Test-Driving FinTech Product Development: An Experience Report [⋆]

Anders Sundelin[1,2], Javier Gonzalez-Huerta[1], Krzysztof Wnuk[1]

[1] Blekinge Institute of Technology, Karlskrona, Sweden
[2] Ericsson AB, Sweden
anders.sundelin@ericsson.com, javier.gonzalez.huerta@bth.se
krzysztof.wnuk@bth.se

Abstract. In this paper, we present experiences from eight years of developing a financial transaction engine, using what can be described as an integration-test-centric software development process. We discuss the product and the relation between three different categories of its software and how the relative weight of these artifacts has varied over the years. In addition to the presentation, some challenges and future research directions are discussed.

Keywords: Test-driven development · Software Craftsmanship · Testing Architecture.

1 Introduction

Software and software products are the critical elements of the Financial Technology (FinTech) [9] revolution that reshape the way how individuals and financial institutions save, borrow, make payments, and manage risk [7]. Software is enabling societal change in our relationship with money, especially in developing economies where alternative financial services are more customer focused and allow more people to have access to finance without the need of a bank [3]. Ericsson has seen the opportunity that FinTech offers as early as 2010 and decided to create a financial product for developing economies that provides access to payment services to users without credit card or bank account. Ericsson developed the product considering market demands and requirements such as *security*, *auditing*, *correctness*, *performance*, *availability*, *flexibility*, *fast time to market* and *development efficiency.*

In this paper, we discuss experiences on how Ericsson tackled the problem of crafting a software product for the financial sector not only migrating to a modern programming language and with a Service Oriented Architecture, but also using modern ways of developing the software such as *test-driven development*, *integration tests*, *continuous integration*, *clean code*, *learning by doing*, *mandatory solution review* and *simple communication.* Several studies analyze the effects that TDD has on code quality and defect rate [12, 10], though few

[⋆] Research supported by Ericsson AB and PLEng School.

studies analyze the long-term effects that TDD might have in the project, regarding the number of defects and the size of the test base as compared to the code base. Moreover, there is a lack of longitudinal experience reports of developing Fintech products for global markets.

2 Background and Related Work

Although it has earlier roots in the Smalltalk community, the term Test-Driven Development was popularized in the late 1990s and described as part of the Extreme Programming process, [1, 2]. Several scientific studies have analyzed the effects of TDD e.g., [12, 10, 5].

In an experiment described in [6], the authors compare TDD with the alternative iterative test-last (ITLD) process, concluding that the claimed benefits of TDD arise not from its test-first approach, but from the fine-grained, steady steps, with fast feedback that improve focus and flow. Our paper supports this conclusion, adding aspects of product development over eight years.

One of the first public tools to support automated acceptance tests was the Framework for Integrated Tests (Fit) [3]. The acceptance-test-driven development (ATDD) process [13] was studied in [11, 8].

Most of the studies of TDD have focused on shorter timescales, from some weeks, up to a few years worth of software development. This paper adds experiences from eight years of building a product from scratch using rigorous testing methodologies, and the effects that this has had on the code base.

3 Case Description and Analysis Method

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. The system is a transaction-intensive application, with incoming and outgoing interfaces, a database, and scheduled tasks such as the sending of notifications. As it is a financial application, security has played a central role in its development.

The studied system consists of the financial core of the application, containing the core business logic, such as the financial transaction management. The core exposes its services via a set of *requests*, similar in spirit to the system calls of the Unix kernel. There are other components in the product, such as user interfaces, both graphical and textual, but these are not studied in this report. All other components use the services of the core in order to perform their tasks. The system is built in Java, using EJB 3[4] principles and is deployed using a custom, light-weight EJB container, also exempt from this study.

[3] http://fit.c2.com/

[4] http://download.oracle.com/otndocs/jcp/ejb-3.1-pfd-oth-JSpec

One of the guiding principles when developing this application was intensive test automation. Testing should take place in different layers, with the bulk of the tests in the lower layers (unit tests), and progressively fewer tests the higher the abstraction level. This follows the principles outlined in the testing pyramid [4].

3.1 Studied artifacts

We analyzed the production code and test artifacts developed by the *development team* during feature development. Ericsson also has dedicated *testing teams*, focusing on testing the complete system, including all required special hardware, such as hardware cryptography modules and application firewalls, but these activities are not studied in this report.

The studied software is classified according to the following three categories:

- *Production code* - which is deployed at the customer site, and perform some useful action in live deployed systems.
- *Unit test code* - which is developed alongside the production code, typically by the same developer.
- *Integration test code* - using the externally visible interfaces of the system, typically describing the use cases that the application shall provide. Integration tests are developed by the same team and at the same time as the production code, though typically by different developers.

In addition to these three categories, the system also contains small amounts of other software and configurations files, such as installation support software, test enablers that aid integration testing, and system test code for testing the entire software system.

The number of authors of the studied software has varied over time, due to business and organizational changes. For the production and unit test code, the number of authors has been between 9 and 74, with a median of 35. The integration tests have a slightly wider span, between 8 and 79 authors, also with a median of 35. The majority of developers (median: 29.5) has developed both production code and integration tests.

3.2 Tools

We use the common open-source tools cloc[5] and Git[6] in order to calculate statistics for the above-mentioned categories. We collected statistics for every feature-enhancing release made of the product, plus three initial ”pre-releases”, denoted P01, P02, P03, before the first commercial release, denoted R01, in mid-2012. On average, 81 days have passed between releases, with a median of 62.5 days and an IQR of 55.5 days. There have also been other releases made of the software, both for error corrections, and candidates meant for internal testing.

[5] https://github.com/AlDanial/cloc

[6] https://git-scm.com/

Table 1. Mean and median number of lines per file.

File type	Mean lines/file	Median lines/file
Production code (Java)	90	55
Unit test code (Java)	228	163
Integration test (XML)	133	97

4 Results and Discussion

4.1 Size of the code base

Table 1 lists the mean and median lines per file and Figure 1 illustrates the number of lines of code per category, for each studied release. The last studied release, R32, consists of about 5000 files of production code, and about half as many unit test files. The number of integration test files is more than double the number of production files, about 12000.

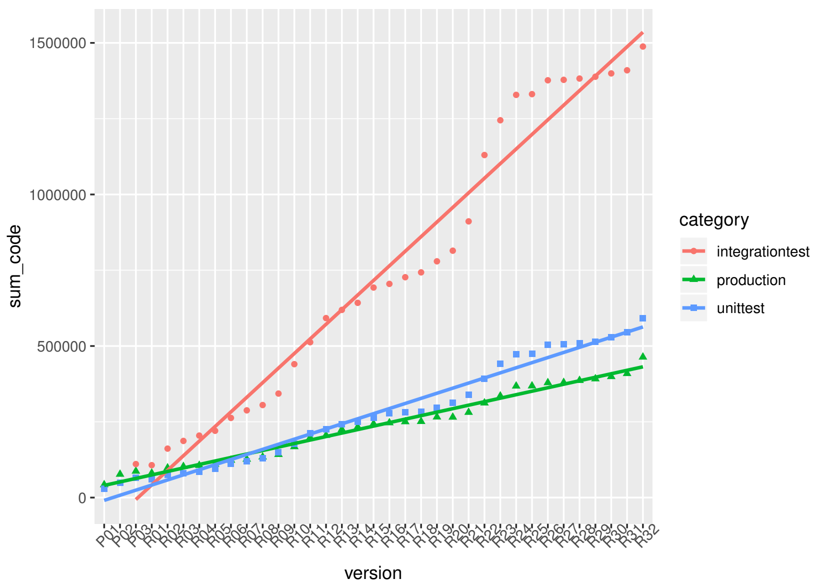

Fig. 1. Lines of code for each category, per release, P01, P02 and P03 are initial prereleases and R01 is the first commercial release.

Figure 1 shows a linear growth, with integration test code (red) dominating over production code (green). While the production code has grown from about 42 kLOC in 583 files in the first preliminary release, to about 460 kLOC in

5166 files in the last studied release, the number of lines of integration tests has grown to 1488 kLOC, in 11211 files. The first two preliminary releases contained no integration tests but were tested using other tools, later discarded due to lack of productivity. To enable efficient integration testing, the developers chose to develop an XML-based testing tool. The tool was first used in the P03 release, which comprised of 110 kLOC integration test code lines, distributed in 622 files.

Regarding the unit test code (blue), we see that starting from release R11, the number of lines of unit test code exceeds the number of lines of production code. In the latest studied release, there are about 30% more lines of unit test code than of production code. Thus, the unit test code base is also growing faster than the production code, though not as fast as the integration tests. The fact that the typical unit test file is larger than the typical production code file supports the notion that the tests are mostly concrete code, in the typical *Arrange*, *Act*, *Assert* fashion, whereas the production code consists of a higher number of interfaces and abstract classes.

We can conclude that this is a product where test code, both unit tests, and integration tests, make up the bulk of the software. As the growth rate of both the unit tests and the integration tests are higher than for production code, it becomes a necessity to manage this growth, for instance by reducing duplication and increasing modularity and reusability. The importance of this increases as the product ages and grows.

4.2 Defect prevalence

Figure 2 shows the number of corrected defects per release. The upper (red) line is the total number of corrected defects, the middle (green) line is the number of defect corrections that are new to the release, and the lower (blue) line is the ratio between the defects new to the release and the size in kLOC of the production code base.

The defects in this statistic include those found by customers in the field, internal testing organizations during system verification, and those found by the developers after the release of a feature. Defects found by developers during the development of a feature are not included. No goals or penalties related to the number of found defects in the product have been used by the organization, though goals related to the defect response time have been used. Thus, it is unlikely that developers have refrained from issuing defect reports in order to fulfill some target objective.

It is quite evident from the picture that the ratio of defects was higher at the beginning of the product lifecycle when there were few lines of production code, and none or few integration tests. It is also quite evident that some releases contain more corrections than others. There are two reasons for this: Some releases (e.g. R24 and R26) contained many corrections from customer branches, which was then integrated into the main branch. Some releases (e.g. R25) was made very close in time to the prior release, causing it to contain fewer changes. Further analysis of the defect origin is deferred to another paper.

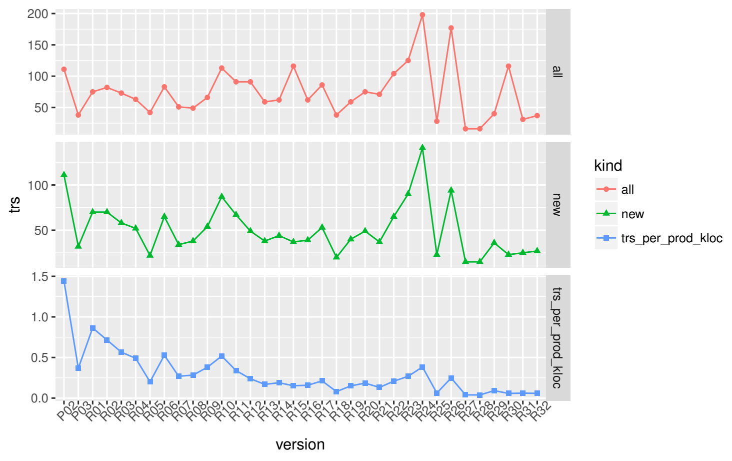

Fig. 2. Number and ratio of corrected defects for different versions.

4.3 Changes to the code base

By using *git diff* to analyze files changed in each version, it is possible to get an overview of the changes in the code base. It should be noted that this is only the ”net change”, as this statistic does not capture files that have changed multiple times between versions. Also, lines are reported regardless of whether it is a code or comment line, and a changed line is counted as both an added line and a removed line.

On average, between versions, production code have added 28372 lines and removed/changed 14535 lines in 884 files, unit tests have added 32891 lines and removed/changed 12934 lines in 565 files, and integration tests have added 75005 lines and removed/changed 39424 in 2530 files. This is another way of illustrating that changes to the integration tests dominate over the production and unit test code.

4.4 Guiding Principle: Test fast, test in layers

While the ATDD and TDD processes were encouraged, explained and exercised during onboarding of new developers, it was still up to each developer to do their tasks in the order they preferred. Thus, it is likely that some developers followed other processes, such as Incremental Test-Last (ITL) [6]. The common ground between these two processes is that tests should be developed as close (organizational and temporal) to the production code as possible, and refactorings should be performed when all tests succeed. This is in contrast to a more traditional ”Design-Implement-Test” approach, where typically the tests are developed once all, or most functionality is implemented.

The organization actively required that tests were developed as the requirements were implemented. No feature was allowed into the product without having the required supporting test base. Also, there were continuous discussions among developers, how a feature should best be verified, and what things that were the most important to verify. As is shown in the defect statistics, figure 2, the integration test base helped limit the number of defects as the product has grown. One of the consequences of this principle is that the amount of test code will grow, in parallel with the growth of the production code. We see in the reported statistics that both the unit tests and the integration tests grow faster than the production code.

Both the number of files and the number of lines of integration tests is much greater than the corresponding metrics for production and unit testing code. In part, this stems from the use of XML as a specification language for the integration tests.

One obvious benefit of having a separate language for integration tests is that it is possible to enforce certain rules, by only implementing wanted features in the language executor. For instance, in the current integration test language, there is no support for conditional branches and only limited support for iterative loops. Another benefit is that developers specifying the integration tests have a clear line between the production code/unit tests and the integration tests. They can safely ignore the Java syntax and features in the Java language.

The most severe disadvantage of the separate integration test language is that the lack of effective module support causes the number of lines of integration tests to grow faster than the production code and unit test code. Another disadvantage is that there is no IDE support, such as code completion, refactoring or debugging support, out of the box. Thus, trivial transformations or reports such as *rename method*, or *find usages* becomes difficult for developers not well versed in file system and text processing tools such as *find*, *grep*, *awk* or *perl*. Due to the lack of code completion, there is also the risk of longer development times, and developers being unaware of similar functions for setting up the scenarios.

5 Implications for Research and practice

As can be devised from the statistics shown in Section 4, the product currently consists of considerably more lines of test code than production code. In order to be able to work efficiently and develop with speed, it is imperative that this test code is kept clean and undergoes a similar refactoring scheme as the production code. A design principle used when developing test cases is that each test should be "self-contained", and "self-specified". Each test case should specify its required state, and asserts should reference this state, and not obscure references to other "magic numbers". The disadvantage of "the self-containment principle" is that naive developers may copy code, instead of extracting sections into methods or modules.

Care has to be taken when refactoring test code. In particular, it is important that the principles of *Arrange*, *Act*, *Assert* continues to hold for each test. Also, each test case should continue asserting everything it asserted before the refactoring, to avoid causing the refactored test case to be more lenient than the original one. The *Arrange* phase, however, should be as lenient as possible, only specifying the minimal state required to make the test succeed. Different initial states are typically referred to as fixtures, and to avoid undue repetition, it is important to keep track of which fixtures that are already available when writing new test cases.

A solution to the test refactoring problem is to introduce known errors in the production code while refactoring the test code, checking that both the old and new test cases catch the introduced errors. Once a satisfactory refactoring has been completed, the new, refactored test case is committed, and the introduced errors reverted, leaving the original (non-faulty) production code. This field should be studied more thoroughly, e.g., whether the refactoring validation process can be automated, or if some static rules could be devised.

References

1. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999)
2. Beck, K.: Test Driven Development: By Example. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)
3. Blakstad, S., Allen, R.: FinTech Revolution: Universal Inclusion in the New Financial Ecosystem. Springer International Publishing (2018), https://books.google.se/books?id=0_VeDwAAQBAJ
4. Cohn, M.: Succeeding with Agile: Software Development Using Scrum. Addison-Wesley Professional, 1st edn. (2009)
5. Erdogmus, H., Morisio, M., Torchiano, M.: On the effectiveness of the test-first approach to programming. IEEE Tran Soft Eng **31**(3), 226–237 (March 2005)
6. Fucci, D., Erdogmus, H., Turhan, B., Oivo, M., Juristo, N.: A dissection of the test-driven development process: Does it really matter to test-first or to test-last? IEEE Transactions on Software Engineering **43**(7), 597–614 (July 2017). https://doi.org/10.1109/TSE.2016.2616877
7. Gai, K., Qiu, M., Sun, X.: A survey on fintech. Journal of Network and Computer Applications **103**, 262 – 273 (2018). https://doi.org/https://doi.org/10.1016/j.jnca.2017.10.011
8. Haugset, B., Hanssen, G.K.: Automated acceptance testing: A literature review and an industrial case study. In: Agile 2008 Conference. pp. 27–38. Toronto, Canada (2008)
9. Lee, I., Shin, Y.J.: Fintech: Ecosystem, business models, investment decisions, and challenges. Business Horizons **61**(1), 35 – 46 (2018). https://doi.org/https://doi.org/10.1016/j.bushor.2017.09.003, http://www.sciencedirect.com/science/article/pii/S0007681317301246
10. Maximilien, E.M., Williams, L.: Assessing test-driven development at IBM. In: 25th International Conference on Software Engineering. vol. 6, pp. 564–569. Portland, OR USA (2003)
11. Melnik, G.I.: Empirical analyses of executable acceptance test driven development. Ph.D. thesis, University of Calgary, Calgary, Canada (2007)
12. Nagappan, N., Maximilien, E.M., Bhat, T., Williams, L.: Realizing quality improvement through test driven development: Results and experiences of four industrial teams. Emp Soft Eng **13**(3), 289–302 (2008)
13. Pugh, K.: Lean-agile acceptance test driven development : better software through collaboration. Addison-Wesley (2010)