept of principal, which, for Type-I TD-items, are proportional to the usage in the code base, and interest, which is proportional to the increased usage. The *First* metric is the usage at the quarter following deprecation (different quarter for each service version), and the *Last* metric is the usage at the end of the study, before servicing the debt. Usage before deprecation is not counted as principal by this metric.

In table 6, the $\sum First$ column is the sum of all the *First* values for all *N* deprecated service versions and the $\sum Last$ column is the sum of all the *Last* values. The $\sum Add$ column totals the net additions, and the $\sum Del$ column the net removals. In case one file contains multiple instances of the same deprecated service, it is only counted once, but in case it contains several deprecated services, it is counted once per such service. This uses the hypothesis that changing several instances of the same pattern in a single file incurs

a relatively minor additional cost, compared to finding the file and changing in one place.

The cost of removing a service version V_{depr} , which has a non-deprecated version V_{actual} can be broken down into the following components:

$$C_{actual}(V_{depr}) = C_{internal}(V_{depr}) + C_{external}(V_{depr}) \quad (4)$$

$$C_{internal}(V_{depr}) = C_{removal}(V_{depr}) + C_{update}(V_{depr}, V_{actual}) \quad (5)$$

where

- $C_{actual}(V)$ is the actual cost of removing V .
- $C_{external}(V)$ is the cost of removing V by integrators and other clients (outside the development organization).
- $C_{internal}(V)$ is the cost of removing V that is internal to the development organization (code base, including tests).
- $C_{removal}(V)$ is the cost of removing only V and its associated test cases.
- $C_{update}(V_1, V_2)$ is the cost of updating all the remaining tests from V_1 to V_2 .

Using the Technical Debt metaphor, the $C_{actual}(V)$ at the time of deprecation can be considered as the principal, as the alternative would be to simply remove V , replacing it with the updated version. We can estimate $C_{update}(V_1, V_2)$ as the additional usage of V_1 , not related to the normal tests of this version. In integration tests, it is common that certain operations are used for setting up the data in order to test other services, such as when a login service is used in several integration tests across the test base. This is what we call *excessive usage* of a service version, as opposed to the *expected usage* of a service version which is its source code, its unit tests and integration tests of the service version itself.

For Type-I TD-items, where the $C_{external}$ is zero, we find that the $TD_{principal}$ is proportional to the usage at the time of deprecation, that is, $\sum First$. The accrued interest is proportional to the growth in usage, that is, to $\sum Last - \sum First$.

$$TD_{principal_I} \propto \sum First$$

$$TD_{interest_I} \propto \sum Last - \sum First$$

Thus, the relative growth of $TD_{interest_I}$ for Java files has been $\sum Last / \sum First - 1$, that is $(598/505) - 1$, or 18%. For the integration tests, the corresponding number is $(3428/2383) - 1$, or 44%. Thus, the interest rate for integration tests is much higher than for Java-based tests. We also note that there have been some decreases. In total, 268 files have been corrected for the 120 different integration test service versions, but this has not been able to keep up with the growth.

For Type-II TD-items, the situation is somewhat more complicated, as one also has to consider the lack of test cases as a debt principal, what Kruchten et al.[7] refers to as “*Misalignment between tests and code.*” We note that there are three service versions not tested in integration tests (Table 6, N for *int.test*), and one of these was among the Type-I TD items (Table 7).

4.3 Servicing the debt

We removed 121 deprecated service versions in late 2019, see table 7. We note that one of the removed service versions was not present

Table 7: Affected files for unused services (Type-I TD items)

Type	N	$\sum First$	$\sum Last$	$\sum Add$	$\sum Del$
javafiles	121	505	598	105	-8
int.test	120	2383	3428	1313	-268
Sum		2888	4026	1418	-276

in any integration tests (closer inspection revealed that those tests had been migrated to the new version at the time of deprecation, in effect leaving the old version untested for five years).

Based on data from 21 of the operations that were not part of any other tests, we found that on average, a service version occurred in 4.4 Java files and 3.6 integration test files. This can be used to estimate the $EXPECTED_USAGE$, using the values from table 7.

$$EXPECTED_USAGE_{java} = 4.4 * 121 = 532 \text{ files}$$

$$EXPECTED_USAGE_{int.test} = 3.6 * 120 = 432 \text{ files}$$

Based on these values, we find that out of the removed operations, there was little, if any, excessive usage amongst the Java files. However, there had been some growth due to interest (93 additional files being affected). The majority of excessive usage can be attributed to the integration tests, which also contributed the most of the additional interest.

The removal of the 121 service versions was carried out as a low-priority task during several months by three developers, with between 4 to 25 years of experience in the industry. The time spent (based on Git logs and estimations by the developers, as this time was not separately time reported) was 160 hours. Part of this time was due to back-porting the changes to an older, but still alive, branch, which proved to be almost as costly as doing the change in the original branch. Given this, it is reasonable to estimate that, had there been no additional growth in technical debt (such as the one during 2016 illustrated in figure 4), the removal would have taken between 0% (in case all file changes could have been automated, e.g. via scripts), and 29% ($1 - 2888/4026$) less time, i.e. 114 hours rather than 160. In practice, the truth is somewhere in between, as some changes could be highly automated, and some required more manual actions. These kinds of tasks are also highly dependent on the skill of the developer doing the change, in particular as the plain-text-based XML language lacks IDE support for refactoring operations such as renaming methods. All three developers were well versed in the Git version control system, scripting tools, and regular expressions, and used them, together with the test base and the Continuous Integration environment to verify system behavior before and after each change.

4.4 Commit counts

It could be argued, from a theoretical standpoint, that a file that is untouched and never read has no technical debt, even if it contains TD-items (in our case deprecated services). In figure 9 we visualize the activity on the top six deprecated service versions, starting from the quarter following deprecation, plotting how many commits per quarter affect files where these services are used.

We see the many commits during late 2015 and 2016, where most of the commits were introducing the deprecated services into the

Figure 9: Count of commits affecting files using the top six deprecated service versions across each quarter

test base. Then, a low period during 2017 and 2018, followed by high activity during Q2 2019, where many commits were touching files related to the top deprecated services. During this quarter, two new requirements were developed, adding configurability where previously more rigid business models had been used. In total, 78% of the commits for the ten most used deprecated services during Q2 2019 were related to these two requirements. Thus, figure 9 illustrates that for almost two years, there was not much movement in the files related to the top deprecated services. Then, during early 2019, two requirements were to be implemented, which caused much effort in maintaining these files (i.e., interest that could have been avoided if these deprecated versions would have been removed). This illustrates the highly non-linear nature of technical debt management, making it even more important to visualize it so that efforts to mitigate it can be estimated appropriately.

4.5 Discussion

Regarding **RQ 1**, whether the rule to avoid updating existing tests contributed to the spread of Technical Debt, we experienced an uneven distribution of deprecation debt among the deprecated services. Since a small number of highly used services caused a significant increase in debt in integration tests, we hypothesize that highly used services should be treated differently than others.

Regarding **RQ 2**, whether the XML-based integration tests and the Java-based files show any difference in technical debt, we experienced very little debt for unit test cases since the number of affected files remains small. The integration tests show large growth in debt, originating from a small amount of highly used services, as shown in figure 4 and figure 5. After 2016, we experience a slower growth of integration test cases in general, and the product appears to have been tested using other means (such as Java-based tests), with minimal usage of the integration test framework.

Regarding **RQ 3**, how the growth in deprecated service version usage has contributed to Technical Debt, we note that most of the growth in deprecated services usage occurred during a limited time, between late 2015 and mid-2016. Integration tests were much more affected than Java-based files. For the services removed in late 2019, the accrued interest in the integration tests was calculated to 44%,

versus 18% for the Java-based files. We have identified considerable duplication among the integration tests, particularly the setup files. Undue duplication incurs technical debt, even if a particular service is not deprecated, but in this case if a service is deprecated but still used in integration tests might lead to unexpected tests results due to the combination between deprecated and non-deprecated services. Thus, the general higher frequency of copied code in the integration tests could explain why there were more duplicates of deprecated service versions.

On a positive side, we note some repayment of interest for both types of files, though the decrease does not weigh up the increase in debt.

Regarding **RQ 4**, about the likely cause of the spreading Technical Debt, we can identify at least three causes of the spreading of deprecation debt:

- Backward compatibility causes technical debt unless the protocol can accommodate the needed changes without breaking older clients. Depending on the strictness of validation in clients or intermediate systems, different amount of debt is incurred. In the studied system, the rigid protocol rules, and strict security requirements (schema validation in firewalls) contributed to the large number of deprecated service versions.
- By examining the integration tests, we could see a pattern of copied setup files. Even though the integration test engine had the possibility of using a “shared function repository” since Q4 2015, this feature was not widely used during early 2016. Since 2014, the level of duplication in the integration test base has been around 15% on a file-by-file basis, comprising approximately 5% of the codebase, as shown in figure 7.
- Regarding the period with the most growth, late 2015 to mid-2016, we could not see direct causes due to the growth in personnel during this time, though indirect causes, such as stress on experienced developers, can not be ruled out. Feedback from three developers with knowledge of the situation unanimously report this time as highly challenging (meeting a customer deadline). This is consistent with what is reported by Kruchten et al. [7]: “The most likely cause [of Technical Debt] we have observed is schedule pressure”.

To summarize, we state a hypothesis that if an appropriate visualization had been in place for the integration tests, it is likely that both the duplicated files and the added deprecation debt could have been avoided during the stressful times of 2015-6. This might have saved up to 29% of the removal effort in late 2019. It is important to note that the distribution of services is highly right-skewed, meaning that measures of central tendency such as the mean, median, and standard deviation values are not very helpful. Instead, “top-N”, or “most-growing” lists should be used to keep track of the code.

5 THREATS TO VALIDITY

We discuss the threats to validity from four different angles: *construct validity*, *internal validity*, *external validity* and *reliability*, following guidelines outlined in [12].

Construct validity deals with whether the studied measures reflect what the researcher has in mind, and what is stated in the research questions.

In this study, we use version control data (source code and revision history information) to draw conclusions. A threat to validity to this approach is that “you only see what was built,” which is a form of survivor bias — i.e., you might see what was built, but not how or why it was built that way. To increase construct validity, we presented our conclusions to five of the remaining developers, validating our assumptions about why certain solutions were used.

When using version control logs as a data source, an important aspect is to consider the branching pattern and whether or not the studied organization commonly used rebasing commits. The Git version control system allows authors to “squash” commits, which may have been performed by different authors, at different times, into one new commit, discarding the constituent commits. For the studied system, this was not an approved practice, as the organization valued to see each commit, as it was written and pushed to the central repository. Most of the development took place in a single “master” branch for the duration of the study. Features were developed in other branches and later introduced into the master branch, typically via the Git rebase function, which keeps a linear history by rewriting commits, preserving author information and commit dates.

Internal validity deals with whether there might be other, non-studied factors that could explain some of the findings. We have gathered quantitative data (Git logs, usage statistics), but also the Git ways of working for the studied system, which allows us to study the phenomena (i.e., the effects of deprecation on TD and its spread). In addition, to validate our conclusions, increasing internal validity, we triangulated our data by providing our conclusions and feedback to five developers of the system, one of which was the author of much duplicates during 2015–2016, and asked about their opinions. This provided valuable insight into the conditions and the time pressure experienced by the development organization.

External validity concerns to what extent it is possible to generalize the findings, and to what extent the findings are of interest to other people outside of the investigated case.

This paper is about experiences from a particular system, with a particular toolset. We have tried to describe characteristics that might enable others to judge whether the findings are applicable for other systems, but we cannot claim generalizability across all possible systems or organizations.

Reliability concerns whether or not the data and analysis are dependent on the specific researchers. Most of the data in this report are collected from quantitative sources, such as Git logs, and processed and visualized using standard statistical tools. As such, there is little room for bias in the processing of the collected data.

Interpretation of the processed data runs the risk of introducing reliability threats. We strove to reduce reliability threats by frequent interactions with the studied organization, especially with the developers that had been part of the product development team during the whole studied period, and elicited feedback from five of them.

6 CONCLUSIONS

As illustrated in this paper, to combat deprecation debt, it helps to keep it visible. Many IDEs today show deprecated classes in a different font (e.g., strikethrough), and this could be one reason

why the unit tests do not show the same growth as the integration tests.

Another finding is the uneven distribution of the contribution to the debt. For the majority of services, the decision not to update deprecated usages in test cases did not spread any technical debt. However, the decision not to update some highly used “core services” caused up to 29% of increased effort in converting the test base at service removal. The addition of this debt mostly occurred during three quarters, between Q4 2015 and Q2 2016, a time that was identified as particularly stressful, which aligns with the findings of Kruchten et al. [7].

After the downsizing of the product, the integration tests were mostly untouched, as the new developers valued other test principles, such as Java-based testing (relegating much of the prior testing to unit tests). This suggests that as the test base grows, so does the importance of the IDE support (refactorings, static code analysis, duplication detection).

The usage statistics turned out to be a valuable tool to identify unused service versions (Type-I TD-items), supporting the removal of these.

ACKNOWLEDGMENTS

This research was supported by the KKS PLEng 2.0 grant at Blekinge University of Technology, and Ericsson AB, through the SHADE KKS Hög project with ref: 20170176, and through the KKS SERT Research Profile with ref. 2018010 project both at Blekinge Institute of Technology, SERL Sweden.

REFERENCES

- [1] Emil Alégroth and Javier Gonzalez-Huerta. 2017. Towards a Mapping of Software Technical Debt onto Testware. In *43rd Euromicro Conference on Software Engineering and Advanced Applications*. Vienna, Austria, 404–411.
- [2] Paris Avgeriou, Philippe Kruchten, Ipek Ozkaya, and Carolyn Seaman. 2016. Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162). *Dagstuhl Reports* 6, 4 (2016), 110–138. <https://doi.org/10.4230/DagRep.6.4.110>
- [3] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. 2016. Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture. *IEEE Software* 33, 3 (may 2016), 42–52. <https://doi.org/10.1109/MS.2016.64>
- [4] Zadia Codabux and Byron Williams. 2013. Managing Technical Debt: An Industrial Case Study. In *Proceedings of the 4th International Workshop on Managing Technical Debt*. 8–15.
- [5] Bill Curtis, Jay Sappidi, and Alexandra Szynekarski. 2012. Estimating the principal of an application’s technical debt. *IEEE Software* 29, 6 (2012), 34–42.
- [6] Martin Fowler. 2014. Microservices. <https://martinfowler.com/articles/microservices.html>
- [7] Philippe Kruchten, Robert L. Nord, and Ipek Ozkaya. 2019. *Managing Technical Debt: Reducing Friction in Software Development*. Addison-Wesley Professional.
- [8] Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2018. Do developers update their library dependencies?: An empirical study on the impact of security advisories on library migration. *Empirical Software Engineering* 23, 1 (2018), 384–417. <https://doi.org/10.1007/s10664-017-9521-5> arXiv:1709.04621
- [9] Anand Ashok Sawant, Romain Robbes, and Alberto Bacchelli. 2017. On the reaction to deprecation of 25,357 clients of 4+1 popular Java APIs. *Proceedings - 2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016* (2017), 400–410. <https://doi.org/10.1109/ICSME.2016.64>
- [10] Anand Ashok Sawant, Romain Robbes, and Alberto Bacchelli. 2019. To react, or not to react: Patterns of reaction to API deprecation. *Empirical Software Engineering* (2019), 3824–3870. <https://doi.org/10.1007/s10664-019-09713-w>
- [11] Will Snipes and Srini Ramaswamy. 2018. A proposed sizing model for managing 3rd party code technical debt. *Proceedings - International Conference on Software Engineering* (2018), 72–75. <https://doi.org/10.1145/3194164.3194179>
- [12] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*. Springer, Berlin.
- [13] Olaf Zimmermann. 2017. Microservices tenets: Agile approach to service development and deployment. *Computer Science - Research and Development* 32, 3-4 (2017), 301–310. <https://doi.org/10.1007/s00450-016-0337-0>