Faculty of Mathematics,
Computer Science, and
Natural Sciences

Research Group
Software Construction

Seminar Winter Term 2015/2016

Full-scale Software Engineering / Current Trends in Release Engineering

FsSE / CTRelEng 2016

February 4, 2016

Supervisors

Prof. Dr. rer. nat. Horst Lichter

Dipl.-Inform. Andreas Steffens

Andrej Dyck, M.Sc.

Konrad Fögen, M.Sc.

Muhammad Firdaus Harun, M.Sc.

Simon Hacks, M.Sc.

Contents

How Much Does Testing Cost?

Alexandra Keus
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
alexandra.keus@rwth-aachen.de

Andrej Dyck
RWTH Aachen University
Software Construction
Ahornstr. 55
52074 Aachen, Germany
andrej.dyck@swc.rwth-aachen.de

ABSTRACT

Testing is often seen as a mainly time and money consuming activity within the software engineering process. For this reason, many companies choose not to automate tests or even eliminate tests completely in order to save expenses. However, the costs of running automated, manual, or even no tests at all differ immensely and are not always obvious at first glance.

There exist metrics that determine the costs of automated testing. Though, cost metrics for manual and not running tests are not studied. This paper presents a promising metric for automated tests and proposes two metrics for the two other cases. An example illustrates the use of the metrics. Having those metrics at hand, it is possible to make a proper comparison between the three test approaches and argue how much testing costs and whether it makes sense not to test.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.8 [**Software Engineering**]: Metrics—*process metrics*

Keywords

automated testing, manual testing, test costs, cost estimation, metrics

1. INTRODUCTION

According to Boris Beizer, an American software engineer, 30 to 90 percent of the labor during the software development process are spent testing [5]. For this reason, many companies try to reduce their software engineering costs by reducing the test costs. There exist several different approaches to do so. They range from simply testing less "in less important areas" [16] to save time, to the use of programs or tools which shall decrease the expenses of the development process [1].

Cost savings, by not spending developers' time on the implementation of tests or manual testing, are obvious if no errors are found while testing. Since "program testing can be used to show the presence of bugs, but never to show their absence" [6], testing cannot guarantee a faultless software. This can lead to the assumption that testing does not make sense. But not testing is not necessarily cheaper than actually executing an automated or manual test. To determine the costs of testing, software metrics are needed.

Software metrics present a way to evaluate, thus compare and review, software. They provide the opportunity to make predictions of future software performance and current code quality [7]. Software testing metrics especially refer to the evaluation of software tests. An example for such a metric is the estimation of a tests efficiency, by comparing the errors found while applying this test with the overall number of errors found [9].

There exist metrics which compute the costs of automated tests, however metrics for manual tests and not testing are not present. To resolve this lack of information, this paper presents metrics to determine the costs of manual testing and leaving tests out. Those are based on an existing automated test metric, to simplify a later comparison of the results. Comparing the three test approaches in matters of their costs helps to reason for or against the development and execution of a test.

Section 2 gives an overview about the costs of manual testing, automated testing as well as not testing. To this end, a metric which computes either the costs of running a test or leaving it out is presented. Section 3 gives an example of use for these metrics. In the 4th section, a discussion about testing is held: whether or when it should be done, what the limits are, and what testing eventually costs. Finally, section 5 concludes this paper.

2. TEST COSTS

Test costs describe what the expenses of a test, from the requirement determination until its final completion, are. In this paper, the term test costs does not include the costs of economic damages, which are caused by not testing or testing too less. Of course, the possible damages caused by using certain testing approaches need to be considered. However, it is difficult to estimate the costs of possible economic damages. Therefore, this paper only focuses on parameters which have a direct influence on the costs of testings. The aim is to finally get a metric which is as simple as possible and as detailed as necessary. Thus, not every possible factor which could influence the overall test costs is included in the metrics presented in this paper.

Test costs can be calculated, though they vary depending on the kind of test used. To compute them, suitable metrics are needed. In the following sections three different metrics are presented, determining the costs for executing automated and manual tests as well as the costs for not testing. These metrics can be used in order to compare the three testing strategies and their costs.

2.1 Automated Tests

"Automated software testing uses automated tools to run tests based on algorithms to compare the developing program's expected outcomes with the actual outcomes. If the outcomes are aligned, your program is running properly and you are most likely bug free. If the two don't align, you have to take another look at your code and alter it and continue to run tests until the outcomes align." [4]

To compute the costs of running an automated test, Microsoft developed a suitable metric [11]. With the help of this metric they try to reduce their test costs, using historic data to determine tests which should not be executed. The data they refer to contains information about former test executions, particularly whether a test actually found an error and whether this error was a false alarm or not. A false alarm in this case means that a bug is reported although the code is correct.

To determine when a test should be skipped, Microsoft developed a program which takes this decision. Though this program skips certain tests, those will not simply be left out, but instead run at a later date or on another code branch. At the latest when the code change gets integrated in the program code all tests need to be run at least once. How to decide when to execute a test and when to skip it, however, will not be part of this paper. Instead, the paper now provides a closer look at the metric which determines, according to Microsoft, the costs of running an automated test.

The concrete cost metric, which Microsoft developed, to determine the costs of executing an automated test ($Cost'_{autom}$) consists of different factors, since test automation includes the determination of the requirements as well as the automation and the inspection of the result of the actual test [18]. The machine costs ($Cost'_{machine}$), which describe the infrastructure costs per minute, are part of the automated testing costs. The inspection costs ($Cost_{inspect}$) are influenced by the probability having to inspect an error which is a false alarm, whereat no actual bug is found by the test. The probability of having a false alarm, thus a false positive result, is thereby described as:

$$P_{FP} = \frac{\#falseAlarms}{\#executions}$$

The number of false alarms and the number of executions both depend on one test executed on one environment. Changing the environment on which the test is executed can also influence the corresponding probability of having a false alarm. The costs of having to inspect a false positive test result are described as:

$$Cost'_{errorFP} = P_{FP} * Cost_{inspect}$$

According to Microsoft, the execution costs for an automated test then are [11]:

$$Cost'_{autom} = Cost'_{machine} + Cost'_{errorFP} \tag{1}$$

The metric presented by Microsoft needs to be further enhanced, because it is influenced by factors which are not necessarily known. This means that, for example, the inspection costs of a test might not be known. However, it is possible to deconstruct these factors into ones which are probably known. The following formulas all consist of these simple parameters. First of all, time as a central factor of the error cost computation, is included in the new metric. Since the costs for automated testing also include the costs for implementing the actual test ($Cost_{impl}$), this factor needs to be included in the enhanced metric. These costs depend on the hourly rate of the developer ($Cost_{pay}$) and the time t_{impl} needed to implement the test, as well as define the requirements and the machine costs of the working station (C_{wS}):

$$Cost_{impl} = (Cost_{pay} + C_{wS}) * t_{impl}$$

Furthermore, the machine costs of the server depend on the duration of the test execution since they are usually specified in dependency of time:

$$Cost_{machine} = Cost'_{machine} * t_{exec}$$

The inspection costs, and therefore the costs of having a false alarm, can be further broken down into the payment of the developer and time needed to take a look at the error as well as the machine costs of the working station:

$$Cost_{errorFP} = P_{FP} * (Cost_{pay} + C_{wS}) * t_{error}$$

Finally, this results in

$$Cost_{autom} = Cost_{impl} + Cost_{machine} + Cost_{errorFP} \tag{2}$$

as the extended metric which calculates the costs of the first execution of an automated test. Thus, all parts of the metric now consist of the machine time, hourly rate of the developer, time of the test execution or duration of the test development and the probability of having a false alarm.

2.2 Manual Tests

"Manual Software Testing is the process of going in and running each individual program or series of tasks and comparing the results to the expectations, in order to find the defects in the program. Essentially, manual testing is using the program as the user would under all possible scenarios, and making sure all of the features act appropriately." [4]

Based on the enhanced metric of section 2.1, this paper now presents a corresponding metric determining the costs of a manual test. Since preferably not only the developer should test his program, because he will not be able to test his own product unbiased, the requirements of it have to be settled before testing manually [18]. The manual test costs ($Cost_{man}$) consist of the testers payment according to the time the tester needs to accomplish the given task. Additionally, the machine costs of the working station ($Cost_{wS}$) need to be taken account of. With $Cost_{pay}$ as the the hourly rate of the tester and t_{test} as the testing duration and time needed to define the requirements of the tested program, this results in

$$Cost_{man} = (Cost_{wS} + Cost_{pay}) * t_{test} \tag{3}$$

as a metric which computes the costs of a manual test.

2.3 Not Testing

To determine the concrete costs of not testing, a suitable metric is presented. It seems that not testing software can easily result in fewer expenses during the software engineering process. The costs described by equations 2 and 3 could simply be saved if testing would not be performed. Thus, not testing seems to be a good idea, especially if the probability of having a false alarm is high.

Since there is no testing, there is also no need to develop a test. Thus, the implementation costs, as well as the costs for determining the requirements, are saved. While not executing a test results in savings when the test would find no bug or even produce a false alarm, there are still times when a test would have detected an error. The costs of an escaped error ($Cost_{escaped}$) depend on the machine costs of the working station ($Costs_{wS}$) as well as the salary of the developer ($Cost_{pay}$) in dependency of the required time to fix the detected bug (t_{fix}):

$$Cost_{escaped} = (Cost_{wS} + Cost_{pay}) * t_{fix}$$

The overall costs of the escaped errors furthermore depend on the number of bugs which have not been found, due to not testing. Thus this results in

$$Cost_{noTest} = Cost_{escaped} * \#bugs \tag{4}$$

as the metric determining the costs of not testing.

3. APPLIANCE

Whether testing automatically, manually or not at all - all testing approaches have advantages as well as disadvantages. Thus, not every kind of testing makes sense in every situation.

A big benefit of automated testing is that the costs will reduce when repeatedly applying a test, because it is not necessary to implement the test more than once. At first, the implementation has to be paid for, but later on the expenses will only include the machine and inspection costs. Nonetheless, the final inspection of the test results still has to be done manually [18]. Inserting large amounts of data into a database or testing the performance of the program, while many users simultaneously access data, are things which definitely should be automated. It's not efficient to test such things manually, since the tester would only repeat similar actions over and over again or too many people would be needed to test sufficiently [12].

Manual testing, though, makes sense if the test needs to be changed frequently or is only applied once, so that the costs for the implementation of the test would exceed the costs of testing manually. Another example for the use of manual tests is checking display outputs or sensor interaction [13]. Both cases either demand a variety of human interaction or an aesthetic comprehension, which is easier tested manually than automatically.

There are several different types of errors that can occur during software engineering. They range from minor, aesthetic errors to expensive or even perilous errors. Examples for serious errors are the failed start of the Ariane 5 and a bug in the Therac-25 radiation therapy machine, which caused injuries or even the death of at least six people [3] [14]. Not testing can be problematic in case an error causes something life-threatening or a high damage, like in the examples given above. Depending on the extend of the error, the costs of not testing will vary.

Table 1: Fictive Historic Test Data

Machine Costs (Server)	0.03$/min
Machine Time	00:08
Implementation Time	01:20
Salary	0.39$/min
False Error Probability	4%
Inspection Time	00:36
Machine Costs (Working Station)	0.01$/min
Duration of Manual Testing	00:45
Escaped Bug Fixing Time	00:53
Number of Bugs	14

The metrics presented in section 2 can be especially interesting for people who want to compare the costs of testing by some means or other. It can also help to reason why testing should be done from the financial point of view, since testing does not only cause costs, but also reduces the costs of future bug fixes.

3.1 Example

The presented metrics give a brief overview about the possible costs of testing, however, to compare the test cost it is easier to compute concrete values with help of an example. A simplified chart of fictive historic data concerning salary, time needed to test etc. can be found in table 3.1 [2] [11]. With help of this information the costs of testing are determined.

First of all, the costs of executing an automated test are computed:

$$\begin{aligned} Cost_{autom} &= Cost_{machine} + Cost_{errorFP} + Cost_{impl} \\ &= 0.03 * 8 + 0.04 * (0.01 + 0.39) * 36 + (0.01 + 0.39) * 80 \\ &= 0.24 + 0.58 + 32 \\ &= 32.82 \end{aligned} \tag{5}$$

For the cost and time values given in table 3.1 this results in overall costs of 32.82$ for a test execution. Next follows the computation of the costs of a manual test:

$$\begin{aligned} Cost_{man} &= (Cost_{wS} + Cost_{pay}) * t_{test} \\ &= (0.01 + 0.39) * 45 \\ &= 0.40 * 45 \\ &= 18.00 \end{aligned} \tag{6}$$

Thus, 18.00$ need to be spent per execution. Finally the costs of leaving tests out are calculated:

$$\begin{aligned} Cost_{noTest} &= Cost_{escaped} * \#bugs \\ &= (0.01 + 0.39) * 53 * 14 \\ &= 296.80 \end{aligned} \tag{7}$$

Computing the costs of not testing, according to our metric, results in 296.80$. Per bug this are average costs of 21.20$.

The costs of the three testing methods are very different. With the given values of table 3.1 the outcome is that manual testing results in fewest costs, next followed by automated testing. Not testing is the most expensive testing approach. However, when taking a look at the costs of only one bug, the costs of not testing lie in between the costs of automated and manual testing. Nevertheless, manual testing only costs fewest if the costs of the first execution of an

Table 2: Data Used for Test Cost Charts

Automated Test Costs	Manual Test Costs	Costs of Not Testing	Number of Bugs
32	18.00	296.80	14
0.82	18.00	0.00	0
0.82	18.00	84.80	4
0.82	18.00	21.20	1
0.82	18.00	212.00	10
0.82	18.00	169.60	8
0.82	18.00	254.40	12
0.82	18.00	63.60	3
0.82	18.00	0.00	0
0.8	18.00	42.40	2

automated test are determined and compared to those of a manual test.

The average costs of an automated test decrease with each test execution. The automated testing metric includes the factor of the implementation of a test which only has to be done once for each test. When repeatedly executing the same test, or only changing it slightly, those costs will nearly disappear. Without having to pay for the implementation, only 0.82$ would have to be spent for the test execution, including machine and inspection costs. The more often a test is executed, the cheaper it gets on an average. After two executions the average test costs for each of them would be 16.82$ and after three test executions the average costs would even drop down to 11.49$. When running the same test 100 times, the average costs are already lowered to only 1.14$ per execution. In this example, the average costs of an automated test after two test execution are already lower than the average costs of testing manually.

Manual test costs depend on the time needed for the execution of the test. This time can be very high but also very low and is different for every test. Therefore, executing a manual test can sometimes be cheaper than executing an automated test. The costs of not testing increase, the higher the number of bugs and the required fixing time is. Even with a low number of bugs and a low time needed to fix the bugs, the resulting costs of not testing can be very high.

3.2 Visualization

The charts in figures 1-4 show how the test costs change for each test execution, respectively, how high the costs of not testing can be. The values determined in section 3.1 show the costs of executing one test once on one feature. In the following section the values of the metrics are now related to the number of releases of this feature, supposing that the same test is executed, or not executed, once for each release. For the visualization is also assumed that there is no fluctuation in the test data so that, for example, test times do not vary. The concrete values used for the test cost charts can be found in table 3.2.

Figure 1 shows that, when executing an automated test for the first time, the costs are very high. This value is the one computed by equation 5. Though, from the second execution on the costs begin to lower. The implementation of the test is completed and the costs now only consist of the machine costs and the costs of having to inspect a false

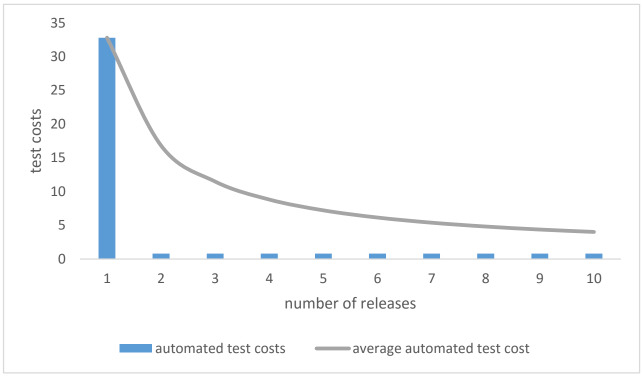


Figure 1: Automated Test Costs

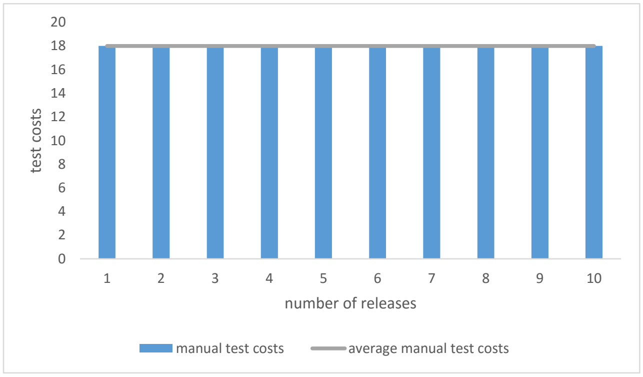


Figure 2: Manual Test Costs

positive error:

$$Cost_{machErr} = Cost_{machine} + Cost_{errorFP}$$

Per execution this would result in 0.82$ which have to be spent after the initial implementation is done. Because of this, the average costs per test lower constantly with each test execution - as displayed in the corresponding chart.

Manual test costs always depend on the time needed to complete the current test task. When repeatedly applying the same test on the same feature, this time will remain constant. Accordingly, the costs of executing a manual test and the average costs of a manual test stay constant for each release as well (see figure 2).

The costs of not testing can be very different. These costs depend on the number of bugs which need to be fixed after each release, as well as the time needed to accomplish this. Since, according to experience, almost every code has bugs before testing, those costs are rather high than low. Figure 3 shows how the number of bugs, found in one release, influences the overall costs. The average costs of not testing vary considerably depending on the detected quantity of those.

Figure 4 shows a cost comparison of the average costs of all test approaches. It is obvious that automated tests and manual tests generally result in fewer costs than leaving tests out. Depending on the number of executions of a test and time needed for the implementation of an automated test, either automated or manual testing is the best way to keep software testing costs low - especially when having similar data like given in table 3.1. Not testing has far higher

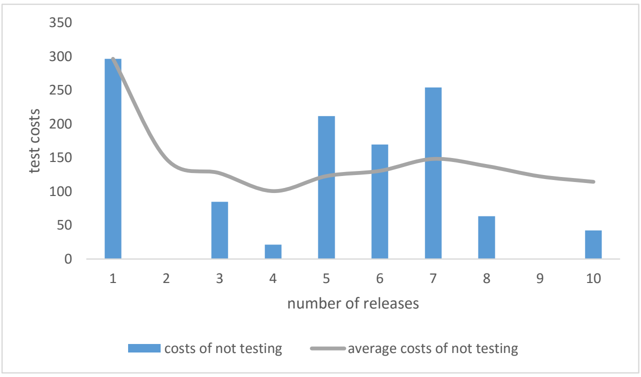


Figure 3: Costs of Not Testing

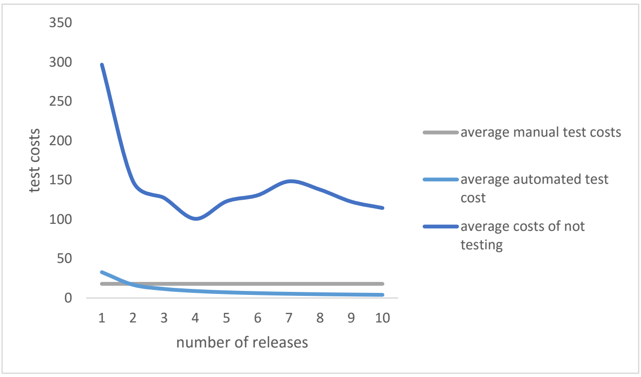


Figure 4: Cost Comparison

average costs, always depending on how many bugs need to be fixed after each release due to not testing and how much time is needed for this.

4. DISCUSSION

This discussion approaches general issues which occur in matters of testing, provides further details about Microsoft's testing approach and compares the three presented test metrics with each other. After presenting those metrics, the question is which method has the lowest costs and whether, respectively when, testing makes sense.

The metrics presented in this paper all use some kind of historic test data. Not all information about former test executions are actually stored for future use. Since this is the case, either popular data needs to be used in the required software testing metric or the corresponding data needs to be gathered or estimated before a proper use of the metric is possible. Gathering this information takes some time - the this period is, the better will the estimation of average values be. This also results in more reliable results given by the metrics.

Microsoft's approach for testing and test cost reduction works by skipping some tests and instead running them at a later time. This idea is based on the consideration that the skipped test will not find an error when being executed, but will instead only lead to higher software testing costs. Their idea of test cost reduction is only applied to automated tests and use historic data to do so.

If the program developed by Microsoft decides to skip a certain test, there is still the chance that the program makes a wrong decision. The metrics used by the program can only estimate how high the probability is that a test will not find an error. The program then, based on this result, decides whether it is cheaper to execute or skip a test. Because of the possibility of making a wrong decision, Microsoft determines that every test needs to be run at least once for each code change. This might delay the execution of a test, but never skips it completely. Hence, Microsoft does not support a general not testing approach.

By delaying the test execution, multiple executions of a test on a code change can be prevented. The longer the execution is delayed, the more changed code can be tested. Using their kind of test selection can result in fewer expenses but does not necessarily has to. When applying their new testing method on their products, Microsoft got very good results with Windows, but when applying it on Office the cost savings were much lower [12].

The reason for these lower savings depend on the effort it takes to fix errors which have not been detected right away, because the corresponding test has been skipped. The earlier an error is found, the smaller is the influence of it on other code parts and the least effort has to be made to locate and fix it. If an error is not found while testing and the code change is submitted into the code trunk, this error can also have an influence on future code changes of other developers. The more people are influenced by an escaped error and the more advanced the software development process is, the more expensive will the corresponding error be [18]. The worst case is that the error is not found until the product has already been delivered. This cannot only result in high costs to eliminate the error, but also in bad publicity for the company.

Automated testing should be used if a certain test has to be executed many times or if the insertion of large amounts of data into a program is necessary. It is also not practicable to test the performance of a program, for example testing the behavior of it while thousands of users access it at the same time, without the help of automated tests [12]. As seen in section 3, the implementation costs have the largest impact on the overall automated test costs. Correspondingly, tests, or at least parts of existing code, should be reused as much as possible while working with automated tests. Using automated tests is the cheapest way of testing if repeatedly applying the same test. But the use of automated tests does not always make sense [12]. The implementation of automated tests is effort and time consuming. In some cases the effort to develop an automated test is higher than the actual savings that could be made by using it, which is especially the case for smaller projects.

If a test needs to be changed a lot and the costs for the implementation of a test would be higher than its benefits, manual testing can be a good idea. Also, when testing appliances which require much user interaction, doing manual tests can make more sense than testing otherwise. Examples for such interactions are, among other things, the testing of displays and sensors [13]. There are also simply tests which cannot be automated, like testing the comprehensibility of a program [15]. Still, the costs of manual testing can be unreasonable high if the decision to automate a, yet manual, test is not made sufficiently early [8].

Not testing is a good idea if the developed program has a very low probability of an error occurrence. When, on the other hand, having a high error probability or a high number of bugs, not testing is very expensive. Also, when choosing not to test, it is important to make sure that having an error will not lead to any life-threatening effects inflicted by the program.

While testing also keep in mind that there are errors from different extends. When a deadline is reached, there should have been at least as much testing that errors with a high severity are eliminated. Thus, it can make sense to sort the tests in this regard or prioritize the inspection and correction of the errors. To prioritize the tests, software testing metrics can be used [17]. Especially when selecting tests is necessary because of strict deadlines, it should be kept in mind that a cheap test is not necessarily the one which needs to be executed first.

5. CONCLUSION AND FUTURE WORK

Generally it cannot be said that one of the test approaches is better than another. If the probability is very high that a test will fail, running it can just raise the test costs and not do any good. On the other hand leaving out a test that will probably find an error will not do any good either, except for saving the costs for just now. Therefore, it is very difficult to generalize whether testing should be done. It always depends on the actual appliance.

Thus, in most cases not testing is rather a disadvantage than an advantage. When choosing how to test, the indirect impact a testing method can have on the overall test costs should not be disregarded. Escaped errors in an already released product, can lead to high economical damages as well as high fixing costs. Too many errors can even harm the reputation of a company.

Nevertheless, there are people who think that too much testing can rather lead to worse than better software [10]. The idea behind this is as follows: the more testing is done, the less effort is put by the developer into improving and checking his own code. Thus, even simple errors are only detected while testing and the focus of testing is shifted to the elimination of small and easy detectable errors - the ones improved testing would not have been necessary for.

The concrete costs of a test highly depend on the given data and have to be computed individually. To do so, the metrics presented in this paper can be used. Even if the metrics do not consider all possible impacts on test costs, they nevertheless provide a good overview about those.

An idea for future work is to take account of the severity of an error when computing the test cost. Lower severity errors might cost nearly nothing, while errors with a high severity can result in high costs, regardless of the development stage the error was found in. Furthermore, a comparison of actual test costs provided by a company and the corresponding costs computed by the metrics presented in this paper would be great to check the metrics' validity.

6. REFERENCES

[1] Better software in less time. `http://www.savignano.net/de/products/savvytest/`. Retrieved January 11, 2016.

[2] Software Developer Salary (United States). `http://www.payscale.com/research/US/Job=Software_Developer/Salary`. Retrieved January 11, 2016.

[3] Ariane 5. `http://www.esa.int/Our_Activities/Launchers/Launch_vehicles/Ariane_5`, 2015. Retrieved November 27, 2015.

[4] R. Arsenault. When You Should Choose Manual vs. Automated Testing. `https://www.utest.com/articles/when-you-should-choose-manual-vs-automated-testing`, 2014. Retrieved January 11, 2016.

[5] B. Beizer. *Software Testing Techniques (2nd Ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[6] P. D. E. W. Dijkstra. Notes on Structured Programming. `http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF`, 1970. Retrieved December 13, 2015.

[7] T. Garrett. Implementing Automated Software Testing - Continously Track Progress and Adjust Accordingly. `http://www.methodsandtools.com/archive/archive.php?id=94`, 2009. Retrieved October 22, 2015.

[8] J. Grenning. Manual Test is Unsustainable. `http://www.renaissancesoftware.net/blog/archives/206`, 2012. Retrieved January 20, 2016.

[9] L. Gulechha. Software Testing Metrics. `http://artemisa.unicauca.edu.co/~cardila/CS__Software_Testing_Metrics.pdf`. Retrieved October 22, 2015.

[10] E. Hendrickson. Better Testing - Worse Quality? `http://testobsessed.com/wp-content/uploads/2011/04/btwq.pdf`, 2000. Retrieved January 11, 2016.

[11] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy. The art of testing less without sacrificing quality. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 483–493, Piscataway, NJ, USA, 2015. IEEE Press.

[12] D. Hoffman. Cost benefits analysis of test automation. 1999.

[13] S.-J. Jang, H.-G. Kim, and Y.-K. Chung. Manual specific testing and quality evaluation for embedded software. In *Computer and Information Science, 2008. ICIS 08. Seventh IEEE/ACIS International Conference on*, pages 502–507, May 2008.

[14] N. Leveson and C. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, July 1993.

[15] P. Liggesmeyer. *Software-Qualität*. Spektrum Akademischer Verlag, 2009.

[16] H. Schaefer. Risk Based Testing, Strategies for Prioritizing Tests against Deadlines. `http://www.methodsandtools.com/archive/archive.php?id=31`, 2005. Retrieved January 11, 2016.

[17] M. Walia. Realizing Efficiency & Effectiveness in Software Testing through a Comprehensive Metrics Model. `https://www.infosys.com/engineering-services/white-papers/Documents/comprehensive-metrics-model.pdf`, 2012. Retrieved October 22, 2015.

[18] F. Witte. *Testmanagement und Softwaretest*. Springer Fachmedien Wiesbaden, 2016.

A Study of Tools for Behavior-Driven Development

Anton Okolnychyi
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
anton.okolnychyi@rwth-aachen.de

Konrad Fögen
RWTH Aachen University
Software Construction
Ahornstr. 55
52074 Aachen, Germany
konrad.foegen@swc.rwth-aachen.de

ABSTRACT

Behavior-Driven Development (BDD) has obtained a lot of attention in recent years from both research and practice points of view. As a new Agile development approach, it is aimed to increase the likelihood of success of a software project by adopting best practices and concepts from Test-Driven Development and Acceptance Test-Driven Development and correcting their drawbacks. There are a lot of tools that were developed in the last few years to assist software developers in BDD. While this study describes underlying concepts and BDD itself, the main goal of the research is to develop criteria for identifying relevant tools which can be applied in BDD, evaluate and compare them and provide guidelines on which toolkit to choose in order to achieve success in a project. The research approach employed in this study is composed of reviewing relevant literature and analyzing current BDD toolkits for JVM-based languages.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.9 [**Software Engineering**]: Management—*productivity, programming teams, software configuration management*

Keywords

Behavior-Driven Development, Test-Driven Development, Automated Acceptance Testing

1. INTRODUCTION

Behavior-Driven Development (BDD) was introduced recently as one of the methods in Agile software development. BDD differs from other approaches in its family by describing a behavior of the system from the perspective of its stakeholders, at all levels of granularity [21]. BDD assures that focusing on such description of the behavior of the system gives better communication and produces a bigger asset for stakeholders when compared to other Agile development methods. It was originally developed and described by D. North in his post [28] as a response to the issues in Test-Driven Development (TDD). BDD is based on Test-Driven Development and Acceptance Test-Driven Development [27]. D. Astels in [19] declared that even those people who apply TDD a lot do not make use of all benefits from TDD and important aspects of TDD are overlooked and simply ignored. He suggested that a big part of developers are focused on writing verification tests instead of thinking in terms of behavior specifications. Taking into account behavior specifications allows software engineers to think more clearly about each behavior, relying less on testing by a class or by a method, and having better executable documentation.

The paper is structured as follows. Section 2 describes the concepts of BDD and other inherited approaches which are needed to understand the requirements for BDD tools. Section 3 gives an overview of the research approach which was used to identify relevant tools for this study. In addition, Section 3 defines diverse dimensions for comparing BDD tools, describes each analyzed toolkit in terms of those dimensions and provides the overall summary of comparison. The last section gives the conclusions.

2. UNDERLYING CONCEPTS OF BDD

BDD is generally regarded as the evolution of TDD and ATDD. This section will briefly describe relevant aspects of TDD and ATDD in terms of BDD.

2.1 Test-Driven Development

Test-Driven Development is a development practice that involves writing tests before writing the code being tested. One should begin by writing a very small test for code that does not yet exist [21]. TDD is an evolutionary approach that relies on very short development cycles and the agile practices of writing automated tests before writing functional code, refactoring, and continuous integration [24]. Each development cycle consists of three steps: the creation of unit test, implementation, refactoring [23]. The aforementioned approach is named TDD since tests, written during the first steps of each iteration, drive the design and implementation. As a code base increases in size, more attention is consumed by the refactoring step. The design is constantly evolving and under constant review though it is not predetermined. This is emergent design at a granular level and is one of the most significant by-products of Test-Driven Development [21].

The evaluation [26] by R. Jeffries and G. Melnik claims that the overall quality of a system in terms of the density of defects improves, although the required effort often increases. A study described in [25] suggests that developers are able to produce a better design of a system with well-focused units with a help of TDD.

2.2 Acceptance Test-Driven Development

Acceptance Test-Driven Development (ATDD) is one type of TDD where the development process is driven by acceptance tests that are used to represent stakeholders' requirements [29]. M. Wynne and A. Hellesoy in [30] justify the name of acceptance tests as such tests express what the software needs to do in order for the stakeholder to find it acceptable. In the same book they state that in ATDD instead of a business stakeholder providing requirements to the developers without any discussion, the developer and stakeholder work together to write automated tests to satisfy the stakeholder.

ATDD assists developers in the creation of test cases based on initial requirements of a system. There is a set of tests or acceptance criteria that correspond to one specific requirement. One can say that a requirement is satisfied if all its associated tests or acceptance criteria are satisfied. In ATDD acceptance tests can be automated. ATDD emphasizes automation of acceptance tests and the specification of customer-readable requirements through concrete examples, which is also referred to as specification by example [18]. Automated acceptance tests encourage all people involved into the process to be focused on the aims of the software projects. Automated acceptance tests help your team to focus, ensuring the work you do each iteration is the most valuable thing you could possibly be doing [30].

TDD and ATDD are adopted widely by the industry because they improve software quality and productivity [19] [25].

2.3 Behavior-Driven Development

The main goal of BDD is to get executable specifications of a system [28] [19]. Dan North stated that the main reason for introducing Behavior-Driven Development was the fact that Test-Driven Development was often perceived as a testing technique. He replaced the word "test" in the name of TDD with "behavior" in order to emphasize that TDD is about design, not testing.

BDD has adopted the concept of a ubiquitous language from Domain-Driven Design [21]. A successful software project requires good communication, which in turn relies on a shared language. Domain experts think and reason in terms of their domain language. Developers do the same, using concepts from the domain of software development. Analysts and developers translate between these domains, mapping domain concepts to design. However, information can be lost in this translation, which causes different people to have different interpretations of concepts [27]. As Eric Evans describes in his book [22], many software projects suffer from low-quality communication between the domain experts and programmers on the team. Tests written with a help of tools for BDD are usually defined using a language that business stakeholders can understand.

One of the key concepts of the BDD is involvement of all stakeholders which is possible via ubiquitous language. Business analysts write down behavioral requirements in the way that will also be understood by developers who later transform these requirements into executable tests. By working together to write these tests, team members decide what behavior they need to implement next. They learn how to describe that behavior in a common language that everyone understands [30].

Currently, the understanding of BDD is far from clear and unanimous. There is no one well-accepted definition of BDD [29].

3. COMPARATIVE STUDY OF BEHAVIOR-DRIVEN DEVELOPMENT TOOLS

This section is aimed to compare BDD tools as well as to describe a research approach that was used to select certain frameworks from a huge number of tools that are present now. The final comparison can be found in table 1. The full support of a specific feature is marked by "+". By "+/-" or "+/- -" is marked partial support depending on the extent.

3.1 Approach for Identifying Relevant Tools

The need to involve all stakeholders in the development process spawned a number of new tools which are aimed to assist all types of stakeholders in applying BDD. Particularly, new tools were needed to help non-technical people to read and understand acceptance tests, although the old tools could still be used and many still continue to do so.

The goal of this research is to create an approach to identify the tools and frameworks which are relevant and can be applied successfully in BDD. BDD is just a technique which can be used without any tools and frameworks. This means that developers can try to utilize not only BDD specific frameworks but also most of the tools for TDD. However, TDD tools tend to be quite free-format and it will take a different amount of time and effort to benefit from those TDD tools in BDD context.

Support to some extent of ubiquitous language is the main criterion and BDD characteristic that was used to distinguish relevant tools for BDD in this study.

A lot of tools from different languages were analyzed during the research. Due to the aforementioned selection approach, the following frameworks were considered as those that cannot be used standalone as BDD frameworks: strictly unit-testing tools for all languages (JUnit [9], etc.), tools for mocking (EasyMock [6], Mockito [10], etc.), most UI-testing tools (Selenium [13]), frameworks for testing Web Services and databases. On the other hand, they are often combined with real BDD tools.

This study focuses on BDD tools for JVM-based programming languages (Java, Groovy, Scala) with a strong support of ubiquitous language. To determine the relevant BDD frameworks to compare, the Wikipedia list [1] was used as the initial source. The most frequently mentioned tools were selected with a help of a search by tags on stackoverflow.com. The last step was to filter the frameworks for JVM-based languages since they can be directly and fairly compared. As a result the following tools were included in the analysis: Concordion [3], Spock [15], Cucumber [4], JBehave [8] and easyb [5]. In addition, Serenity (previously known as Thucydides)[14] framework was considered but not included in the comparison. It is less popular with the small community and the main benefit of it is reporting. Selected frameworks satisfy all main BDD requirements and match specific needs of the study. Therefore, these frameworks were further compared.

3.2 Dimensions for Comparison

Different dimension for comparing BDD frameworks were found during the study. BDD is a technique which is perfectly applicable at various levels. For instance, it can be applied at the code/unit level and at the acceptance/integration level as well. Moreover, these usages are not exclusive and can be combined.

3.2.1 *Comparison Based on a Primary Target Group*

One dimension for comparison was inspired by J. Band who differentiates the following flavors of BDD tools based on their origins and target groups in [20]:

1. Tools with a business readable output
2. Tools with a business readable input

Frameworks from the first category are usually focused on the developers. All artifacts involved are owned by the developers and are typically code. This does not make such frameworks useless since responsible and committed developers are often the main stakeholders in successful software projects. Other stakeholders get only reports which they can understand [20]. Such kind of frameworks is usually seen as a replacement/extension for TDD at a unit-testing level.

Tools from the second category (business readable input) try to widen the focus of the BDD process by enabling the bigger involvement of all other stakeholders: customers, business analysts, testers maybe even operations. This involvement is possible upfront, meaning before the developers have done their work [20]. Such kind of tools is usually aimed at ATDD.

3.2.2 *Comparison Based on Support of Characteristics of BDD*

Another dimension for comparing tools comes from characteristics of BDD. The following main characteristics were identified during the study:

3. Ubiquitous language

This concept is an integral part of BDD. Therefore, support of this characteristic was used as a selection criterion for tools that were compared in the study. Creating the ubiquitous language needs to involve anyone (domain experts and developers) who will use the language.

The important point at this moment is to distinguish the ability of tools for creating a ubiquitous language based on the business domain and ability to use a predefined version of such language which is domain independent. BDD itself also includes a predefined simple ubiquitous language for the analysis process [29].

4. Automated Acceptance Testing

All scenarios must be run automatically. This requires automatic import and analysis of acceptance criteria. The code responsible for the execution usually has to read the plain text specifications and process them in a corresponding way. Such approach lets stakeholders have executable plain text scenarios. In this case, there also should be a standard mechanism of mapping scenarios to test code which executes them. However, scenarios can be simply defined directly in code.

5. Templates for plain text description of user stories and scenarios

Descriptions of features, user stories and scenarios cannot be done in an arbitrary form in BDD. All of them should follow the existing templates and guidelines.

Each user story describes an activity done by a user, clarifies a role of the user and which feature of a system allows the user to perform this activity. Moreover, each user story outlines the benefit which the user acquires after performing the activity. Such template contributes to a clear way of representing features the system should support and why they should be supported by the system. In addition, such approach helps to understand what features are more important by comparing the benefits which they provide. Developers may use this information to adjust their strategy, priorities, and deadlines.

A scenario describes how the system that implements a feature should behave when it is in a specific state and an event happens. The outcome of the scenario is an action that changes the state of the system or produces a system output [29].

3.2.3 *Comparison Based on Specific Features of Selected Tools*

The last but not least dimension to compare BDD tools is based on specific additional features that each tool provides. It is a good idea to combine other useful features with BDD ones since such kind of tools can be used standalone to cover more cases without any need to integrate other frameworks.

The following specific features of analyzed frameworks were considered important:

6. Unit-testing facilities.

There are some TDD techniques that may be helpful in BDD as well. For instance, mocking. It is not a good idea to make use of mocks in acceptance tests on a regular basis. Such tests are supposed to cover the whole system and to test each aspect of it.

By mocking some parts of the system, you exclude them from coverage. However, there are certain cases when mocking is really appropriate: for instance, a module or component of a system can communicate with a 3rd party system. In this case, the scenario depends on the 3rd party system which is out of the control. Therefore, running such scenarios may be difficult and not stable, and the best option here is to mock or simulate that 3rd party system so that your application or product can still be tested.

Another useful application of mocking is to follow ”test as soon as possible” technique. Developers can mock unimplemented parts with predefined behavior and test small parts really early in the development cycle. This approach helps to spot all potential bugs during initial implementation. At this point of time, it is required less amount of time to investigate and fix the issue than when you have a full complex and comprehensive module.

7. Facilities for testing Web applications.

Web applications are extremely popular nowadays. Most of the new applications are developed for usage in Web. Moreover, there is an emerging strategy for application software companies is to provide web access to software previously distributed as local applications. Depending on the

Table 1: Comparison of BDD Tools

Support of Features	Cucumber	Concordion	Spock	JBehave	easyb
Business readable input	+	+	-	+	-
Business readable output	+	+	+	+	+
Creation of a ubiquitous language	-	+	-	-	-
Support of a predefined ubiquitous language	+	-	+	+	+/-
Automated acceptance tests	+	+	+	+	+
Plain text description of user stories and scenarios	+	+	+/-	+	-
Unit-testing facilities	-	-	+	+/- -	+/-
Facilities for testing Web applications	+	+	+	+	+

type of application, it may require the development of an entirely different browser-based interface, or merely adapting an existing application to use different presentation technology [17]. Therefore, it is important for BDD tools to cover Web development and provide corresponding facilities to make this process easier.

There are a lot of high-level frameworks that allow the definition of acceptance tests in natural language. But when it comes to the technical implementation of the test cases, developers often have to use the rather low-level WebDriver API directly. Thus, it is important to consider to which extent modern BDD tools can be used for developing Web applications and how much effort it might require.

Functional web stories are a powerful mechanism to verify the proper behavior of web applications from a user's standpoint. Combining a framework that supports stories and scenarios with other tools for UI tests yields an easy way to deliver software more quickly and collaboratively.

3.3 Comparison of Selected Tools

The following section describes each of analyzed frameworks independently in terms of developed criteria in the previous section.

3.3.1 Cucumber

Cucumber is definitely a framework with a business readable input since it supports writing plain text user stories and scenarios which can be later utilized as a basis for creating automated acceptance tests. Analysis of BDD-related questions on stackoverflow.com during this study confirms that Cucumber is one of the most popular and widely used frameworks of this type.

Cucumber supports various readable report formats. The basic output prints the whole content of the feature which is not always necessary. Luckily, you can easily customize the output to match your needs. Cucumber has a set of built-in formatters. They allow you to visualize the output from your test run in different ways. There are formatters that produce HTML reports, formatters that produce JUnit XML for continuous integration servers like Jenkins, and many more. Moreover, there are a lot of custom formatters which are developed by a huge community of developers who use this framework.

Cucumber does not allow you to create your own domain dependent ubiquitous language. However, it supports a predefined version of a ubiquitous language called Gherkin. It is plain text with a little extra structure. Gherkin is designed to be easy to learn by non-programmers, yet structured enough to allow the concise description of examples to illustrate business rules in most real-world domains. A Gherkin file is given its structure and meaning using a set of special keywords. There is an equivalent set of these keywords in each of the supported spoken languages [30]. This means that developers can write specifications not only in English but also in more than 60 other spoken languages and allows to widen the target group.

Cucumber supports automated acceptance tests. In addition, it is flexible in defining scenarios and it gives you an opportunity to write scenario outlines, share short setup steps or assertions. You can even call step definitions from other step definitions.

Cucumber easily allows to transform plain-text specifications into the code out of the box. However, it does have much to offer in terms of unit-testing due to its main aim and origins.

Cucumber doesn't know how to talk to databases, web apps, or any external system. People install other libraries and use them in their step definitions and support code to connect to those external systems [30]. For instance, you can integrate Selenium or Capybara [2]. The latter framework poses special interest in combination with Cucumber since both of them are written in Ruby. This language fits BDD since it is natural to read. There is no specifically suited framework for UI testing.

Serenity has also a separate module for integration with Cucumber. It is an easy way to get incredible reports that are automatically generated for the BDD tests.

3.3.2 Concordion

Concordion is also a tool with a business readable output. Despite the fact that Concordion requires basics of HTML, it is still a framework from the second category since it allows to write specifications in a highly custom way.

Concordion also provides readable output from tests which can be understood and used by all stakeholders. If all tests are executed then framework produces a complete set of colored output HTML files, which developers or their managers can publish on a web-server. There is also a possibility to use custom CSS or JavaScript, or include images or other resources, in the Concordion output by means of simple extensions. Moreover, there are some existing extensions. For instance, one of them adds screenshots to Concordion output to diagnose problems or improve the documentation.

Rather than forcing product owners to use a specially structured language for specification by example, Concordion lets you write the specifications in a normal language using paragraphs, tables, and proper punctuation. This makes them much more natural to read and write and helps everyone understand and agree about what a feature is supposed to do [3]. However, Concordian requires basic knowledge of HTML which can be a significant drawback. This framework also does not support predefined ubiquitous languages such as Gherkin.

Concordion allows to write automated acceptance tests. It also provides a big level of flexibility in doing it as Cucumber. Moreover, Concordion allows to have and edit plain text descriptions of stories and scenarios.

Concordion does not offer a lot in terms of unit-testing. It as well as Cucumber does not have any specific framework for UI testing that suits particularly well only for it. However, Concordion can be used to test Web applications since it is commonly used with Selenium.

3.3.3 Spock

Spock is a good example of tools with a strictly business readable output. It is not only as powerful as strictly unit-testing frameworks in terms of applicability at code/unit level, but it also supports writing specifications. Spock can not only fully replace JUnit but also provide the extended set of features with mocking and stubbing mechanisms.

Spock does not support the creation of a ubiquitous language. Moreover, it out of the box supports the concept of a ubiquitous language with some significant restrictions. For instance, developers have to mix the story descriptions and code. There is an extension called Pease that creates Spock tests from Gherkin specifications. With Pease, you are able to separate your requirements and your test code and still access the full power of the Spock framework [11].

Spock allows you to write automated acceptance tests. Spock can be used as a replacement or extension for standard unit-testing frameworks, such as JUnit. Moreover, Spock has the widest range of features in terms of unit-testing. It is a complete testing framework with mocking, stubbing, and other helpful techniques.

Spock provides simple integration and takes advantage of Geb framework. Geb is a browser automation framework written in Groovy based on Selenium WebDriver. It is aimed to make all code for modeling behavior of a user on UI pages concise and clear. Spock has also a great support for testing RESTful APIs.

3.3.4 JBehave

JBehave is similar to Concordion and Cucumber since it is a tool with business readable input. It lets execute text-based user stories with a help of Gherkin or its own syntax. JBehave provides different output formats. For instance, it can print a text-based console output, produce a text-based output file, an HTML file or an XML file.

JBehave does not provide an ability to define a ubiquitous language, but it supports the aforementioned Gherkin. In addition, you can make use of its own syntax to describe scenarios.

JBehave can be used to implement automated acceptance tests. It also lets transform plain-text specifications into the code out of the box.

JBehave has limited unit-testing facilities. For instance, this tool bundles a mocking framework known as a Minimock. JBehave has an extension called JBehave Web which provides support for web-related access or functionality. JBehave integration with Selenium and WebDriver APIs aims to facilitate common tasks. Amongst these, one of the most common is the management of the lifecycle, e.g. starting and stopping the browser [8].

JBehave works well with Serenity since there is a separate module in Serenity for combining with JBehave. Serenity uses simple conventions to make it easier to get started writing and implementing Serenity stories and reports on both JBehave and Serenity steps, which can be seamlessly combined in the same class, or placed in separate classes, depending on your preferences.

3.3.5 Easyb

Easyb is one more example of tools with an only business readable output. It is similar to Spock in this respect.

Easyb does not allow to create a ubiquitous language. This framework provides the worst support of the concept of ubiquitous language since the code and specification are mixed together and there was no plugin or extension to support, for instance, Gherkin or any predefined language at the moment of study. However, the code with given/when/then sections helps all stakeholders to get insight about the tested scenario easily enough.

Easyb provides functionality for automated acceptance tests, but there is no way to support plain text descriptions.

Easyb has fewer features at the unit-testing level than Spock, but more than other analyzed frameworks. It also can be used together with Selenium [13], Selenide [12] and Tellurium [16]. Moreover, easyb can be combined with FEST [7] framework to enable testing of Swing-based applications. Tellurium is built on UI module concept, which makes it possible to write reusable and easy to maintain tests against the dynamic RIA based web applications. Selenide is simple and powerful in use wrapper-library over Selenium intended to short the lines of code to make the whole tests more readable and understandable. There is a special plug-in for working with databases.

3.3.6 Summary of Comparison

All analyzed tools are suitable for BDD but they are aimed at different levels. Spock and easyb are focused on the unit-testing level, while JBehave, Concordion, and Cucumber are more suitable for acceptance/integration testing.

Only Concordion supports to some extent creation of a specific ubiquitous language for a project. JBehave, Cucumber support predefined ubiquitous languages, while Spock and easyb have some significant restrictions in this regard. For instance, developers mix the story description and corresponding code using these tools. Even despite the fact that you can use a plain text to define all method names, story and code are very tightly coupled and reside in one file.

All analyzed frameworks support automated acceptance tests. However, Concordion, JBehave, and Cucumber have more ways to define the scenarios. These tools also provide a clear separation between the code and scenarios allowing to define user stories and scenarios in plain text. Hence, these tools are more flexible and powerful for this particular task. Spock has the aforementioned Pease extension which provides the ability to define scenarios in Gherkin, but there is no such solution for easyb.

Both Spock and easyb have much more to offer than Cucumber, Concordion, and JBehave from the unit-testing point of view. However, there are a lot of standalone specific tools such as Mockito, EasyMock which can be integrated into all analyzed frameworks to add needed functionality.

Other toolkits that can be easily combined with analyzed frameworks were mentioned per each framework. Those tools were selected by review of the literature, tutorials, and documentation.

4. CONCLUSIONS

BDD inherits main concepts from TDD and automated acceptance testing augmenting them with other ones such as ubiquitous language. This combination is aimed to make use of all benefits provided by each inherited approach and address their drawbacks. BDD can be adapted and applied at various levels of development. It puts the strong focus on behavior instead of structure at each level. BDD changes the way all stakeholders think about testing. Its main goal to verify what a tested object does and not what the internal structure of the object is. This difference makes a huge impact on the overall development process since behavior is much more significant than the internal structure.

The main intends of the study were to provide all underlying concepts of BDD, develop the research approach for identifying relevant tools for applying BDD and to compare the selected tools for JVM-based languages from different perspectives. One of the most important features of BDD is involvement of all stakeholders in the development process. Therefore, the special attention was paid to the concept of the ubiquitous language. Support to some extent of a predefined ubiquitous language or creation of a new domain specific one was chosen as the criterion to select relevant tools for comparison. The study defines three dimensions for comparing BDD frameworks: based on a target group, on the support of characteristics of BDD and based on specific features of selected tools.

The results of the performed comparison indicate that there is a strong support of main BDD concepts by analyzed tools which makes BDD possible with JVM-based languages. However, the study also shows that tools with better support of unit-testing facilities usually require some tuning to pose an interest for all stakeholders. All analyzed tools have a nice integration with a vast variety of other tools. This is crucial since it enables applying BDD for different kinds of applications. For instance, there is a set of frameworks for each analyzed tool that makes possible BDD for Web applications.

5. REFERENCES

[1] Behavior-driven development. `https://en.wikipedia.org/wiki/Behavior-driven_development`. Retrieved November 20, 2015.

[2] Capybara. `https://rubygems.org/gems/capybara`. Retrieved December 2, 2015.

[3] Concordion. `http://concordion.org/`. Retrieved November 23, 2015.

[4] Cucumber. `https://cucumber.io/`. Retrieved December 9, 2015.

[5] Easyb. `http://easyb.org/`. Retrieved December 9, 2015.

[6] Easymock. `http://easymock.org/`. Retrieved December 9, 2015.

[7] Fest. `https://code.google.com/p/fest/`. Retrieved December 2, 2015.

[8] Jbehave. `http://jbehave.org/`. Retrieved December 2, 2015.

[9] Junit. `http://junit.org/`. Retrieved December 9, 2015.

[10] Mockito. `http://mockito.org/`. Retrieved December 9, 2015.

[11] Pease. `http://pease.github.io/`. Retrieved December 1, 2015.

[12] Selenide. `http://selenide.org/`. Retrieved December 2, 2015.

[13] Selenium. `http://www.seleniumhq.org/`. Retrieved December 2, 2015.

[14] Serenity bdd. `http://www.thucydides.info/`. Retrieved December 9, 2015.

[15] Spock. `https://code.google.com/p/spock/`. Retrieved December 2, 2015.

[16] Tellurium. `https://code.google.com/p/aost/`. Retrieved December 2, 2015.

[17] Web application. `https://en.wikipedia.org/wiki/Web_applicationt`. Retrieved December 2, 2015.

[18] G. Adzic. *Specification by Example: How Successful Teams Deliver the Right Software*. Manning Publications, 2011.

[19] D. Astels. A new look at test-driven development. Technical report, 2005.

[20] J. Bandi. Classifying bdd tools. `http://blog.jonasbandi.net/2010/03/classifying-bdd-tools-unit-test-driven.html`, 2010.

[21] D. Chelimsky, D. Astel, B. Helmkamp, D. North, Z. Dennis, and A. Hellesoy. *The RSpec Book: Behaviour Driven Development with RSpec, Cucumber, and Friends*. Pragmatic Bookshelf, 2010.

[22] E. Evans and M. Fowler. *Domain-Driven Design*. Addison-Wesley Publishing Company, 2004.

[23] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Publishing Company, 1999.

[24] D. Janzen and D. H. Saiedian. Test-driven development: concepts, taxonomy, and future directions. *Computer*, 38(9):43–50, September 2005.

[25] D. Janzen and D. H. Saiedian. Does test-driven development really improve software design quality? *Software, IEEE*, 25(2):77–84, March-April 2008.

[26] R. Jeffries and G. Melnik. Guest editors introduction: Tdd - the art of fearless programming. *Software, IEEE*, 24(3):24–30, May-June 2007.

[27] J. H. Lopes. Evaluation of behavior-driven development. Master's thesis, Faculty EEMCS, Delft University of Technology, 2012.

[28] D. North. Introducing bdd. `http://dannorth.net/introducing-bdd/`, 2006. Retrieved November 1, 2015.

[29] C. Solis and X. Wang. A study of the characteristics of behaviour driven development. In *Proceedings of the 7th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 383–387, 2011.

[30] M. Wynne and A. Hellesoy. *The Cucumber Book: Behaviour-Driven Development for Testers and Developers (Pragmatic Programmers)*. The Pragmatic Bookshelf, 2012.

An Analysis of Current Mutation Testing Techniques Applied to Real World Examples

Daniel Klischies
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
daniel.klischies@rwth-aachen.de

Konrad Fögen
RWTH Aachen University
Software Construction
Ahornstr. 55
52074 Aachen, Germany
konrad.foegen@swc.rwth-aachen.de

ABSTRACT

Mutation testing, a method to improve the quality of a set of unit tests, has been a research topic for more than 30 years but has not yet made it into widespread industry usage. This paper aims to evaluate a current mutation testing tool for the Java programming language named Pitest, in order to assess it's practicality for industry usage. In doing so several possible metrics for the applicability of mutation testing frameworks in real world projects are presented and used to determine Pitests impact on an exemplary open source project. Whereas previous papers mostly focus on decreasing the number of created mutations while keeping the number of living mutants as high as possible, this paper investigates which settings and mutation operators lead to the best improvement of test suites relative to the time consumed by conducting the relevant mutation tests. The test results indicate that the efficiency in finding live mutants largely differs between different mutation operators. Furthermore, a very low percentage of equivalent mutants has been discovered, suggesting that these are less of a problem than commonly assumed. Additionally, several implementation advantages and disadvantages of byte code mutation and mutation injection have been found, albeit generally proofing it's viability.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.9 [**Software Engineering**]: Management—*productivity, programming teams, software configuration management*

Keywords

Mutation Testing, Java, Pitest

1. INTRODUCTION

Unit testing via test frameworks, such as JUnit, is common practice in current software projects [12]. Traditionally code coverage analysis is used to determine whether the used test set is sufficient. However, this only ensures that every line of code has been executed at least once. Especially this does not guarantee that the code completely complies to a given specification.

The idea of mutation testing is to automatically modify a program for which passing tests exist. If the tests no longer pass when run on the mutated program, they are sufficient to detect this mutation and the mutant is considered *killed.* If the mutation is not detected it is considered *live.* There are two different possible reasons for this to happen. Either the mutated program is equivalent to the original program in that equal input always causes equal output and side effects for both versions, or the mutated program is different, but the tests are not sufficient to find the difference. The former might be a sign of dead code[1], as modifying dead code will not show any difference in output, the latter is particularly interesting since it shows that there is the possibility of a programming mistake being undetected by the current set of unit tests. The ratio of the number of killed mutants over the number of total mutations is called *mutation score.* The number of total mutants is determined by the used set of *mutation operators* [15]. Mutation operators determine the syntactical changes a mutation toolkit applies to the program it mutates. The impact of different operator sets on the number of generated mutations will be further analyzed in section 2.1 and section 4. Figure 1 illustrates the usual workflow of mutation testing enabled development.

Considering the following code and the test data set $x = \{(0, \text{false}), (5, \text{true})\}$:

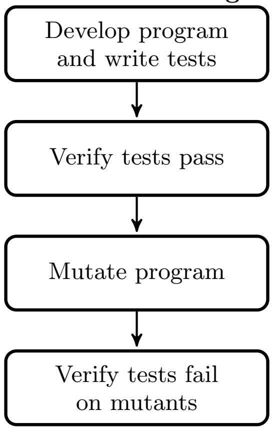
Listing 1: Wrong implementation of a greater than method.

```
function greaterOne(int x) {
  return (x >= 1);
}
```

For the given x this function behaves as expected and, as every line is executed at least once, it would have 100% test coverage. However, a mutation framework could replace `>=` by `>` and the test set x would still pass all tests. Thus the mutant would stay live and point the developers' attention to the unit test set. A test set killing all mutants is $y = \{(0, \text{false}), (1, \text{false}), (2, \text{true})\}$, also detecting that `if (x >= 1)` should be `if (x > 1)` since y will not pass on the original program either.

[1]Code being unreachable and thus never executed.

Figure 1: Mutation testing workflow.



This paper mainly focuses on a mutation framework called Pitest. It is being developed by Henry Coles in a non academic context and is intended to be used in real world projects [8]. Section 2 gives an overview of theoretical aspects of mutation testing and how they are implemented in Pitest. In Section 3 several metrics are developed to measure the quality of a test set and a suitable open source project is chosen to determine the quality of Pitest. The results of the conducted tests are presented in Section 4.

2. PITEST'S IMPLEMENTATION OF COMMON MUTATION TESTING ASPECTS

2.1 Mutation operators

The set of mutation operators determines which modifications to a program's source code a mutation framework may make. Historically a mutator set by King and Offutt[16], originally designed to mutate Fortran programs, has been used as a reference point. In 2011 Ma and Offutt adapted this operator set to the Java programming language. For instance, a commonly used mutator called 'arithmectic operator replacement' (Ma and Offutt) or 'math mutator' (Coles, Pitest), searches for any arithmetic operation and replaces it by another operation. Ma and Offutt proposed that "basic binary arithmetic operators [may be replaced] with other binary arithmetic operators"[18].

However, in Pitest every arithmetic operation is replaced by a predetermined arithmetic operation. Table 1 shows a comparison between the mutator sets of Ma and Offutt and Coles' Pitest. For most operators, Pitest limits the possible replacements. In doing so it reduces the search space, which will reduce the runtime of Pitest as less mutants have to be generated and tested but also reduces the likelihood of an incomplete test staying live.

Apart from these standard mutators Pitest also features mutators removing calls to methods having a void return type, replacing constructor calls by null and replacing return vales by constants. For a full list of all mutators and their exact specification see [11] and section 3.3.

Pitest handles every mutation separately. It does not combine multiple mutations in one test run, to see if the unit tests detect problems caused by a cascaded fault. This reduces the search space exponentially, while the assumption that almost all unit test set insufficiencies can be found still holds. This is based on the Coupling Effect, a theorem stating that complex errors, represented by applying multiple mutation operators at once, are likely to be detected by a test set that detects single mutations. Offutt conducted several experiments, suggesting that this is valid for at least second order mutants, which are generated by applying 2 mutation operators at once [19].

2.2 Mutation level

There are two possible points in an applications lifecycle where mutation could be applied. Either the source code is mutated, or the byte code is mutated after the source code has been compiled. The latter lowers the time consumption of mutation testing, because the program does not have to be recompiled for every mutation. However, this makes the process of mutating more complicated for the developer of the mutation framework, possibly leading to more errors [8]. The `iinc` operator, for instance, increments local variables only, which means that while incrementing local and class variables looks identical at source code level it is not the same byte code instruction [17]. This also decreases the possibilities of debugging a certain mutant as jdb's[2] source code options do not work for byte code mutated programs. Due to performance considerations, Pitest exclusively uses byte code mutation.

Furthermore there are multiple ways to enable the mutations for testing. One is to create one class file per mutation. This is inefficient because a new JVM[3] instance has to be initialized for every mutation. Pitest uses the Java Instrumentation API [5] instead, which allows creating all mutations in primary memory, eliminating the bottleneck of writing all changes to disk as well as allowing to reuse the same JVM for each mutation within a single class, although this causes some problems as discussed in Section 4.5 [8].

To reduce the amount of tests being run on a certain mutation, Pitest initially performs a coverage analysis, and sorts the unit test execution order by runtime. During the mutation analysis, the testing phase after the mutations have been created, only those tests which execute the mutated line [9] are considered. This ensures that no time is wasted on running tests unable to detect the mutation in any case.

2.3 Computability theory based problems

Mutation testing is affected by two computation theoretical problems:

Each living mutant might be an equivalent of the original program. Detecting whether two programs are equal is undecidable in general [22]. However, there are several ways to decide this for some programs, for instance by converting them into dynamic single assignment form [21] or restricting the number of allowed differences between program to enable easier invariant generation [13]. None of these methods is integrated into Pitest, but there is a Plugin API allowing extensions to Pitest. As there is currently no such plugin, equivalent detection has to be done by a human.

Pitest might also generate mutants which do not terminate. Detecting these is again undecidable, as the halting problem is undecidable [22]. There exist heuristics to cope with this problem [7], but these are also not integrated into Pitest. Instead, Pitest uses timeouts to avoid running tests

[2]Java debugger

[3]Java virtual machine

Table 1: Comparison between Ma et al. and Pitests mutation operator sets

Ma and Offutt	Pitest	Restrictions by Pitest
Arithmetic Operator Replacement [binary]	Math Mutator	Fixed replacement per operator
Arithmetic Operator Replacement/Deletion [unary]	Invert Negatives Mutator	Only removed negation of numbers
Arithmetic Operator Replacement [short-cuts]	Increments Mutator	Stack variables only, attributes are handled by Math Mutator
Arithmetic Operator Insertion	-	
Arithmetic Operator Deletion [binary]	-	
Relational Operator Replacement	Conditionals Boundary Mutator, Negate Conditionals Mutator, Remove Conditionals Mutator	Two fixed replacements per operator
Conditional Operators	Negate Conditionals Mutator	Pitest only mutates !=
Shift Operators	-	
Logical Operator Replacement	Math Mutator	
Logical Operator Deletion	-	
Logical Operator Insertion	-	
Short-Cut Assignment Operator Replacement	Math Mutator	Fixed replacement per operator

forever. As the program passes the original test set, the developer has an estimation of how long the tests need to run and can set the timeout accordingly. Mutants violating this timeout can be considered killed, as the test set at least detected some change, even if it was not a change in output. By default Pitest considers a test as timeouted if it took 1.25 times the original test runtime plus 3 seconds [10].

3. STUDY DESIGN

3.1 Metrics

The main constraints for applying mutation testing in practice are likely to be time or low code coverage. Since byte code mutation is extremely fast, the most time will be spent running the test sets, which is mostly independent from the mutation framework and depends on the test framework. As computation time depends on the actual machine being used as well as on the test set, the ratio of living mutants over total mutants is a sensible metric. This is equivalent to the inverse mutation score:

$$\begin{aligned} MutScore &= \frac{Mut_{killed}}{Mut_{killed} + Mut_{live}} \\ \Leftrightarrow MutScore &= \frac{Mut_{total} - Mut_{live}}{Mut_{total}} \\ \Leftrightarrow 1 - MutScore &= \frac{Mut_{live}}{Mut_{total}} \end{aligned}$$

The idea behind this metric is that a high mutation score indicates a good set of unit tests and since the mutation operator set competes against the set of unit tests, a high inverse mutation score characterizes a good mutation operator set. A good mutation operator set would create a huge number of living mutants while spending a minimum time on creating and testing killed mutants. This corresponds to the “do fewer” principle introduced by Offutt and Untch [20]. Offutt and Untch also proposed the ideas of “doing smarter”, that means reducing workload by leveraging code caching, as done by Pitest’s mutation injection, and “doing faster”, which is going to be investigated by examining the possibility of scaling Pitest.

The other time factor is time spent by humans sorting out equivalent mutants. This is part of the human oracle problem [23], meaning that it is either very hard or impossible to perform this work automatically. Time spent sorting should correlate with the size of the program, as more source code allows more mutations, which will lead to more live mutants and equivalents. Instead of measuring this in absolute numbers, the ratio $\mathcal{R}_{eq}$ of equivalents over live mutants will be measured. The smaller this ratio is, the less time will be consumed by sorting out equivalents compared to the time used to improve the test set to kill more mutants.

3.2 Test subjects

As mutants can only be killed if there are corresponding tests executing the mutated instructions, mutation testing requires very high test coverage. Many Apache Commons packages have test coverage above 80%, are widely used and thus very relevant regarding practical use. One test candidate will be apache.commons.math4 [2], with 42392 NCLOC[4] in 63 packages and 581 classes. Math4 has been chosen as a test subject representing math heavy libraries. It has a test coverage of 90% according to Pitest’s internal measuring module. These 90% only include JUnit compatible tests. Math4 also has R tests, for instance used to verify probabilistic distribution functions. These cannot be automatically run by Pitest, which shows one of it’s downsides: It requires a test framework integration for all tests that should be run on the created mutants. There is a plugin interface for Pitest, allowing adding arbitrary test frameworks but requiring additional time to set up.

The other test subject will be apache.commons.io [1], representing operating system dependent libraries. It has 4966 NCLOC in 100 classes distributed over 7 packages. Test coverage is 89%, all tests are JUnit compatible.

Pitest provides per-line results on which mutations were live and which were killed. To reduce the amount of work, 6 classes have been randomly chosen to be analyzed in depth:

- org.apache.commons.math4.util.TransformerMap

[4]Non Comment Lines Of Code, lines of code without blank, header, footer, import or comment lines.

- org.apache.commons.math4.geometry.VectorFormat
- org.apache.commons.math4.linear.ArrayFieldVector
- org.apache.commons.io.output.LockableFileWriter
- org.apache.commons.io.input.XmlStreamReader
- org.apache.commons.io.FileSystemUtils

3.3 Mutation operator sets

In order to analyze the implications of different operator sets the following operator sets will be used:

Pitests default set

Conditionals Boundary Mutator weakens or strengthens relational operators

Negate Conditionals Mutator replaces conditionals by their logical counterparts

Math Mutator replaces arithmetic operations by their mathematical counterparts

Increments Mutator replaces short cut increment operators with decrement and vice versa

Invert Negatives Mutator removes unary "-" operations

Return Values Mutator replaces method return values with constants of the same type (`null` in case of non primitives)

Void Method Call Mutator removes calls to methods of void return type

And a mutation operator set containing all available Pitest mutators, except for those which are labelled as experimental by Pitest's developer:

Full set (all available mutators)

All of the above

Constructor Calls Mutator replaces constructor calls with `null`

Inline Constant Mutator replaces constants, mostly by either incrementing or inverting them

Non Void Method Calls Mutator replaces non-void method calls with constants, mainly `null` or 0

Remove Conditionals Mutator replaces conditionals with `true` and `false`

3.4 Test environment

All tests will be run on an Intel i5-2500k with 3.3 GHz clock rate, 4 threads and 8 gigabytes of RAM. The used operating system is Fedora Linux[5] with Oracle Java version 1.8.0_31. Unless explicitly specified, all settings, such as RAM configuration for JVM, are system defaults. Pitest version is 1.1.7 downloaded and integrated via Maven. Math4[6] and IO[7] were obtained from the Apache git and Subversion repositories.

[5]Kernel version 4.1.7-200.fc22.x86_64
[6]http://git-wip-us.apache.org/repos/asf/commons-math.git at commit 8ed2209b1f8e2452d71ef8c3149f3ed3d89d4dfa
[7]http://svn.apache.org/repos/asf/commons/proper/io/trunk at revision 1717147

4. TEST RESULTS

4.1 Runtime and mutation score

Table 2: Runtime and inverse mutation scores of Apache Commons Math4 and IO

Library	Mutators	Runtime	1-Score	Live mutants
Math4	default	03:09:43	20%	8297
Math4	full	09:08:46	23%	23862
IO	default	00:19:08	18%	575
IO	full	01:02:21	21%	2119

Table 2 shows a comparison of runtimes, living mutants and mutation scores for the tested libraries and mutation operator sets. For Math4 the runtime and number of living mutants scales almost linearly, however that means that for the full mutation operator set, compared to the default set, a 181% increase in runtime has led to a 14% decrease of the mutation score. That means that increasing the mutation operator set does neither significantly decrease or increase the mutation score but requires linearly more computation time. For IO scaling was slightly better, as 3 times higher computation time caused a 3.69 times higher amount of living mutants, although the mutation score did, again, not differ significantly. The assumption that the main amount of time will be consumed by running the test while generating the mutants will have a negligible time consumption has been validated: The longest mutation building phase took 5 seconds for the Math4 full test. The same holds for coverage and dependency analysis, which took 3 minutes and 51 seconds for this test and remained constant for the default mutation operator set.

Testing the IO library was almost as efficient as testing Math4. It is not clear whether the 2 percent points increase in mutation score was caused by a better test set of the IO library, by the structural differences discussed in Section 3.2 or if this is coincidental. Noteably, the 2 percent point difference was maintained for both mutation operator sets. Nevertheless, due to the diminutiveness of the difference, mutation testing seems to be an as valid tool for less mathematical programs as for mathematical libraries.

The runtime seems to linearly scale not only with the number of mutation operators but also with the number of tests and NCLOC. Increasing the number of threads improved the runtime significantly. Using 2 threads decreased the runtime of the full math4 test to 5 hours, 8 minutes and 18 seconds. Adding an additional thread decreases the runtime even further to 4 hours, 25 minutes and 51 seconds. The most probable cause for computation time not being linearly dependent on the number of threads is either a bad load balancing between the threads or other system processes competing with Pitest for resources.

4.2 Equivalent mutants

The number of equivalent mutants is generally very low, contrasting the results of Grün et al. who used a competing mutation framework, named Javalanche, to perform a mutation test on the several open source libraries and discovered that 40% of live mutants were equivalents [14]. A possible cause for this could be differences in the source code of the tested programs, with the libraries tested by Grün et al. being more likely to cause equivalent mutations. The exem-

Table 3: Number of equivalent mutants per class and operator set

Library	Class	Mutators	$\mathcal{R}_{eq}$
Math4	TransformerMap	default	1 in 5 (20%)
Math4	VectorFormat	default	0 in 8 (0%)
Math4	ArrayFieldVector	default	0 in 144 (0%)
Math4	total	default	1 in 157 (0.7%)
Math4	TransformerMap	full	1 in 36 (2.8%)
Math4	VectorFormat	full	0 in 16 (0%)
Math4	ArrayFieldVector	full	0 in 269 (0%)
Math4	total	full	1 in 321 (0.3%)
IO	LockableFileWriter	default	0 in 18 (0%)
IO	XMLStreamReader	default	1 in 12 (8.3%)
IO	FileSystemUtils	default	0 in 23 (0%)
IO	total	default	1 in 53 (1.9%)
IO	LockableFileWriter	full	0 in 53 (0%)
IO	XMLStreamReader	full	1 in 119 (0.8%)
IO	FileSystemUtils	full	0 in 100 (0%)
IO	total	full	1 in 272 (0.4%)

plary listing referenced in their paper contains at least one conditional which can be removed since it always evaluates to false, indicating that dead code will increase the number of equivalents.

Equivalents in Math4 and IO were caused either by hash functions where adding or subtracting did not make a difference or by weakening conditional boundaries such as $x > 0$ to $x >= 0$, where executing the guarded code block with $x = 0$ did not make any difference apart from a minor performance regression undetected by the test set and Pitest's timeout.

4.3 Improving the mutation operator set

As discussed above, the inverse mutation score could not be significantly increased by simply adding more mutation operators. To determine a more efficient mutation operator set in the sense of having a greater inverse mutation score, the inverse mutation scores were analyzed per mutation operator in Table 4. The bold lines indicate which operators were above the average inverse mutation score and which were below. The biggest increase of the inverse mutation score could be achieved if testing would just include the conditionals boundary mutator. However this decreases the total number of live mutations significantly as only 2098 out of 23862 (9%) live mutants in the Math4 test were caused by this mutator.

Interestingly, for both libraries, which are developed independently, the same mutators led to roughly the same ordering if ordered by inverse mutation score. This suggests that some mistakes are more commonly tested against than others, although the test data set might be too small to completely proof this. In case this would be true, generating an optimized test set, that is optimal for any program, would be possible. The meaning of optimal depends on project specific criteria. If a project requires thoroughly tested code, for instance because it is life critical, optimal most probably means that all mutation operators should be applied to get as many live mutants as possible. Other possible operator sets could include maximum efficiency, in the sense of finding a mutation operator set such that adding any additional operator would cause a runtime increase being percental bigger than the percental increase in live mutants. However runtime per operator depends on the actual project (Table 2) and thus, any time dependent optimal mutation operator set depends on the project it is applied to.

Table 4: Inverse mutation scores per mutation operator

Mutator	Math4	IO
Conditionals Boundary Mutator	42%	34%
Void Method Call Mutator	33%	39%
Inline Constant Mutator	31%	27%
Remove Conditionals Mutator	24%	22%
Constructor Call Mutator	23%	24%
NonVoid Method Call Mutator	21%	24%
Invert Negatives Mutator	18%	-
Return Values Mutator	18%	14%
Math Mutator	16%	15%
Negate Conditionals Mutator	12%	9%
Increments Mutator	10%	2%
Remove Increments Mutator	10%	2%

4.4 Applicability to real world software

Mutation testing seems to be generally viable, given the above test results. A low number of equal mutants allows to mostly ignore the equivalent mutant problem since finding an equal mutation becomes an edge case as long as the tested project is unlikely to cause a lot of equivalents because of dead code. However Pitest's very long runtimes as unit test sets become larger is a huge problem. Math4 has around 850 tests, depending on the exact configuration. Apache Solr[3], a widely used full text search server, has about 10500 unit tests and about 225000 NCLOC. If the runtime of Pitest scales linearly with the number of tests and NCLOC as the above experiments indicate, running Pitest on Solr would take around 60 times longer than the Math4 full test, which would be around 540 hours when run on a single thread, assuming it would have a 90% test coverage. Actually Solr's test coverage is lower[4], leading to another problem of mutation testing, as decreased line coverage also decreases the efficiency of mutation testing. If the line coverage of a project is far from 100% it is easier to increase it by testing the uncovered lines than it is to run mutation testing and try to increase the number of killed mutants. Additionally, the choice of mutation operator sets largely influences the runtime, requiring a prioritization of runtime, maximum amount of live mutants and efficiency on a per project basis.

4.5 Bugs in Pitest

The experiments showed several problems within Pitest. For some classes, for instance math4's HermiteRuleFactory, running the default set led to all mutants being killed, while for the full set no mutant was killed, including those mutants present in both tests. This is most probably a bug within the mutant injection, since mutants should not influence each other and thus the same mutants should be killed in both sets, disregarding which additional mutants are generated. A similar behavior has been reported previously [6] and Pitest's developer suggests to reduce the size of mutation units to one. That causes every mutation to be run in a seperate process and slows down the testing but solved the problem for this particular class.

Another problem lies within the mutator application. For no apparent reason the Invert Negatives mutator did not appear in the results of the IO-full test. Another iteration of mutation testing with verbose logging enabled did not show any reason for this behavior.

Testing the IO library also showed that it is important to make sure all dummy data gets erased between mutations. The test set of IO contains a test attempting to maliciously create a file via a library method. If this test passes, the file was not created, because the library successfully detected the misconfiguration. One mutation removed this check and the file got created. Because this file was not erased after the test run completed, all test reruns failed. This problem can be resolved by writing the unit tests in a way that they clean up dummy data after each test. It is not always sufficient to delete dummy data after the whole mutation test completed since the same unit test may be run multiple times during a mutation test.

5. CONCLUSIONS

Mutation testing seems to be generally useful in real world projects, as suggested by the conducted analysis. The number of equivalents appears to be low enough to be ignored, given a low amount of dead code, and the process is fast enough for small to medium sized projects such as utility libraries. Problems exist with bigger projects, such as Solr, as the runtime becomes too long. Also unit test sets with low line coverage need to be improved prior to mutation testing in order to increase efficiency of the latter.

However, it is likely that unit test coverage will improve in the future [12]. Meanwhile the efficiency of different mutation operators could be investigated further to determine optimized mutation operator sets, while the remaining issues with the implementations of frameworks such as Pitest could be resolved, allowing to generate more consistent test results and further improving performance. The performance analysis also suggested that increasing the number of threads is beneficial. With cloud computing consistently becoming cheaper, mutation testing on multiple nodes might lower the time consumption further by highly parallelizing work.

6. REFERENCES

[1] Apache Commons IO. `https://commons.apache.org/proper/commons-io/`. Retrieved December 2, 2015.

[2] Apache Commons Math4. `https://commons.apache.org/proper/commons-math/`. Retrieved December 2, 2015.

[3] Apache Solr. `http://lucene.apache.org/solr/`. Retrieved December 2, 2015.

[4] Clover report: Apache Lucene/Solr 6.0.0-413. `https://builds.apache.org/job/Lucene-Solr-Clover-trunk/413/clover-report/dashboard.html`. Retrieved December 2, 2015.

[5] Instrumentation (Java Platform SE 7 b99). `https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/Instrumentation.html`. Retrieved November 30, 2015.

[6] Mutation survives when it should actually get killed. `https://github.com/hcoles/pitest/issues/181`. Retrieved November 30, 2015.

[7] M. Brockschmidt, T. Ströder, C. Otto, and J. Giesl. Automated detection of non-termination and nullpointerexceptions for java bytecode. In *Formal Verification of Object-Oriented Software*, pages 123–141. Springer, 2012.

[8] H. Coles. Mutation testing systems for Java compared. `http://pitest.org/java_mutation_testing_systems/`. Retrieved November 20, 2015.

[9] H. Coles. Pitest: Basic Concepts. `http://pitest.org/quickstart/basic_concepts/`. Retrieved November 20, 2015.

[10] H. Coles. Pitest: Maven Quick Start. `http://pitest.org/quickstart/maven/`. Retrieved November 20, 2015.

[11] H. Coles. Pitest: Mutation operators. `http://pitest.org/quickstart/mutators/`. Retrieved November 20, 2015.

[12] E. Daka and G. Fraser. A survey on unit testing practices and problems. In *Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on*, pages 201–211, Nov 2014.

[13] B. Godlin and O. Strichman. Regression verification: proving the equivalence of similar programs. *Software Testing, Verification and Reliability*, 23(3):241–258, 2013.

[14] B. J. Grün, D. Schuler, and A. Zeller. The impact of equivalent mutants. In *Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on*, pages 192–199. IEEE.

[15] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on*, 37(5):649–678, 2011.

[16] K. N. King and A. J. Offutt. A fortran language system for mutation-based software testing. *Software: Practice and Experience*, 21(7):685–718, 1991.

[17] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. The Java® Virtual Machine Specification. `http://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html#jvms-6.5.iinc`. Retrieved November 30, 2015.

[18] Y.-S. Ma and A. J. Offutt. Description of Method-level Mutation Operators for Java. `https://cs.gmu.edu/~offutt/mujava/mutopsMethod.pdf`, 2005. Retrieved November 20, 2015.

[19] A. Offutt. The coupling effect: Fact or fiction. *SIGSOFT Softw. Eng. Notes*, 14(8):131–140, Nov. 1989.

[20] A. J. Offutt and R. H. Untch. Mutation 2000: Uniting the orthogonal. In *Mutation testing for the new century*, pages 34–44. Springer, 2001.

[21] K. Shashidhar, M. Bruynooghe, F. Catthoor, and G. Janssens. Verification of source code transformations by program equivalence checking. In *Compiler Construction*, pages 221–236. Springer, 2005.

[22] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.

[23] E. J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982.

An Analysis of Information Needs to Detect Test Smells

Delin Mathew
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
delin.mathew@rwth-aachen.de

Konrad Foegen
RWTH Aachen University
Software Construction
Ahornstr. 55
52074 Aachen, Germany
konrad.foegen@swc.rwth-aachen.de

ABSTRACT

Test smells are symptoms of poorly designed test cases which do not comply with the unit test criteria. The quality of test codes tends to deteriorate when it is frequently modified. High quality in terms of coverage and maintainability is necessary in order to make automated software tests effective in the long run. Refactoring is the most commonly used technique in order to achieve this. In this paper, we focus on three of the test smells that are specific for test code(s), which are, (1) General Fixture (2) Eager test and (3) Obscure Test. We identify methods to detect them and also develop a conceptual model to describe the information needed to detect these test smells in actual test cases and their dependencies with each other.

Keywords

Unit test, maintainability, refactoring.

1. INTRODUCTION

Software testing has become one of the essential elements in development cycles [1] due to the adoption of agile or lean methodologies [2] like eXtreme Programming and Test-Driven Development [3], [4], [5]. Every software system is constantly evolving and unit testing takes care that these systems do not regress. A good practice would be to perform frequent system verification by test execution[6]. Hence, it is necessary to write and maintain these unit tests continuously. These tests are written for each class in the system and usually in the same programming language as of the production code. Their objectives include checking the functionality of the system, verifying that the system is generating the expected results and in most of the cases, checking for bugs and bug fix [7].

It is important for the test cases to adhere to the unit test criteria. Some of the desired test design criteria as mentioned by Van Rompaey and Du Bois in their paper [8] are, (1) Tests should be consistent in their overall behavior (2) Every test is needed as they verify a part of the system (3) They should be maintained and should satisfy most of the object-oriented design principles (4) Tests should have repeatable outcomes (5) They should also have the expected outcome defined to check and compare the results (6) They should be isolated and should conform to a clear and rigorous structure (7) They should also be independent of external factors (8) They should be automated, persistent and run fast to reduce the turnaround time. Tests that fail to comply with any of the above criteria show symptoms of poorly designed test cases which are termed as test smells.

The research method includes extensive literature review and we also developed a conceptual model to understand the detection techniques of test smells and their dependencies with each other. This paper is structured as follows: Section 2 gives a general introduction to test smells. Section 3 gives a detailed description of the three test smells,(1) General Fixture (2) Eager test and (3) Obscure Test. It also gives a brief explanation on the maintainability and causality of each test smells and goes on to describe the techniques to identify these smells. A conceptual model which indicates the dependencies between these test smells is depicted in Section 4 and we formulate a conclusion in Section 5.

2. TEST CODE SMELLS

Several test smells have been introduced by Van Deursen in his paper [7]. He reflects upon the unit tests that are dependent on external resources, tests that take long time to run, tests that are long and complex, tests that depend on other tests for their execution, and tests which are redundant and contain undesirable duplication. This will have an adverse effect on the maintainability and readability of the tests and makes it hard to use tests as documentation. Such tests will also have non-deterministic outcomes. The presence of such smells is against the test code quality criteria and will harm the repeatability, isolation and stability of the tests [6].

Furthermore, some of the problems caused by automation of the tests has been stated by Meszaros in his book[2]. The issues commonly associated with test automation are, Behavior Sensitivity, Interface Sensitivity, Data Sensitivity, and Context Sensitivity. Behavior Sensitivity occurs when tests are broken by changes to the behavior of the System Under Test (SUT). Interface Sensitivity is caused as a result of the changes to the test programming API or the user interface used to automate the tests. When changes are made to the data already in the SUT, this results in Data Sensitivity and Context Sensitivity occurs when tests are broken by differences in the environment surrounding the SUT [2].

Meszaros has also made a clear distinction between the test smells and classified them into three main categories as code smells, behavior smells and project smells. Code smells are coding-level anti-patterns that a developer or tester may notice while reading or writing a test code. That is, the code just does not look quite right or does not communicate its intent very clearly. Behavior smells, on the other hand, are much more difficult to ignore because they cause tests to fail or in some cases not compile at all. Project smells indicate that something has gone wrong in the project. The root cause is more likely to be indicated by the presence of one or more of the code or behavior smells [2].

3. TEST SMELLS AND DETECTION TECHNIQUES

This sections gives and overview of the three test smells, which are, (1) General Fixture (2) Eager test and (3) Obscure Test. We mainly chose these test smells because they are very common, specific to test code and also act on different levels such as test case and test command.

3.1 General Fixture

3.1.1 General Description

General Fixture smell is mainly caused when the setup fixture (also known as the test context) is too general and has a broad functionality resulting in different tests accessing only certain parts of the fixture. So even if a test uses only a small part of the entire fixture, it still has to execute the whole fixture even though major part of the fixture is irrelevant for that test. This will impact the performance of the test execution which will deteriorate as the test uses a fixture that is designed to support many other tests. The problems that result could be, (1) This will cause the test to be fragile, ie. the cause-effect relationship between fixture and the expected outcomes is less visible. This results in poor readability and the test will be hard to understand. A change that is made for one test affects the other tests as too much of functionality is covered in the fixture. (2) Since they do a lot of unnecessary work, the tests run slower. This will result in the tests taking a long time to complete and thus interfering with other processes and ultimately leading the programmers to avoid executing them [7], [9], [10].

A solution to this could be refactoring the General Fixture by creating a minimal fixture which will contain only the setup code that is common to all test methods. Individual setups can be placed only in the method that uses it. Extract class refactoring can be applied in cases where the test methods do not share too much setup code. Extract class refactoring is a process of creating a new class and moving methods and data from the old class to the new one [9].

There are two kinds of General Fixtures, viz., (1) Large Fixture, which initializes many objects during setup, (2) Broad Fixture, which is a special case of Large Fixture. This fixture comprises of objects which do not logically belong together as they use different production types [8].

3.1.2 Impact on Maintenance

A test case with a large fixture requires more setup and tear down time and thus has a negative effect on the test execution time. In case of large fixtures, it is also difficult to understand the state of the unit under test and the test command executing it. There also exists a risk of modifying the General Fixture as there are many dependencies on the production types under test. These dependencies mean that the tests does not logically belong together and the units are not tested under isolation[6].

3.1.3 Causality

The presence of Large or Broad Fixture can yield the following scenarios:

- After refactoring is done and the original unit under test has been broken into different classes, it is required that the tests reflects these changes. This is critical because if test cases are not split up accordingly the fixture tends to grow [8].
- The developers usually write unit tests for groups of classes that logically belong together. This will lead to the resulting fixture containing more than one production type [8].
- The fixture tends to grow when a particular unit testing framework is used to perform other types of testing, e.g., integration testing [8].
- To verify a unit's behavior with various data configurations, a test case could contain multiple instances of the unit under test. When a bug fix is done, a test command using an additional instance can prove that the bug will not be reintroduced [11][8].

3.1.4 Detecting General Fixture

Charecterizing Metrics- Greiler and van Deursen in their paper [9] have defined two indicators for the detection of General Fixture.

- The two smell indicators to measure General Fixture in a test method, introduced by Greiler and van Deursen are (1)*setupFlds*- Fields set in implicit setups or class header and (2)*usedSetupFlds*- SetupFlds used in test methods [9]. This is a measuring technique where the ratio of both these values (indicators) are found and compared against a certain threshold value.

Four test case metrics have been introduced by Van Rompaey in his paper [8] to characterize the fixture of a unit test based on the concepts introduced by Briand et al [12].

He has defined the following two metrics for the detection of Large Fixtures:

- He defines the *Number of Fixture OBjects NFOB(tc)* as the number of attributes of a test case *tc*. This includes all the implemented as well as the inherited attributes. This is derived by considering the metrics *A(tc)* and *T(a)* where *A(tc)* is a set of Attributes of a test command *tc* and *T(a)* is the attribute type of an attribute *a* belonging to the system code [8].
- *The Number of OBject Uses in Setup NOBU(ts)* is defined as the number of method or attribute references to non-test object uses in the test setup *ts* of a test case. Both the direct and the indirect objects used by the test setup and its test helper methods have been hereby included [8].

The set of Direct OBject Uses *DOBU(ts)* from *ts* is calculated using the metrics $IM(ts)$, $IM_H(ts)$, $AR(ts)$ and $AR(TEST)$; and the set of Indirect OBject Uses $IOBU_0(ts)$ from *ts* is calculated using $IM(h)$, $IM_H(h)$, $AR(h)$, $AR(TEST)$ where *IM* is the union of set of polymorphically invoked methods and statically invoked methods, IM_H is a set of invoked test helper methods, *AR* is a set of referenced attributes, *h* refers to the helper method and *TEST* refers to the test code. In other words, *IOBU(ts)* collects the set of non-test methods and attributes invoked or referenced by helper methods of *ts* [8].

Now, $IOBU_{i+1}(ts)$ (with $i+1$ of indirection) can be determined using $IOBU_i(ts)$ and $IOBU_0(h)$. Then, finally the number of non-test object uses *NOBU(ts)* in the setup *ts* of a test case is derived as a union of *DOBU(ts)* and $IOBU_i(ts)$ [8].

Van Rompaey have characterized Broad Fixtures using the following two metrics:

- The *Number of Fixture Production Types NFPT(tc)* is defined as the number of production types that the attribute set of a test case consists of. This is derived by considering the metric *UUT(tc)* where *UUT* is the Unit Under Test [8].

- The *Number of Production Type Uses NPTU(m)* has been defined as the number of production types (1) from a test command *m* or (2) from direct or indirect test helpers called from *m* [8].

  The set of Direct Production Type Uses *DPTU(ts)* is defined using the metrics $M_I(c)$, $IM_P(ts)$, $A_I(c)$ and $AR_P(ts)$. And, the set of Indirect Production Type Uses is defined as $IPTU_0(tc)$ using the metrics $M_I(c)$, $IM_P(h)$, $A_I(c)$ and $AR_P(h)$ where M_I is the set of methods implemented in a class *c*, IM_P is the set of invoked production code methods, A_I is a set of implemented attributes in a class *c* and AR_P is the referenced production code attributes [8].

  Then $IPTU_{i+1}(tc)$ is derived as a union of $IPTU_i(tc)$ and $IPTU_0(h)$. Finally, for every test setup $ts \in TCS$ (Test Case Setup), *NPTU (ts)* is calculated using the metric $DPTU(ts)$ in union with the derived metric $IPTU_i(ts)$ [8].

Interpretation- In case of the measuring technique proposed by Greiler and van Deursen, the ratio of the number of usedSetupFlds in a test method to the number of existing setupFlds in the class is calculated. If the resulting ratio falls below a certain predefined threshold value, then the presence of General Fixture smell in the test method is concluded. In the experiments conducted by Greiler and van Deursen, they had set the threshold to 70% [9].

For the metrics proposed by Van Rompaey, assume we have a group of test cases and also their respective values which are necessary to obtain the metrics. Inorder to find out the test cases where Gerneral Fixture smell could occur, we can start off by distinguishing between those test cases that have an explicit setup method and those that do not have. For those cases which have an explicit setup method, we can check if this method have a complex initialization. If the *NOBU* value for a test case is high, then it is a clear indication of a complex fixture which contains multiple objects of production as well as library types[8].

The *NFOB* and *NFPT* metrics are checked in the case of test cases without an explicit setup method to detect a large fixture. If the value for *NFOB* is high, then this is a clear sign of a large fixture. And, a Broad Fixture is indicated if the Large Fixture has a high *NFPT* value [8].

3.2 Eager Test

3.2.1 General Description

This smell is caused when a test tries to check several methods of the object to be tested. This will reduce the readability which makes the test hard to understand and also harder to be used as documentation because the test tries to check too many functionalities in a single method. This will also affect the maintainability as the tests are more dependent on each other [7].

The solution is to split the test code into several methods that test only one method at a time, using a meaningful name which indicates the purpose of the test method. But separating the test code into smaller methods can slow down the tests due to increased setup or tear down overhead [7].

3.2.2 Impact on Maintenance

- The complex body of the Eager Test method makes it incomprehensible and hence hard to read [6].
- Since it is not very clear as to which method a test command is checking, other tests that cover the same production type will face a risk of being introduced [6].
- There could exists interdependencies between cycles as a consequence of sequence of stimulate-verify subcycles in the Eager Test Command [6].

3.2.3 Causality

Eager Test can yield the following scenarios:

- When a test command is not refactored as and when the production code is refactored, it tends to grow [6].
- Eager test appears when all the steps in a scenario-based testing approach that are applied to unit testing are executed in one test command.[6]
- A production method that contains multiple parameter value combinations can also result in Eager Test [6].

3.2.4 Detecting Eager Test

Characterizing Metrics- Van Rompaey has defined the following metric in his paper [8] based on the definitions given by Briand et al [12].

The *Production Type Method Invocations PTMI(tm)* is defined as the number of invocations to methods that belong to production types from a test command.

$PTMI(tm)$ is derived using the metrics, $NSI(tm, mc)$, $NPI(tm, mc)$ and *PTU* where *PTU* is calculated as the union of the metrics $DPTU(tm)$ and $IPTU_i(tm)$ (see section 3.1.4) and *NSI* is the number of static invocations of *mc* (method calls) originating in the body of *tm* (test method) and *NPI* is the number of polymorphic invocations of *mc* originating in the body of *tm*.

Furthermore the previously introduced NPTU is computed for every test command [8].

Interpretation- The value of the metric *PTMI* is checked to detect Eager Test. A high value of *PTMI* in test commands is a sign of Eager Test smell[8].

3.3 Obscure Test

3.3.1 General Description

Obscure Test is caused when a test is difficult to understand at a glance and thus is not suitable for documentation purposes. Automated tests should satisfy at least two objectives. Firstly, they should serve as a documentation of how a SUT must behave and this is generally called Test as Documentation and, secondly, there should be a self-verifying executable specification. These two purposes often contradict each other because the level of detail needed for tests to be executable may make the test so verbose as to be difficult to understand [2]. An in-line setup should only consists of the steps and values to understand the test. Other irrelevant steps which might be essential in the long run should be included in the helper methods. This can obstruct the developers from seeing relevant verification steps of the test [9].

One of the main solutions is to refactor the test code by moving the setup code into delegate setup methods or, in case the in-line setup is common to all tests, an implicit setup could be used [9].

3.3.2 Impact on Maintenance

- Since the test is hard to understand, it will also be difficult to maintain. This will lead to high test maintenance costs as it will take a lot of effort to for the documentation of such a test [2].

- Another issue is that it may allow bugs to slip through because of test coding errors hidden in the Obscure Test. This can result in buggy tests. Moreover, failure of one declaration may lead to many more errors, leading to loss of test debugging data [2].

3.3.3 Causality

An Obscure Test can be caused due to too much of information or could also be the result of too less information in a test case. Mystery Guest (explained below) is an example of too little information and Eager Test is an example of too much of information [2].

The root cause of Obscure Test is lack of effort in keeping the test code as clean and simple as possible. The main path to achieve this is by refactoring the test code frequently because a test code is as important as the production code. A prime cause of an Obscure Test is putting the code in-line. This results in large and complex test methods [2].

Some of the major causes of Obscure test as listed by Meszaros in his book [2] are:

- Eager Test: The test verifies too much functionality in a single test method which causes the test to be hard to understand and therefore difficult to document [2].

- Mystery Guest: The user is not able to see the cause and effect between fixture and verification logic because part of it is done outside the test method [2]. For example, when a test is dependent on an external resource like a file containing test data [7].

- General Fixture: The test references a larger fixture which is too general than is needed to verify the functionality in question and diffrent tests only access parts of the fixture [2].

- Irrelevant Information: The test distracts the user by containing a lot of irrelevant details about the fixture and the user is not able to focus on what really affects the behavior of SUT [2].

- Hard-Coded Test Data: This is caused due to the embedding or hard-coding of data values in the fixture of the SUT into the test method. This obscures cause-effect relationships between inputs and expected outputs [2].

- Indirect Testing: The test method makes the interactions more complex by using another object to indirectly interact with the SUT [2].

3.3.4 Detecting Obscure Test

Characterizing Metric- Greiler and van Deursen in their paper [9], have introduced the indicator *LocalVars* to detect Obscure Test. *LocalVars* refers to the variables declared in a test method. The obscurity of an in-line setup is measured based on the number of local variables (*LocalVars*) directly defined within a test method.

Interpretation-An obscure in-line setup is detected if the number of *LocalVars* exceeds a certain threshold. In the experiments conducted by Greiler and van Deursen, they had set the threshold to 10 variables per method. This threshold was chosen as it follows the best practices for the length of a method. Greiler and van Deursen concluded that with the increasing length of the test method, the primary focus of the test may be hidden [9].

4. CONCEPTUAL MODEL

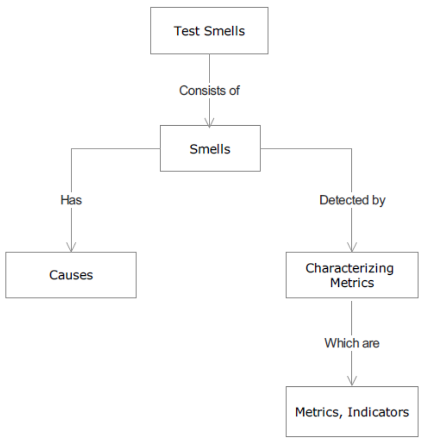


Figure 1: Meta Model

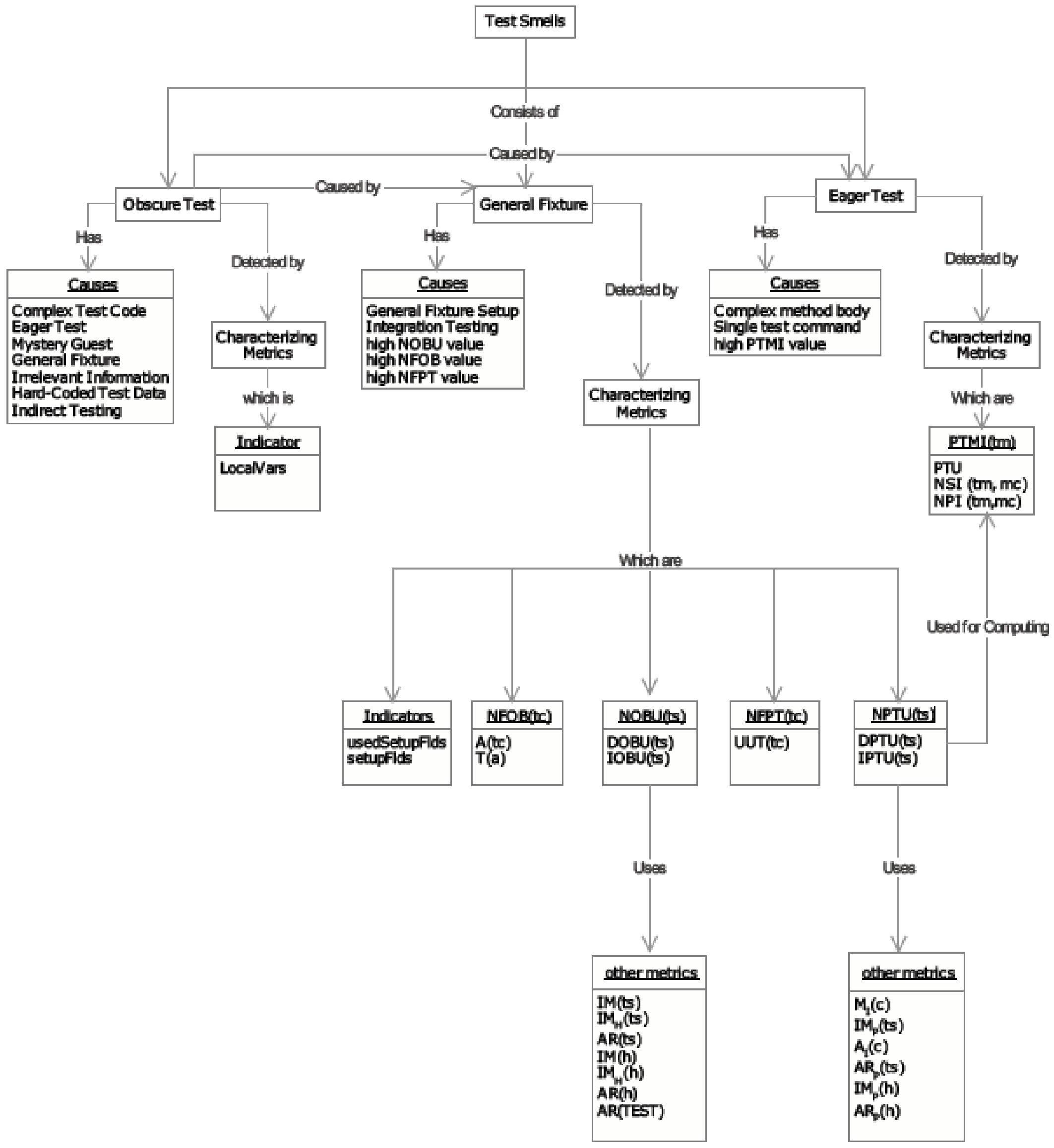


Figure 2: Conceptual Model

Figure 1 gives the meta model on which our conceptual model is constructed upon. This model is relatively similar to an Entity Relationship model. Here, the entities are written inside the rectangular boxes and the relationships between the entities are mentioned along the links between them. The attributes (if any) of each entity are specified within the boxes.

The model shows Test Smells as the main entity, which consists of three smells and their respective causes and characterizing metrics based on which they are detected. Since we are mainly focusing on the detection techniques and the causality of test smells in this paper, we have structured this model such that each test smell is characterized by its respective causes and detection techniques.

Figure 2 gives the conceptual model which is constructed based on the meta model shown above. We focus mainly on the causes and detection techniques of test smells in this model. Various causes of each smell are listed based on the findings in this paper. Under the entity Characterizing Metrics, the different metrics and indicators including the various components under them which help in the test smell detection have been depicted. And under the metrics *NOBU(ts)* and *NPTU(ts)*, the other metrics which are used to calculate these metrics are specified. In addition, we have also shown the dependencies between the test smells. It is interesting to find out that Obscure Test is caused due to the presence of General Fixture and Eager Test in the test code. Another dependency is, we compute the previously introduced $NPTU$ in General Fixture, for every test command in Eager Test. So, there is a dependency of the metric PTU that is used in the Eager Test with the metrics in General Fixture such as $DPTU$ and $IPTU$ (which are used for the calculation of $NPTU$) for the calculation of PTU.

5. CONCLUSION AND FUTURE WORK

In this paper, our main goal was to focus on the different detection techniques and to develop a conceptual model for the three test smells, (1) General Fixture (2) Eager Test and (3) Obscure Test. We have briefly explained the impact of these test smells on the maintainability and also their consequences on the SUT. We have compiled a number of metrics and indicators that measure certain properties of the test code. The metrics allow us to assess the relative significance of the test smells in the test code according to the violation of unit test criteria [8]. The indicators specified for General Fixture and Obscure Test measure the smell and check it with a pre-defined threshold value [9]. Finally, we have developed a conceptual model which depicts the various detection techniques of General Fixtures, Eager Tests and Obscure Tests and also the dependencies between them.

Future work is to find out more detection techniques for each smell. It is also possible to include more test smells that are similar to the test smells mentioned in this paper and develop a bigger conceptual model which could throw light to many more dependencies among the smells.

6. REFERENCES

[1] P. Runeson and J. Felsing. *A Practical Guide to Feature-Driven Development.* Prentice Hall, Feb 2002.

[2] G. Meszaros. *XUnit Test Patterns: Refactoring Test Code.* Addison-Wesley, May 2007.

[3] K. Beck. *Extreme Programming Explained: Embrace Change.* Addison-Wesley, 1999.

[4] K. Beck. *Test Driven Development by Example.* Addison-Wesley, 2002.

[5] S. Palmer and J. Felsing. *A Practical Guide to Feature-Driven Development.* Prentice Hall, Feb 2002.

[6] Bart van Rompaey, Bart Du Bois, Serge Demeyer, and MatthiasReiger. "On the Detection of Test Smells: A Metrics-Based Approach for General Fixture and Eager Test". *IEEE Trans. Softw. Eng.,* 33(12):800-817, December 2007.

[7] A. van Deursen, L. Moonen, A. van den Bergh, and G. Kok. "Refactoring Test Code". *Proc. Second Int'l Conf. Extreme Programming and Flexible Processes in Software Eng.,* M. Marchesi and G. Succi, eds., 2001.

[8] Bart van Rompaey, Bart Du Bois, and Serge Demeyer. "Characterizing the Relative Significance of a Test Smell". *In Proceedings of the 22nd IEEE International Conference on Software Maintenance.* ICSM '06, pages 391-400, Washington, DC, USA, 2006, IEEE Computer Society.

[9] M. Greiler, A. van Deursen, and M-A Storey. "Automated Detection of Test Fixture Strategies and Smells". *In Proc. Of the Int'l Conf. on Software Testing, Verification and Validation (ICST).* IEEE CS, 2013.

[10] M. Greiler, A. Zaidman, and A. van Deursen. "Strategies for Avoiding Text Fixture Smells during Software Evolution". *In Proc. Of the Int'l Conf. on Mining Software Repositories (MST).* 10th IEEE Working Conference, 2013.

[11] D. E. DeLano and L. Rising, Patterns for System Testing, In R. Martin, D. Riehle, and F. Buschmann, editors. *Pattern Languages of Program Design 3.* Pages 503-527. Addison-Wesley, 1998.

[12] L. Briand, J. Daly, and J. Wuest. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering,* 25(1):91-121, 1999.

Changes in Requirements Engineering After Migrating to the Software as a Service Model

Johannes Schäfer
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
johannes.schaefer2@rwth-aachen.de

Horst Lichter
RWTH Aachen University
Software Construction
Ahornstr. 55
52074 Aachen, Germany
lichter@swc.rwth-aachen.de

ABSTRACT

Service-oriented architectures are widely considered to be the determining trend in software engineering. Vendors of software products want to benefit by migrating to cloud environments. However, when transforming an existing software system from the *Software as a Product* model to the *Software as a Service* model the software engineering process changes. While the process in general has been researched sufficiently, very low effort has been put into understanding the impact on requirements elicitation.

This paper investigates the necessary changes in the requirements engineering process and provides a systematic approach for a successful transformation. Furthermore, it discusses the new benefits in requirements elicitation that are inherent in a cloud environment. The paper then discusses the identified problems and developed solutions with regards to deduced guidelines and best practices.

We conclude that the requirements engineering process profits from a systematic transformation when migrating a traditional software product to the *Software as a Service* model.

Keywords

Software Engineering, Requirements Engineering, Software as a Service (SaaS), Cloud Environment, Reengineering

1. INTRODUCTION

Studies show that 20% of the IT companies consider using Software as a Service (SaaS) as important or very important. For the majority of the IT specialists the topic is of average importance or lower [21]. Nevertheless, this is due to reservations regarding security (75%), performance and availability (63%) and integration with existing systems (61%), as these companies describe [21]. Another study points out that hiring a software instead of purchasing yields in a saving of 45% of the customer's expenses in a three year time span [17].

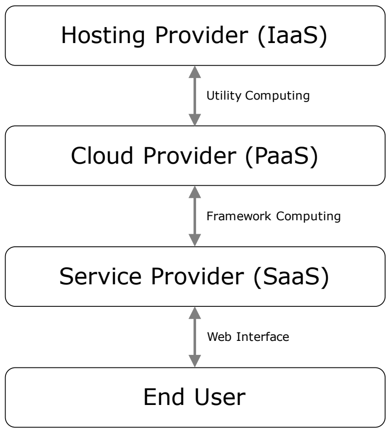


Figure 1: The cloud computing model [1] [11] [18]

SaaS is an element of the Internet-based computing model *Cloud Computing*. A cloud computing environment is essentially characterized by on-demand self-service, broad network access, resource pooling (using multi-tenancy), rapid elasticity and measured service according to the National Institute of Standards and Technology (NIST) [11].

To complete the cloud infrastructure two further elements – besides SaaS – have been identified by the NIST. The provision of runtime environments, libraries, other services and software tools by a certain provider is called Platform as a Service (PaaS). The customer of a PaaS has hardly any management control over the underlying platform components, but full control over the deployed applications [11]. Another step away from the end user is the Infrastructure as a Service (IaaS), which completes the cloud computing model. The IaaS provider is accountable for storage and network facilities and other hardware components, while the customer can install and run arbitrary software, including even operating systems [11].

Following the SaaS model, both the software system itself and the user data are hosted and stored centrally. Instead of purchasing a product, the user rents a software system, IT infrastructure and annexed services from the vendor and is typically charged on a pay-per-use principle [10]. A SaaS-based software system, however, differs from one that is developed under the Software as a Product (SaaP) model in many ways. Its architecture is mainly database-oriented, middleware-oriented, PaaS-based and service-oriented (see

Figure 1) [18]. This results in differing non-functional requirements compared to classical software products [17]. When transforming a software product into a software service, the vendor has to consider these changes in architecture and requirements [4] and in the whole software development process [7] [13].

In this paper, we collect differences in software requirements between the two models. For this purpose, we have conducted a review of pertinent literature. We also studied research on existing software migration processes and developed conclusions on how to consider variations in requirements in such process changes. The aim of the present work is to provide a generic approach that helps those software developers who want to migrate their software product to the SaaS model.

Section 2 covers the background information auxiliary for a comprehension of the SaaS model and the concomitant changes in software requirements. Section 3 outlines the work related to the requirements engineering process in a SaaS environment. In Section 4 the necessary changes in a requirements engineering process are presented and such a transformation is systemized. Section 5 discusses the developed process by means of deduced guidelines and best practices. In Section 6 limitations of the proceeding are discussed and Section 7 draws conclusions and provides future work.

2. BACKGROUND

2.1 Software as a Service

For many years, software has been produced in a supply-side oriented manner [17]. A software vendor puts effort into the requirements elicitation for a certain problem, develops and tests the software and releases the final product to the market. The customer or the software vendor's support team installs a copy of the software product at the customer's infrastructure after purchasing a licence. While minor software updates are usually conducted via an Internet interface and included in the one-time price, major upgrades often require buying a new software product [3].

The SaaS model, in comparison, is *the* trend in software engineering of the 21st century that challenges this traditional model [1]. The customer of a SaaS-based software purchases a usage right for a certain time span. In return, the vendor grants access to the online service, often combined with an individual number of accesses depending on the customer's price plan.

Since its first mentions in research in the 2000s, SaaS has gained more and more attention both from scientific and production points of view. While different approaches – such as iterative and incremental development processes and modular software products – have been established to address the issues of developing and deploying more complex software products, the SaaS model is a radical shift of the means by which software is engineered.

Providing Software as a Service in contrast to a product, at a first glance, is a manner of distribution policy affecting business issues like time to market, customer involvement and release cycles [12]. The service-orientation of software, however, also comes with major paradigm changes regarding the software development. The SaaS model utilizes services as the rudimentary factor for organizing the complexity of software [13]. The underlying principle of software design

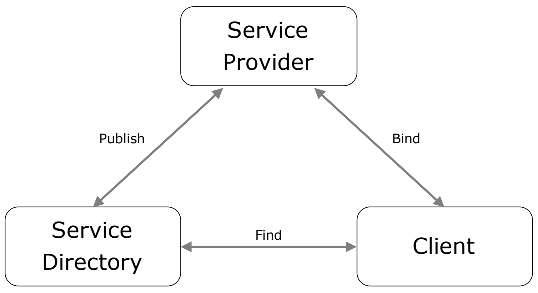


Figure 2: The service-oriented architecture model

is Service-oriented architecture (SOA), an architecture in which loosely coupled but strictly separated software components (usually single business functions) interact via public interfaces as composite services. This allows for binding components only when they are needed and in a scalable way. SOA itself is platform-agnostic and does not define the manner of service orchestration, security etc [13].

These services are made available by service providers that come up with the service infrastructure and the implementation and provide the interface description for access over the Internet (*web-based*). In order to publish and find integration-ready services, a common service directory is needed (see Figure 2). Services itself are composed of other services recursively [3].

2.2 Changed Requirements

Compared to the traditional SaaP model, SaaS relies on a different infrastructure and varies in distribution and access (see Section 2.1). When migrating a software product to the SaaS model, one usually intends to maintain most of the software's functionality [9]. As a result, the differences in the software engineering process narrow down to non-functional requirements [2] and other aspects affecting the software development process like operation, management and architecture [17], albeit not functional requirements.

Those aforementioned differences in non-functional requirements are basically due to three factors:

1. SaaS-based software is necessarily hosted in cloud environments either operated by the software vendor itself or by a third party offering PaaS solutions (see Section 1). A few very large companies offering software services unify the PaaS part and the SaaS part under a single roof, such as the on-demand video streaming platform Netflix. These companies act as platform providers for themselves.

2. Such software is primarily distributed as a web-based application [14] [19] [20] using the Internet and associated protocols for data transmission.

3. A high proportion of software offered as a service is realized as browser-supported applications [14] [20], meaning that no dedicated software is necessary on the client's device except the already existing web browser.

Factor 1 results in a focus of the non-functional requirements on security, data confidentiality, privacy and compliance, since the server location determines legal aspects such as data protection laws and a company's compliance regulations [7] [17]. In addition, a cloud service provider is a more probable victim of security attacks than a decentralized structure or a company's private network. Albeit it is harder to conduct successful attacks on professional cloud service providers, special precautions have to be considered.

Most differences in non-functional requirements are a consequence of Factor 2: Multi-tenancy, user concurrency, configurability, scalability, reliability, performance, availability, compatibility, interoperability, portability, efficiency and immediacy [5] [7] [8] [16] [17]. Other aspects include continuous evolution, the involvement of a higher number of stakeholders and increased usage monitoring [7].

Special demands on aesthetics and user interface design and the limitations of browser-supported applications are influenced by Factor 3.

3. RELATED WORK

Back in 2000, Bennett et. al [3] have recognized trends in software development that are influenced by the emerging Internet. They develop a future vision in which software is flexible, interactive, personalized and self-adapting and the software engineering is demand-led, service-oriented and focusses on the requirements elicitation. In their conclusion the authors point out that future work tshould focus on the necessary changes in the software engineering processes.

Papazoglou [13] has published seminal work on the basic concepts behind SOA. As one of the first authors he described the effects of SaaS on business processes and on software engineering. Papazoglou concludes that the SOA requires strong alterations in software design.

Olsen, the author of [12], has investigated necessary paradigm changes from a business point of view. He outlines that a SaaS-based software system creates a very different customer relationship than a SaaP-based. Olsen makes the update mechanisms responsible as they require long-term committment of the vendor and facilitate non-disruptive upgrades. The author also points out the advantages of modularity and rapid releases for the customers.

In their study [1] Armbrust et al. present definitions for the different aspects of the topic *cloud computing*. They locate the role of SaaS and list benefits as well as obstacles and demonstrate means of how to avoid them.

Since these seminal works, research has made a lot of progress. In their study [7] Kumar and Sangwan present traditional software engineering process models and main concepts (e.g. iterative development). They continue collecting aspects which make the development of web-based applications different from traditional software. According to the authors the main aspect is the continuity of the process that also requires a systematic, repeatable and iterative process. Together with lists of attributes and characteristics of web-based applications they provide a very general adaption of a traditional software engineering process model towards a model which is suitable for web-based applications. However, the authors fail to present a detailed process as a result that can be used for developing such applications.

Nogueira da Silva and Lucrédio [15] have also conducted an extensive literature review. They found out that the research interest has incresead over the last years. They identify the main challenge for cloud-based software engineering to be the lack of standardization. E.g. choosing a PaaS-provider may result in platform lock-ins where customers cannot easily switch to another service provider. Besides a grouping and the presentation of challenges for SaaS developers, the authors provide definitions of the terms SaaS and SOA. They conclude that a research gap exists regarding the formalization of a complete reengineering process in terms of reconstructing the software for a new platform.

Balian and Kumar [2] group and review studies in the field of SaaS development. They introduce literature which focusses on development from scratch as well as studies for migration and reengineering. Furthermore, the authors discuss research on quality models for SaaS and draw the conclusion that the adaption of software engineering process models, quality models and metrics for SaaS is not sufficient.

The most relevant recent works have been conducted in the field of comparing the software engineering processes of the SaaP model and the SaaS model. Tariq et al. [17] address the impact a cloud environment has on the requirements of an application. They list technical non-functional requirements, legal concerns and other issues from the data management. The authors then categorize these topics and identify the new stakeholder *cloud service provider*. As a result, they propose an addition to the Capability Maturity Model Integration (CMMI) reference model that includes a checklist for the new stakeholders.

Research has provided detailed descriptions of the SaaP model and the fundamentals behind the SaaS model. Recent work also exists which covers the transformation of a service-oriented system into a cloud-based software that follows the SaaS model [4] [5] [9] [14] [20]. However, there is no process support for migrating an existing software product into such a service-based software. Furthermore, we could not find any migration strategies that cover the differences in the requirements elicitation process. This paper intends to fill this gap by providing a systematic and generic approach for sustainably migrating a traditional software product to the SaaS model. This approach covers the software adaptions as well as the necessary changes in the existing software engineering process in a clear and repeatable way with focus on the changed requirements elicitation.

4. REQUIREMENTS ENGINEERING PROCESS FOR SAAS

4.1 Differences Between Processes

This section strictly focusses on the requirements engineering process. However, some aspects affect different phases of the software engineering process as well and others are as a matter of fact just side issues from requirements' point of view. Nevertheless, all aspects are included in the enumeration in order to provide a holistic view of the differences between the traditional and the SaaS-based requirements engineering process. This understanding of the requirements and their importance is crucial for the general software development process [17].

First of all, in comparison with the SaaP model, SaaS involves more kinds of stakeholders. Kumar and Sangwan [7] identify those as analysts, graphic designers, customers, marketing, security experts etc.

But the requirements engineering process not only has an expanded stakeholder basis. As mentioned in Section 2.2, SaaS comes with a stronger customer involvement and long-term relationships between the SaaS provider and the end user [12]. The user is motivated to provide feedback – di-

Table 1: Differences in requirements engineering between SaaP and SaaS

Software as a Product	**Software as a Service**
Overseeable number of stakeholders	Many different stakeholders
Little or no customer involvement	High customer involvement
Customer relationship on a per-version basis	Long-term customer relationship
User feedback only via special surveys	Direct user feedback motivated through regular updates and via usage monitoring
Bug fixes on a scheduled basis	Bug fixes immediately
No feature enhancements without upgrade	Continuous rollout of feature enhancements without time delay
Updates and upgrades usually require downtime	Seamless update integration process
Upgrades have high impact and often require retraining	Continuous update process causes less disruptive changes
Difficult to test and predict acceptance	Two or three variations tested on small user groups simoultanously

rectly or indirectly via usage monitoring –, since a feature enhancement can be expected and he/she will profit from it without extra cost and in foreseeable time. The integration of bug fixes and new features is seamless and without interruptions because the software is centrally hosted on the company's servers instead of on the customer's infrastructure. They are furthermore integrated without time delay since time to market is reduced significantly due to the fact that new versions are released early and often and are not considered to be a distinct software product [17].

The difference between enhancements and bug fixes becomes indistinguishable to the end user [12]. These less disruptive updates, which respectively approach just a few problems but in return happen more frequently, require fewer amount of retraining on the end user's side [12].

Moreover, the centrally hosted, multi-tenant software as a service offers additional opportunities in testing new features. The acceptance can be evaluated by rolling out the feature to just a selected proportion of users and awaiting their feedback. Even providing two or three variations of a feature to several user groups is possible and allows for comparing differences and selecting the implementation with the highest user approval.

The described characteristics together lead to and require a process with continuous development, frequent modifications and feedback [7]. This means shifting from a user-centered requirements engineering process to a service-centered process [17] that is at best iterative and parallel to the productive use of the software. At the same time, the requirements engineering and the development in general focus on only one stable version and do not require support for legacy software [12], which saves development resources.

Altogether, this is no less than a paradigm change in requirements engineering compared to the traditional SaaP model. Table 1 contrasts the differences of the requirements engineering processes mentioned beforehand.

4.2 Systematic Transformation

Requirements engineering is an important part of the software engineering process (see Figure 3). The aim is to elicit, structure, prioritize and coordinate the particular requirements of the software system, respectively of a single feature of the software. When migrating a software product to the SaaS model, the requirements engineering process has to be transformed in order to effectively accompany the new software service, as was mentioned above.

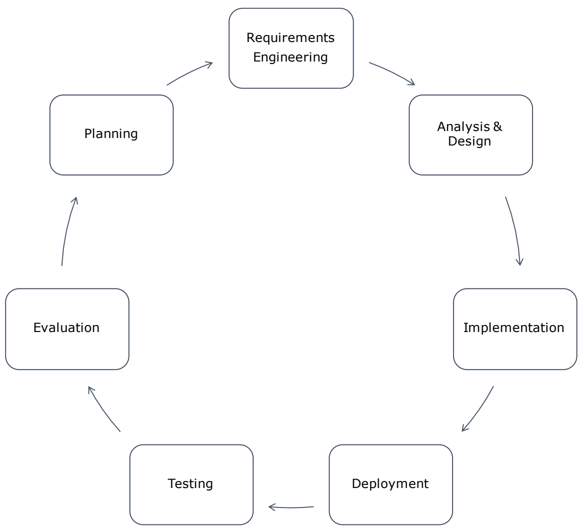


Figure 3: The iterative software engineering process

A systematic transformation of the requirements engineering process follows these steps:

Step 1: Establish a paradigm change with respect to highly fluctuating requirements. Developers who are used to traditional software products need to adapt to the non-persistence of requirements in the SaaS context. The vicinity to agile development and the new methods of elicitation make the requirements volatile.

Step 2: Integrate requirements engineering into an iterative and incremental software engineering process. Such a software engineering process is not a unique characteristic of service-based software and can be found in traditional software development as well. However, the volatile nature of the requirements and the frequent release cycles demand such iterations and regular software increments.

Step 3: Identify and prioritize stakeholders using systematic methods.

Step 4: Involve customers through integration into the requirements engineering process. Invitations for feature requests and bug reports are crucial for taking advantage of the migration to SaaS. The users need to get the feeling that their involvement can have an impact on future feature enhancements and short-term bug fixes.

Step 5: Implement instruments for user feedback (e.g. usage monitoring, feedback forms). In order to encourage customers to provide feedback (see Step 4), such a culture needs to be established. This can be achieved by e.g. providing feedback buttons on single features, offering side-wide available feedback forms and by using the many ways of usage monitoring offered by cloud software.

Step 6: Develop mechanisms for seamless update integrations. As stated before, a major benefit of cloud-hosted software is the deployment in the hand of the software developers. Thus, the integration of updates comes handy: The new software pieces only need to be installed once and on a predictable server environment – the companies cloud server – and not the client's infrastructure. In addition, the high frequency of *small* updates makes it easy to integrate without shutdown times, since the number of lines of code or the changes in the database design are proportionally smaller. The short downtime of parts of the system is less noticeable than a traditional maintenance downtime of the whole system.

Step 7: Develop support for software variations per user group for acceptance testing reasons. The new requirements engineering process makes it possible to develop multiple versions of unknown acceptance and roll out the variations to different user groups. Acceptance can then be tested using the methods of Step 5.

5. DISCUSSION AND BEST PRACTICES

The main goal of this paper was to outline the differences between the requirements engineering process of a traditional software product and the process of a software service and to provide a systematic approach for migrating from one to the other.

Following the presented approach reduces the risk for leaving out necessary changes in the requirements engineering process. For those who consider migrating a software product but have not decided yet, the approach defines the scope of changes which would be intrinsic to a planned migration. As such this paper's approach offers benefits that can not be found in literature as of today.

In order to accomodate the transformation approach we provide a collection of best practices, which came across during literature review, for some of the steps:

Step 2 is suited best by applying agile software development methods, such as Scrum. The stakeholder analysis of Step 3 is well conducted when using socio-digrams or power-matrices. The authors of [6] provide an extensive description of their process of stakeholder identification and impact analysis. The measurements already mentioned in Step 5 for motivating users to provide feedback have been successfully conducted in practice and can be recommended. That is implementing application-wide feedback forms and applying usage monitoring.

6. LIMITATIONS

This approach covers the requirements engineering process, which is only one part of others in the whole software development process. The migration of software products to the cloud can still fail due to other implications of such a process migration.

Another limitation of this approach is the focus on web-based service-oriented architectures. This circumstance is owed to the experiences from the literature review. Most research does not differentiate between SaaS and web-based systems, which makes the development of a transformation approach generalized for other kinds of customer interface nearly impossible.

7. CONCLUSION AND FUTURE WORK

The way we practice software engineering has changed dramatically. Developing SaaS is one of the reasons why changes in software engineering processes are indispensable. The requirements elicitation of a software realized as a service differs to the traditional product in many ways, some of them are fundamental (see Section 2.2). However, the differences come with numerous advantages, such as long-term customer relationships, focus of resources and more frequent feature enhancements. The requirements engineering process requires transformation when migrating from an existing software product to the SaaS model. This paper has offered a systematic approach for this transformation that can be used by software developers who want to adapt the way they determine and meet the requirements of their software system.

Future work is to research on how to combine the benefits of this new requirements elicitation methods with agile software engineering processes that already focus on iterative and incremental development.

8. REFERENCES

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[2] N. Baliyan and S. Kumar. Towards software engineering paradigm for software as a service. In *Contemporary Computing (IC3), 2014 Seventh International Conference on*, pages 329–333. IEEE, 2014.

[3] K. Bennett, P. Layzell, D. Budgen, P. Brereton, L. Macaulay, and M. Munro. Service-based software: The future for flexible software. In *Software Engineering Conference, 2000. APSEC 2000. Proceedings. Seventh Asia-Pacific*, pages 214–221. IEEE, 2000.

[4] M. A. Chauhan and M. A. Babar. Migrating service-oriented system to cloud computing: An experience report. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 404–411. IEEE, 2011.

[5] M. A. Chauhan and M. A. Babar. Towards process support for migrating applications to cloud computing. In *Cloud and Service Computing (CSC), 2012 International Conference on*, pages 80–87. IEEE, 2012.

[6] A. Khajeh-Hosseini, D. Greenwood, and I. Sommerville. Cloud migration: A case study of

migrating an enterprise it system to iaas. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 450–457. IEEE, 2010.

[7] S. Kumar and S. Sangwan. Adapting the software engineering process to web engineering process. *International Journal of Computing and Business Research*, 2(1), 2011.

[8] J. Y. Lee, J. W. Lee, D. W. Cheun, and S. D. Kim. A quality model for evaluating software-as-a-service in cloud computing. In *Software Engineering Research, Management and Applications, 2009. SERA'09. 7th ACIS International Conference on*, pages 261–266. IEEE, 2009.

[9] G. Lewis, E. Morris, and D. Smith. Service-oriented migration and reuse technique (smart). In *Software Technology and Engineering Practice, 2005. 13th IEEE International Workshop on*, pages 222–229. IEEE, 2005.

[10] D. Ma. The business model of software-as-a-service. In *Services Computing, 2007. SCC 2007. IEEE International Conference on*, pages 701–702. IEEE, 2007.

[11] P. Mell and T. Grance. The nist definition of cloud computing. *Special Publication 800-145*, 2011.

[12] E. R. Olsen. Transitioning to software as a service: Realigning software engineering practices with the new business model. In *Service Operations and Logistics, and Informatics, 2006. SOLI'06. IEEE International Conference on*, pages 266–271. IEEE, 2006.

[13] M. P. Papazoglou. Service-oriented computing: Concepts, characteristics and directions. In *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on*, pages 3–12. IEEE, 2003.

[14] E. Saleh. Migrating traditional web applications into multi-tenant saas. *Proc. 6th Ph. D. Retreat of the HPI Research School Serviceoriented Syst. Eng*, pages 145–155, 2013.

[15] E. A. N. d. Silva and D. Lucrédio. Software engineering for the cloud: A research roadmap. In *Software Engineering (SBES), 2012 26th Brazilian Symposium on*, pages 71–80. IEEE, 2012.

[16] J. Song, Z. Yan, F. Han, Y. Bao, and Z. Zhu. Introducing saas capabilities to existing web-based applications automatically. In *Web Technologies and Applications*, pages 560–569. Springer, 2012.

[17] A. Tariq, S. A. Khan, and S. Iftikhar. Requirements engineering process for software-as-a-service (saas) cloud environment. In *Emerging Technologies (ICET), 2014 International Conference on*, pages 13–18. IEEE, 2014.

[18] Q. Zhang, L. Cheng, and R. Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications*, 1(1):7–18, 2010.

[19] W. Zhang, A. J. Berre, D. Roman, and H. A. Huru. Migrating legacy applications to the service cloud. In *14th Conference companion on Object Oriented Programming Systems Languages and Applications (OOPSLA 2009)*, pages 59–68, 2009.

[20] X. Zhang, B. Shen, X. Tang, and W. Chen. From isolated tenancy hosted application to multi-tenancy: Toward a systematic migration method for web application. In *Software Engineering and Service Sciences (ICSESS), 2010 IEEE International Conference on*, pages 209–212. IEEE, 2010.

[21] M. Zhou, R. Zhang, W. Xie, W. Qian, and A. Zhou. Security and privacy in cloud computing: A survey. In *Semantics Knowledge and Grid (SKG), 2010 Sixth International Conference on*, pages 105–112. IEEE, 2010.

An Overview on Automated Test Data Generation

Concepts + Tool Support

Junior Lekane Nimpa
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
junior.lekane.nimpa@rwth-aachen.de

Horst Lichter
RWTH Aachen University
Software Construction
Ahornstr. 55
52074 Aachen, Germany
lichter@swc.rwth-aachen.de

ABSTRACT

Test automation is a means of reducing the time and cost spent during software testing. It includes the automatic generation of test data which is an interesting, active and a vast area of research in software engineering. There is a large number of specific publications introducing new algorithms and concepts without linking them to the existing ones. In addition, some publications use expressions which create some confusion. As an example, the expressions "test case generation" and "test data generation" are often used interchangeably although they denote different problems. All that makes it hard for students and researchers interested in this area to get a solid understanding of what generating test data is about.

In this article, we first introduce the test case and test data generation problems. Thereafter, we review some basic notions like constraint-based test data generation, symbolic and concrete execution. Next, we focus on the state of the art techniques of test data generation which are grouped under the term search-based test data generation. We also discuss some known limitations of test data generators. In the second part of the article we present an automated black-box test data generation technique for web services which is based on Design by Contract and mutation testing. Thereafter, we present some tools which can be used by software testers to enrich their daily test automation experience. Finally,we conclude this article by giving our opinion on the subject.

Categories and Subject Descriptors

D.1 [**Software Testing**]: Automated Test Data Generation

Keywords

Symbolic Execution, Real Execution, Evolutionary Algorithms, Genetic Algorithms, Contract Mutation

1. INTRODUCTION

Although 50% of the software development costs are spent on software testing [11], software providers still release systems which sometimes fail. Depending on the type of systems, failures may have disastrous damages on business and the society [13]. Hence, there is a real need to find effective and economic ways of assuring the quality of software systems.

Test automation is a means of reducing the time and cost spent during software testing. It includes the automatic generation of test data. Since the 70's, researchers and practitioners have been developing algorithms and techniques which were mainly based on symbolic or concrete execution of the program under test [20]. Nowadays, a promising approach is to model the test data generation problem as a search problem which is then solved using meta-heuristic search algorithms like Genetic Algorithms, a sub-class of Evolutionary Algorithms [21]. There is a large number of specific publications introducing new ways of using these algorithms to tackle the test data generation problem. These new techniques are often not linked to the existing ones. Besides, there is some confusion in the literature. The expressions "test case generation" and "test data generation" are often used interchangeably although they denote different problems. All that makes it hard for students and researchers interested in this area to get a solid understanding of what generating test data is about.

The contribution of this article is as follows: in section 2, we first introduce the test case and test data generation problems. Thereafter, we review some basic techniques like constraint-based test data generation, symbolic and concrete execution. Next, we focus on the state of the art techniques of test data generation which are grouped under the term search-based test data generation. We also discuss some known limitations of test data generators. In Section 3 we present an automated black-box test data generation technique for web services which is based on Design by Contract and mutation testing. In section 4 we present some tools which can be used by software testers to enrich their daily test automation experience. Finally, we conclude this article in section 5 by giving our opinion on the subject.

2. CONCEPTS

2.1 Definitions

In software engineering, testing is the process of exercising or evaluating a system or system component by manual or automated means to verify that it satisfies specified requirements or to identify differences between expected and actual results [4]. The system under test can be a method, a class, a module or a whole application. Automating the testing process consists of multiple facets: Automatic test execution, automatic generation of test artifacts like test scripts, test classes, test methods, etc, automatic generation of test cases and test data [16].

In the literature, the expressions "test case generation" and "test data generation" quite often refer to the same problem although they denote in reality different problems.Given a system under test (SUT) and a testing criterion like "every statement has to be executed at least once". The goal of software testers is to generate a set of test cases, commonly called test suite, so that a high coverage can be achieved. The generation of a test suite involves the generation of test cases which in turn involves the generation of test data. The definition of both problems given below is inspired by the definitions found in [12, 20, 16, 4, 3].

The Test Case Generation Problem.

- Define a sequence of operations to be executed on SUT.
- Define corresponding input data and expected results.

The Test Data Generation Problem.

- Define input data which can be used to execute a sequence of operations on SUT.

Note that the test data generation program does not address the problem of generating expected results since their definition requires a third party, also called test oracle, which can be a man or another running system.

Automating the process of generating test data is achieved by means of a test data generator. We distinguish between black-box, white-box and gray-box test data generators where the latter is a combination of the first and second type of generator.

2.2 Black-box Test Data Generator

Black-box generators are systems which produce test data without the need to know anything about the internal structure of the system under test.

2.2.1 Random Generators

The most simple form of test data generators is a random generator. Random generators can theoretically be used to generate any kind of test data because data on a computer are basically bit streams. These generators are often used by other generator types for the generation of initial data which are then refined depending on the technique used. We consider in the following another type of black-box generator called syntax-based generator.

2.2.2 Syntax-based Generators

Grammar-based or syntax-based test data generators are generators which produce test data based on a grammar defining the syntax of the system under test's input [12]. The grammar can be expressed in Backus Naur Form (BNF). Grammar-based generators were initially developed for testing language processing tools like compilers and interpreters. But, they can also be used for testing GUI applications and command-driven software.

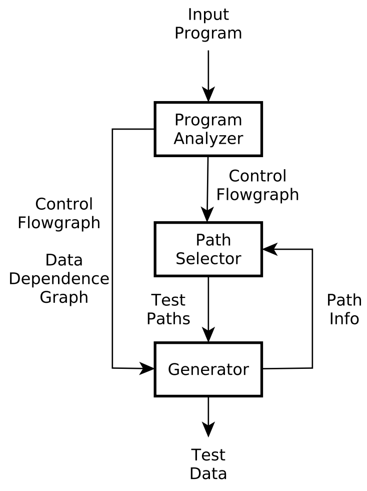


Figure 1: Architecture of a white-box generator

2.2.3 Other black-box generator types

Beside random and grammar-based generators, there are other types of black-box generators. There are publications reporting the use of meta-heuristics search techniques introduced in section 2.3.3 to generate test data based on a Z specification, a logical description, of the system under test [21]. In section 3 we present another black-box generation method which is based on Design by Contract and mutation analysis.

2.3 White-box Test Data Generator

White-box test data generators get the source code of the system under test as input and generate test data which satisfy a structural testing criterion like statement, branch and path coverage. Two well-known classes are path-oriented and goal-oriented test data generators. The architecture of a generator belonging to one of these classes is shown in figure 1. The program analyzer provides all relevant information needed during the generation, such as data-dependence graph and control flow-graph. The path selector then selects the paths that need to be traversed in order to satisfy the testing criterion. Depending on whether the generator is path-oriented or goal oriented, the selected paths can be either specific or unspecific [15]. A specific path is a sequence of nodes belonging to the control flow-graph such that there is an edge connecting each node in the sequence with its successor node except for the last node.

Given now a set of paths, which is a singleton set in the case of a path-oriented test data generator, the goal of the generator is to produce test data such that these paths are traversed during the execution of the system under test. Several approaches to tackle this problem exist.

2.3.1 Real Execution

A random test data generator is used to produce initial test data which are then used as input of the system under test. During the execution, the execution flow is monitored. In case that an undesirable path is taken, some search techniques are used to determine how the input data should be modified. The modifications are then performed and the execution backtracks to the node where the control flow took the wrong direction [15, 20]. This process is repeated until the right path is taken. The generator finally outputs the data for which the right path was taken.

2.3.2 *Constraint-Based Approach*

A constraint-based approach builds up a system of algebraic constraints in terms of the input variables which can then be solved by a constraint solver. Sometimes, the constraint system is unsolvable. This can occur when floating-point variables are used or when the constraints are non-linear [22]. Constraint-based approaches differ in the way the constraint system is built.

Symbolic Execution.

Symbolic execution is a technique which was widely used in the 70's. By symbolic execution, we mean the process of assigning expressions to program variables as a path is followed through the code structure [21]. Thereby, symbolic values are used instead of real values. Consider, for example the following program:

```
public boolean function(int x, int y){
  if(x * y < 100){
   return true;
  }else{
   return false;
  }
}
```

When symbolically executing this program with symbolic values a and b for the variables x and y respectively, we obtain the constraint: $a * b < 100$, if we want to traverse the "if" block.

Concolic Execution.

We have seen above an example of a non-linear constraint. In practice, non-linear constraints are more complex. To reduce the complexity, researchers came to the idea of combining symbolic and concrete execution of the system under test. This is known as Concolic Execution [22]. In the above example, the values a = 1 and b = 4, satisfy the constraint. One can replace the symbolic value a in the constraint with the real value 1 to get the linear constraint $b < 100$.

Mutation testing.

A constraint-based test data generation method which is based on mutation analysis was introduced by Richard DeMillo and Jefferson Offutt [14]. A brief overview on mutation analysis is given in section 3. One goal of mutation analysis is to design test data which kill program mutants[1], that is, test data which causes the program under test to produce an output which differs from that of program mutants. That difference in output can be expressed using algebraic constraints which depend on the input variables. By solving the obtained constraints we obtain test data which are able to kill program mutants. Consider, for example the following program under test:

```
public void swap(Object a, Object b){
  Object c;
  c = a;
  a = b;
  b = c;
}
```

and the mutant

```
public void swapMutant(Object a, Object
    b){
  Object c;
  c = b;
  a = b;
  b = c;
}
```

Note that the mutant is obtained by replacing the statement $c = a$ by $c = b$. To kill the above mutant, the constraint: a != b should hold. Setting a = 1 and b = 2 give us concrete test data.

2.3.3 *Search-based Test Data Generation*

Search-based test data generation is the state of the art in automated test data generation. A search-based test data generator uses meta-heuristic search algorithms such as Hill Climbing, Simulated Annealing and Genetic Algorithms, special Evolutionary Algorithms [21]. These algorithms are not stand-alone algorithms like merge-sort, but rather frameworks that need to be adapted to a particular problem. To use such an algorithm, the testing criterion needs to be transformed into an objective function which guides the search. Furthermore, the search space (set of all possible program inputs) should be encoded such that neighboring solutions in that space are also in the same neighborhood in the encoded space [21]. For further information on Hill Climbing and Simulated Annealing, the readers are referred to [21].

Genetic Algorithms.

Genetic Algorithms are special Evolutionary Algorithms which use operators inspired by genetics and natural selection. These algorithms maintain a population (set) of solutions which are encoded as a sequence of simple components. As an example: the singleton population <112, 255, 52> might be represented as 011100001111111100110100, where each number is replaced by its 8-bits binary representation.

The population is iteratively recombined and mutated to evolve successive populations known as generations. The recombination operator takes two parent solutions and produces two new offsprings. As an example, consider the following two individuals: 000000001111111100000000 (<0, 255, 0>), 111111110000000011111111 (<255, 0, 255>). A single cross-over at location 12 yields the following children: 000000001111000011111111 (-<0, 240, 255>), 111111110000111100000000 (<255, 15, 0>). To decide which individuals of the current population are used for the recombination, selection algorithms are used. The selection is based on the "fitness" of an individual. The "fitness" can be the value obtained from the objective function or that value scaled in some way.

2.4 Limitations of Test Data Generators

We first note that the test data generation problem is theoretically undecidable [21]. Due to this, most of the work presented in the literature is based on simple programs, programs short in length or low in complexity [15]. We present

[1]A mutant is obtained by performing some syntactical changes on SUT

in the following some known limitations of test data generators.

The probability of discovering faults which are only revealed by a small percentage of the program input is low when using random generators. These generators do not perform well in terms of coverage. Dynamic generators, generators which execute the system under test, perform better. But, they require many iterations before the right path is taken during the execution. It is not guaranteed that such generators can find test data which traverse the considered path. Static generators, generators which rely on symbolic execution, have difficulties to handle function calls, since the code of the called functions might not be accessible. Dynamic structures like arrays, collections and objects pose another major problem since they cannot be inferred in a static manner if the program code is the only source of generation [15]. If a logical description of the system is available, then the behaviour of objects can, to some extent, be automatically generated [17]. We also note that constraint-based generators might sometimes fail to find a solution since constraint-satisfaction is a difficult problem.

3. A BLACK BOX GENERATION METHOD

3.1 Mutation Analysis

Mutation Analysis is a fault-based technique which is based on a defined fault-model. The model mimics common programming mistakes such as using the wrong operator or variable [12]. The model is implemented by the so-called mutation operators which affect the syntax of the input. The method works by applying one operator at a time on the input program to create mutants. The goal is then to design test data which kill program mutants. A mutant is killed when its output differs from that of the program under test. The test coverage is defined in terms of the number of mutants killed [12].

3.2 Web Services

Web services are commonly used as a means to integrate different applications. Their reliability is therefore an important concern. Since the interface expose by a web service can be defined in a WSDL file which only focuses on syntactic information, the World Wide Web Consortium has proposed Web Service Semantics (WSDL-S) [10] as a means to annotate the WSDL document with semantic information like contract specifications.

3.3 Contract-Based Mutation for Web Services

Unlike traditional mutation analysis, which applies mutation operators on program code [12], the method described here applies them on web service interface contracts [19]. The contracts, consisting of a precondition and a postcondition, are defined using WSDL-S. The grammar defined in [18] shows the structure of the expressions which are supported when defining contracts.

3.3.1 Fault-Model

The fault-model presented below focuses on the discordance between web service specification and contract definition.

Contract Negation (CN):
The contract may be mistakenly negated.

CE		CS	
Original	Mutant	Original	Mutant
Precondition	Postcondition	Precondition	True
Postcondition	Precondition	Postcondition	False

Table 1: Effects of CE and CS

PW		PS	
Original	Mutant	Original	Mutant
==	>=, <=	>=	>, ==
>	>=, !=	<=	<, ==
<	<=, !=	!=	>, <
P >= Q	P >= Q-constant	P > Q	P > Q+constant
P <= Q	P <= Q+constant	P < Q	P < Q-constant
forall	exists	exists	forall
&&	||	||	&&

Table 2: Effects of PW and PS

Condition Exchange (CE):
The precondition of the service interface may be used to express the contract postcondition, or the postcondition of the interface result may be used to define the contract precondition.

Precondition Weakening (PW):
The contract precondition is weaker than the interface precondition.

Postcondition Strengthening (PS):
The contract postcondition is stronger than the service interface postcondition.

Contract Stuck-at (CS):
This fault-model element is more on the logical level. It can be that programmer defines a contract precondition which is logically equivalent to true and a contract postcondition which equivalent to false.

3.3.2 Mutation Operators

Five mutation operators are defined: CN, CE, PW, PS and CS. Their effects [18] are presented in tables 1,2, 3. These tables show how a symbol which occurs in the initial contract can be transformed into a symbol which occurs in the mutated contract.

3.3.3 Data Generation

The generation process consists of three major steps: initial data generation, selection of initial test data, and contract mutation. The last step can be further divided into two sub-steps which are: mutants generation and final test data selection.

Initial Data Generation and Selection.

Original	Mutant
P	!P
==	!=
>	<=
<	>=

Table 3: Effects of CN

During the generation of initial test data, equivalence class and boundary value testing techniques are combined. For each parameter of the service interface, the data type and the precondition are retrieved. Then the input domain of the parameter is partitioned into valid and invalid equivalence classes. A certain number of random values are generated for each equivalence class. Then the boundary values of the class are added to the randomly generated data. Finally, the Cartesian product of all data sets is built, resulting to the initial test data set which is then filtered according to some heuristics [19].

Contract Mutation.

As stated in section 3.1, the generation of contract mutant is done by simply applying mutation operators one at a time on the original contract. The mutants are then encapsulated with the web service as a whole and executed on each previously selected test data. The selection of the final test data is done one by one with the help of a greedy algorithm. A test data is selected if it kills the maximal number of mutants not killed by the previous selected data. A contract mutant is killed if one of the following conditions holds:

- The results of the original precondition and the precondition of the mutated contract are not the same.
- The results of the original postcondition and the postcondition of the mutated contract are not the same.

3.3.4 Experimental Results

The method was applied on the triangle type program. The contract specification for the program is as follows:

$$\begin{array}{lll} \textit{Precondition} & : & i >= 0 \wedge j >= 0 \wedge k >= 0 \\ \textit{Postcondition} & : & @return == 1 \vee @return == 2 \\ & & \vee\ @return == 3 \vee @return == 4 \end{array}$$

where i, j and k denote the length of the three sides. The return values 1,..,4 mean isosceles, equilateral, general and not a triangle. When choosing 10 and 7 data items in the valid and invalid equivalence class of each parameter respectively, the method first generates 4913 test data [19]. That number is reduced to 3100 after the initial selection. The number of program mutants generated by the method is 24. After finally selecting the data so that the largest number of mutants is killed, we get 6 test data. With these, the method achieved a statement coverage of 65% and a branch coverage of 60%. The contract mutation score was 100%

4. TOOL SUPPORT

In this section we briefly present some tools that software testers can use to generate test data.

When "googling" for the expression "test data generator", the Google search engine will return some interesting results. Among them are the websites [1, 6]. These two sites give testers the possibility to generate a limited number of data records which can be used to populate a database. Both provide the possibility to download the generated data set as a .csv, .json, .xml or .sql file. To generate a larger number of data records, users have to register and pay some fees.

Beside websites, there are also many language specific tools. On the Microsoft side, PEX is an interesting choice. It is a tool incorporated in Visual Studio [16]. The web site [7] gives the possibility to test the capabilities of PEX by writing a method in C# which defines some constraints. The tool will then remotely attempt to find some inputs which satisfy the constraints specified in the method. PEX integrates several techniques for test data generation: random, Design by Contract, concolic. In the survey carried out in [16] PEX is one of the tools which passed all tests defined to assess to the capabilities of mature test data generators. Randoop is another test data generator which can also be used on the .NET platform [8].

Randoop was initially developed in Java and ported later to .NET. The Java version is available for download [8]. It is primarily a command-line tool. It can also be integrated into the Eclipse IDE via the provided Eclipse plug-in [9]. The plug-in officially supports Eclipse 3.5 and 3.6. To see how Randoop performs in comparison to the method presented in section 3, we implemented the triangle type program. We installed the Randoop Eclipse plug-in in Eclipse 4.5. The following shows how we have configured Randoop:
Maximum test per file: 50
Maximum test size: 100
Random Seed: 0
Stopping Criterion:
When Randoop has generated 100 tests or when it has generated tests for 100 seconds. With the above configuration, Randoop generates two test files and one file containing a main method which can be used to launch all generated tests. The following shows an example test method generated by Randoop:

```
public void test50() throws Throwable {
  if (debug) System.out
    .printf("%nTriangleTest0.test50");
  test.data.gen.TriangleType var0 =
    new test.data.gen.TriangleType();
  int var4 = var0.getTriangeType(3,
      100, 3);
  int var8 = var0.getTriangeType(3, 3,
      100);
  int var12 = var0.getTriangeType(0, 0,
      10);
  int var16 = var0.getTriangeType(0, 1,
      100);
  int var20 = var0.getTriangeType(3,
      100, 0);
  int var24 = var0.getTriangeType(10,
      2, 100);

  // Regression assertion
  //(captures the current behavior of
      the code)
  assertTrue(var4 == 1);

  // Regression assertion
  //(captures the current behavior of
      the code)
  assertTrue(var8 == 1);

  // Regression assertion
  //(captures the current behavior of
      the code)
  assertTrue(var12 == 1);

  // Regression assertion
  //(captures the current behavior of
      the code)
```

```
        assertTrue(var16 == 3);

        // Regression assertion
        //(captures the current behavior of
            the code)
        assertTrue(var20 == 3);

        // Regression assertion
        //(captures the current behavior of
            the code)
        assertTrue(var24 == 3);

    }
```

The test suite achieved the following coverage results: Statement Coverage: 100%, Branch Coverage: 80%

Beside open source solutions, there are also commercial ones which can be used to generate test data for Java programs. Two of them are JTest [5] and AgitarOne [2].

5. CONCLUSION

Like code generators which are supposed to speed up the software development process, test data generators are a means to speed up the testing process. Although many fortune companies rely on tools like AgitarOne and JTest to produce top-quality software [5, 2], my personal opinion on the subject is that test data generators are not widely used in industry because of the following observation: Many people tend to do what they are used to and are reluctant to try something new. What we are used to often depends on the teaching we received. But, since there is no special class which teaches how to use test data generators and how to successfully incorporate them in the testing process, many software testers will never come across the idea of generating test data. If some do, they will probably not integrate it in a proper way in their testing process. This will lead to some complications and possibly to abandoning the test data generator.

6. REFERENCES

[1] `http://www.generatedata.com/`. Accessed in November-2015.

[2] AgitarOne. `http://www.agitar.com/solutions/products/automated_junit_generation.html`. Accessed in December-2015.

[3] IEEE Standard 24765-2010: Systems and Software Engineering Vocabulary. Accessed in December-2015.

[4] IEEE Standard 610.12-1990: Glossary of Software Engineering Terminology. Accessed in December-2015.

[5] JTest. `https://www.parasoft.com/product/jtest/`. Accessed in December-2015.

[6] Mockaroo. `https://www.mockaroo.com/`. Accessed in November-2015.

[7] PEX for fun. `http://www.pexforfun.com/`. Accessed in December-2015.

[8] Randoop. `https://randoop.github.io/randoop/`. Accessed in December-2015.

[9] Randoop Eclipse Plugin. `https://rawgit.com/randoop/randoop-eclipse-plugin/master/plugin/doc/index.html`. Accessed in December-2015.

[10] Web Service Semantics. `http://www.w3.org/Submission/WSDL-S/`. Accessed in November-2015.

[11] D. S. Alberts. The economics of software quality assurance. In *Proceedings of the June 7-10, 1976, National Computer Conference and Exposition*, AFIPS '76, pages 433–442, New York, NY, USA, 1976. ACM.

[12] P. Ammann and J. Offutt. *Introduction To Software Testing*. Cambridge University Press, 2008.

[13] R. Charette. Why software fails [software failure]. *Spectrum, IEEE*, 42(9):42–49, Sept 2005.

[14] R. DeMillo and A. Offutt. Constraint-based automatic test data generation. *Software Engineering, IEEE Transactions on*, 17(9):900–910, Sep 1991.

[15] J. Edvardsson. A survey on automatic test data generation, 1999.

[16] S. Galler and B. Aichernig. Survey on test data generation tools. *International Journal on Software Tools for Technology Transfer*, 16(6):727–751, 2014.

[17] S. Galler, A. Maller, and F. Wotawa. Automatically extracting mock object behavior from\ dbc specification for test data generation. 2010.

[18] Y. Jiang, S.-S. Hou, J.-H. Shan, L. Zhang, and B. Xie. Contract-based mutation for testing components. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 483–492, Sept 2005.

[19] Y. Jiang, Y.-N. Li, S.-S. Hou, and L. Zhang. Test-data generation for web services based on contract mutation. In *Secure Software Integration and Reliability Improvement, 2009. SSIRI 2009. Third IEEE International Conference on*, pages 281–286, July 2009.

[20] B. Korel. Automated software test data generation. *IEEE Trans. Softw. Eng.*, 16(8):870–879, Aug. 1990.

[21] P. McMinn. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.

[22] M. Nazim and M. Yadav. Automatic program based test data generation. *International Journal of Emerging Technology and Advanced Engineering*, 5, 5 2015.

Supportive Role of Big Data Technologies in Enterprise Architecture Management

Simon Hacks
RWTH Aachen University
hacks@swc.rwth-aachen.de

Mahdi Saber
RWTH Aachen University
mahdi.saber@rwth-aachen.de

ABSTRACT

By the advent of more complex enterprise systems consisting of different types of processes, applications and information which are linked to each other, the growth of scalable supportive technologies becomes an inevitable necessity. Additionally, expanding these systems over heterogeneous machines and making sophisticated interdependencies between them cause their management to become a complex task which needs to monitor divergent types of data with no holistic schema. Based on these assumptions, Big Data technologies seem to be good candidates to maximize the value of grasped insight.
This article describes some state of the art approaches with regards to the usage of Big Data technologies in Enterprise Architecture Management (EAM). In addition, it will be discussed how these approaches can be fitted into related EAM patterns to tackle the complexity of EA structures during its lifecycle.

Keywords

Enterprise Architecture Management, Big Data, Hadoop, MapReduce, Spark, Storm, EAM Pattern Catalog, Business Analytics, Management, EAM

1. INTRODUCTION

Regardless of your inclination towards the technology, due to its daily advancement, hearing about the new magical world, Big Data is inevitable. Due to the emergence of various data-related advancements, it is now regarded as one of the hottest topics in media. Last year, Edward Snowden, the American whistle-blower, has revealed how NSA is able to store and analyze huge amounts of data collected from different sources all around the globe [11]. This new phenomenon is not limited to political area, it also affects different aspects of our lives and revolutionizes how we live, work and think from economy to health care, from art to politics [13, 14].
From an economic point of view, we are now living in a more complex world. Integrating technology into our everyday lives provides enterprises with the opportunity to deal with and grasp new markets. As a result, if a company wants to increase its share of market, it inevitably needs to empower itself with managing different scenarios; in other words, empowerment means running sophisticated processes and handling more data. In recent years, because of having too much data at hand resulted from increasing useage of electronic devices with various operating systems, applications and usage scenarios, new advent of analytical techniques for getting more insight into these sources is necessary. Consequently, the complexity of new era reflects its nature not only on the data, but also on internal structure of a company dealing with these sources.
The general structure of an enterprise is reflected in its EA (Enterprise Architecture) which consists of both IT and business aspects. Although the concept of "architecture" was accepted for enterprise management in the early 80s, there was no significant cases to support it till 90s [21]. Since then, some popular management approaches have been appeared to mitigate this deficiency. Among those, after an early concise discussion, we stick to the pattern-based trend [4]. The main purpose of this approach is to improve the interaction of business and IT aspects. Therefore, similar to its root being originated in software engineering, this method tries to identify recurring problems in EAM (Enterprise Architecture Management) and suggest some abstract solutions as building blocks of an architectural blueprint[1].
Additionally, due to data explosion in our society, a whole new range of advanced techniques has been come out as an emergency relief. Among these, we can name some of the well-known ones such as data mining, graph mining and predictive modelling. However, developing and using such techniques for every scenario from the scratch seems to be too complex and costly. As a result, some of Big Data platforms currently used in renowned enterprises like Google, Yahoo and Amazon yield simpler tools and techniques for dealing with such situations [15].
All in all, the main focus of this article is to identify some appropriate enterprise architecture patterns which benefit from the usage of renowned Big Data tools and platforms. To achieve this goal, after specifying the related works, we use EAM pattern catalog [12] to determine some patterns. Then, after clarifying our definition of Big Data, we move on to enrich these patterns by some suggested state-of-the-art platforms and tools in this area. Finally after a brief discussion on their differences, we try to pick the best one for a concrete scenario, analyzing technology homogeneity, with respect to its requirements.

1.1 Related Works

By the advent of complex enterprises during the last two decades, Enterprise Architecture (EA) management has become more sophisticated. Although one of most renowned approaches, the Zachman framework [20], did not grasp too

[1]https://wwwmatthes.in.tum.de/pages/3b4t6l34g936/EAM-Pattern-Catalog-Wiki, Accessed: 2015-12-1.

much attention in its first beginnings, it gradually becomes more popular for managing this complexity. The essence of this approach is developing an ontology of Enterprise Architecture in a structured manner. The main focus of this approach is to find abstract answers for these kinds of questions: Why, How, What, Who, Where, When. However, it does not suggest any specific methodology for collecting and managing the information [22].
On the other hand, The Open Group Architecture Framework (TOGAF) [16] tends to modularize the enterprise architecture and tries to optimize the interaction of these components. In order to achieve this aim, it not only recommends some formal standards, but also uses of successful technologies and products. In the core part of this well-documented framework, the Architecture Development Method (ADM) has a key role. Although ADM is based on a stepwise cyclic approach, this iteration should not last indefinitely. It is recommended to be used during the project coarse with specific times for completion [18].
All in all, in this article we stick to the third one which is the EAM pattern-based approach. It was firstly introduced by [1] and tried to find a solution for implementation difficulties of previous approaches. According to the perception of its supporters [12], EAM frameworks are too abstract or extensive to be implemented in a productive manner. Too abstraction causes executives to miss the initiation of their tasks. By lacking the beginning point, companies tend to gather information from all potential stakeholders. This all embracing demand gathering puts a lot of effort for collecting and neglecting of unnecessary information EAM frameworks and EAM pattern-based approaches are not contradictory, so the patterns can be used as complements of frameworks.
Additionally, Big Data frameworks need this initiation to be deployed. Therefore, they come into the context by providing more implementation-centric analyses and suggestions. As a result, we believe that sticking to a multilateral approach with less abstraction will be more reasonable.

2. RELATED EAM PATTERNS

During the research phase, we have noticed that EAM pattern catalog [12] has a significant role in EAM pattern-based approach. It is based on experimental surveys among some renowned enterprises. According to the definitions given in the catalog, all patterns can be characterized as the following:

- **Methodology pattern (M-Pattern):** clarifies the steps in order to satisfy some concrete concerns with respect to the intended usage context. These kind of procedures can cover different activities such as visualizations, group discussions and metrics calculations. One of the benefits of these patterns is their ability for integrating with EAM frameworks, e.g. TOGAF ADM.
- **Viewpoint pattern (V-Pattern):** Since industrial users generally clarify their viewpoints by examples, this kind of patterns propose a visual way to present data. These patterns can be used by M-patterns; they can be supported with some minor textual explanations.
- **Information Model Pattern (I-Pattern):** contains the definitions and explanations of the used objects in one or several V-patterns. These descriptions are presented as an information model.

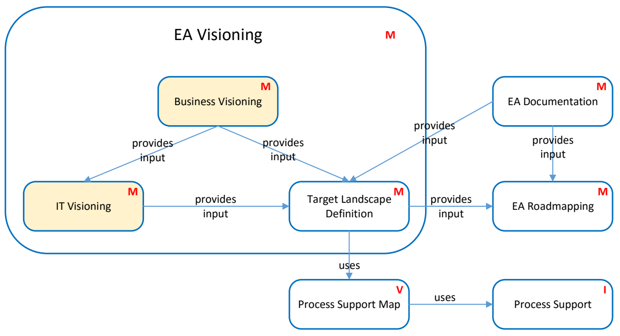


Figure 1: Related EAM Patterns Map [2]

Upon the above definitions, all the related patterns to this article are depicted in Figure 1. By looking closer into the suggested patterns we have noticed that most of them are about currently established structures. To invite Big Data technologies into the picture, we need some future-oriented patterns. Fortunately, we found our desired patterns as the following:

1) **Business Visioning:** a M-pattern selected from the EAM Pattern Catalog V2[2] which discusses the development of a business vision in a stepwise documented manner. This pattern is mainly used for arranging the business strategy of the enterprise [2].

2) **IT Visioning:** a M-pattern selected from the EAM Pattern Catalog V2[2] which tries to give some insights into the development and maintenance of a vision about future usage of information technologies in the governed enterprise [2].

These two patterns are part of a larger pattern language and therefore presenting the relations between all patterns of this bundle are an integral part of our research. These relations are also shown in Figure 1, and you can find some brief explanations in the following:

1) **EA Documentation:** a M-pattern which takes the responsibility of documenting the including elements of current enterprise architecture. Some of these elements are current business applications and business processes [2].

2) **EA Roadmapping:** a M-pattern which is responsible for creation and maintenance of our planned roadmap strategies towards the future development of the enterprise architecture [2].

3) **Process Support Map:** a V-pattern which depicts the supportive roles of BusinessApplications with respect to their BusinessProcesses in each OrganizationalUnits [2].

4) **Process Support:** an I-pattern which describes the concepts and information model behind the Process Support Map pattern [2].

5) **EA Visioning:** a composite M-pattern which describes the process of gaining a holistic enterprise vision by considering the results coming from the Business Visioning,

[2]https://wwwmatthes.in.tum.de/pages/ugsyi19wmmvl/, Accessed: 2015-12-1.

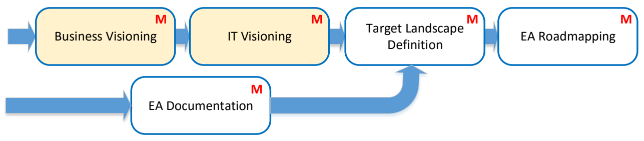


Figure 2: EA Visioning Process [2]

IT Visioning and EA Documentation. The development of an EA vision is depicted in Figure 2.

6) **Target Landscape Definition:** a M-pattern describes the process of defining a target landscape derived from the business and IT vision of the enterprise. The current documentation of the EA is also used as an input for the development process. Thereby, the target landscape defines the vision of future business processes and the support provided by the IT [2].

Big Data technologies, as the name suggests, seem to be related with IT visioning pattern. By doing a literature review however, we have seen that the topic grasps a lot of attention in business area. Considering Figure 2, we can immediately notice the initiator patterns, namely Business Visioning, IT visioning and EA Documentation. These patterns are fed to Target Landscape Definition pattern and consequently EA Visioning pattern.
Since this article is not based on a real project, there is no EA Documentation. In other words, one of the essential initiators is missing. Hence, it is not possible to talk about Target Landscape Definition, EA Visioning and other subsequent patterns.
In a real project, the results coming from applying Business Visioning pattern affect our decisions in IT Visioning phase. However, in this article, we are going to talk about them in an independent way. The main abstract focus of this article is how Big Data technologies can support us in both phases.

3. BIG DATA CHARACTERISTICS

This chapter is the starting point of our survey on Big Data concepts and technologies. In the following chapters after counting important properties of Big Data and their associated problems, we will introduce some related technologies, i.e. Big Data frameworks and tools. Big Data frameworks presented in chapter 5 can be used to support both Business Visioning and IT Visioning patterns. The topic of chapter 6, Self-service Business Intelligent tools, is mainly related to Business Visioning pattern. In chapter 8, Technology Homogeneity, we target only the IT Visioning pattern.
Large amount of data is somehow a raw definition of Big Data. We need a more formal way to detect them. The following characteristics can help us to get more insights:

Volume: Data is never ceasing to increase in huge amounts. Petabytes of data, which are now considered as the yardstick of Big Data, would be possibly overshadowed by the Zettabytes era of data in near future. This exponential growth of data could be explained by many cases. For example, thousands of Terabytes are now generated daily merely due to users' interactions on social networking websites [10].

Variety: One of the things that make Big Data larger than ever is the variety of data and its sources. New various data types came with technology progress in different forms and brought new challenges to data analysis. The Big Data spectrum encloses an eclectic mix of data types that can be mainly categorized into structured, unstructured and semi-structured types. Structured data refers to data that has a predefined length and format, and is usually contained in relational databases and spreadsheets. These are easy to be stored and analyzed and, for instance, could be managed by e-commerce websites. Semi-structured data are structured data in essence that does not conform to the relational database model and have rapid-changing structures, like HTML, XML and RSS feeds that are mostly contained in web server logs. Unstructured data covers all rest of data types that does not follow specified a format or model. Those are still the most challenging type of data for analysis, since it is difficult to derive some structures from them using traditional analytic tools. Such data can be represented by social interactions, images, video, audio, etc [8, 3].

Velocity: Time is always a big factor and plays a significant role in some branches such as financial markets where data analysis should respond promptly to market changes. Data is increasing not only in volume and variety dimensions, but also in terms of data processing time. Hence velocity in Big Data reflects the speed at which nowadays data is being transmitted and processed. The generated data sources vary usually from batch to streaming data, of which the latter is the most challenging in terms of analysis. Streaming data, due to its high velocity and the urgent need of reaction, are more difficult to process. Clickstream data are, for instance, consistently collected and analyzed by Google AdWords in order to generate suitable purchase recommendations to its visitors. Geologists are empowered from analyzing incoming streams of data from underground sound sensors. They should listen to earth layers movements, and therefore able to make predictions on earthquake occurrences earlier. Those predictive analytics often play a far-reaching role in mitigating risks [6].

Besides the 3Vs Model mentioned above, there are other properties that may clear up other ambiguities about the definition of Big Data:

Veracity: Nowadays, it is even harder to prove the veracity of incoming data due to their diverse formats and different sources. Cleaning data of any impurities is a critical necessity for firms acting instantly on changing markets, if they want to get the right business insights and resonable decisions. This hence requires some mechanisms dealing with imprecise and inaccurate data [3].

Value: The most important character of Big Data analytics after all is the abstracted value of the collected data. Most businesses are interested in this issue to increase their opportunities. However, now it is more challenging task because of all other data exploitation problems. From the emergence of new data sources to real time processing of unstructured contents, new analytic workloads have to be set by IT professionals to design robust systems which are able to deliver an effective and efficient analysis on large amounts of data [3].

The last two mentioned properties not only confirm the complexity and its related issues, but also extend the 3Vs-Model of Big Data to the 5Vs. The following section focuses on the exploitation problems related to the characteristics of Big Data.

4. BIG DATA PROBLEMS

As stated before, Big Data is voluminous, comes from different sources with various formats, and arrives mostly at a very high rate, which reflects the increasing velocity of incoming data. Institutions from different background and companies are now more interested in all the concealed information behind Big Data. Thus, due to complexity, extracting knowledge and insights from large sets of data, in order to adjust business processes to prompt market changes or support decision making, is no longer a straightforward task. Data value retrieval is hence one of the key challenges of Big Data especially in the case of unstructured data. The portion of unstructured data on the Internet has surprisingly increased over the last few years through users' interaction on social media. The unstructured data cannot be uniformly analyzed and are more difficult to process than structured data. Therefore, getting an effective expression of these unstructured data is definitely one of the key exploitation problems of Big Data. Besides the variety problem of incoming data, processing large amounts of them takes a long time. These days, data spans rapidly in many dimensions and exceeds the present processing and storage capacities. At the moment, the available computing resources have already been far behind the exponential growth of data. Hence, processing and storage liabilities affect the core competitive capacities of enterprises and constitute another exploitation issue in this area, notably the stream processing. This occurs because analyses of data have to take place before the data is stored. Considering the velocity at which data is generated, human analysis is mostly unfeasible. Evolutionary analytic workloads and mechanisms are required to handle stream data, and help automate the decision-making process in order to get instantaneous responses and keep the business optimized and up to date. The efficient processing of Big Data has become a core problem. Given the openness and richness of the digital universe, data is very diversified and can originate from heterogeneous sources. The aggregated data may contain large amounts of uncertain, incomplete, incorrect and repeated data. This makes it even harder to distinguish appropriate data or to ensure the authenticity of data. The quality of the acquired data far surpasses the quantity of data, bringing to the light relevance of 80-20 rule. Thus the reliability of Big Data has become a serious issue to extract valuable data. Therefore moving forward, these are the main challenging questions around the exploitation of data [10]:

- Does all the data need to be stored or analyzed?
- How do we find out which data is most relevant?
- Does the extracted data add value?
- Is the stored data accurate enough for decision making?

In order to deliver high quality data, incoming data has to be preprocessed to prepare them for future analysis and storage. Unfortunately this topic is beyond the scope of this article, but interested readers can refer to [7].

5. BIG DATA FRAMEWORKS

More abstract and low-level type of tools formed by parallel computing are frameworks, which allow for a general involvement of different procedures, since they are commonly not explicitly bound to a certain functionality. Some analytic approaches exploit underlying frameworks for an efficient distributed processing, without particularly concerning about parallel computing issues such as synchronization or file management. One of the most prominent frameworks in this context is Hadoop, which provides a simple usage of the MapReduce model based on a scalable and robust parallel system. Starting with Hadoop and its associated tools, some of the state-of-the-art frameworks for processing and analyzing Big Data will be introduced and summarized as the following.

5.1 Hadoop [17]

Hadoop is an open-source framework from Apache based on the MapReduce parallel programming model, which was originally introduced by Google in 2004 and is used for distributed computing on large data sets. It was designed to solve the common problems of distributed computation, like synchronization, scalability, and load-balancing. Therefore, MapReduce is a modular structured model, with the focus on two core methods, namely map and reduce. They form the actual interface between the programmer and framework, while the mentioned tasks remain hidden in the background. Basically, huge amounts of data, either stored in a single source or multiple sources, are read in parallel and automatically distributed to different nodes in a computing cluster. Managed by a single "master-node" called JobTracker, any node within the cluster then initializes so called mappers. Mappers are classes including the map method implemented by the programmer. While each mapper object is responsible for a set of records, obtained from the input files, each including map-task processes exactly one record. Due to this, every node is able to apply a common function to chunks of the original data set in parallel, without having to worry about synchronization. Results produced by mappers are afterwards emitted as key-value pairs and shuffled to the reducers, where each of them collects all pairs having the same key. This step is also the only communication taking place between nodes during the whole procedure. The reducer class represents the merge part of the popular Divide&Conquer-paradigm. Since the data set was split up and processed individually, reducers need to combine the individual results. How this is exactly done as defined in the reduce method or in optional intermediate shuffle and combiner methods, which can be customized by the programmer. In the last step the final results are automatically written to the output source in parallel. For a better understanding, a scheme of this process is illustrated in Figure 3.
Besides the implementation of MapReduce paradigm, Hadoop also provides an underlying distributed file system called HDFS (Hadoop Distributed File System), to store and handle large amounts of data even up to hundreds of petabytes. It was designed to operate on common hardware by considering arising problems in distributed and high-performance data processing. Some of those problems are high throughput of data, high accessibility regarding failure occurrence and efficient partitioning. Therefore, HDFS uses a blockwise and redundant storage architecture, where every node in the cluster stores predefined sized blocks of data. In gen-

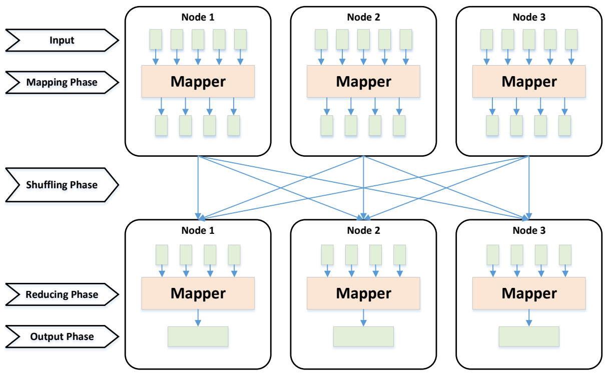


Figure 3: MapReduce Phases

eral, these blocks are chosen to be proportionally small such that they allow for a higher degree of parallelization. Furthermore, the blocks are stored redundantly among different nodes to prevent transfer bottlenecks and data losses.
Hadoop's biggest advantage is the abstraction of the actual application scenario, while automatically dealing with the tasks arising with parallel programming. However, Hadoop is just recommended for batch processing, in other words, streams of incremental or real-time data cannot be analyzed efficiently. In addition, the exploitation of the framework to parallelize existing approaches is not always trivial due to the static MapReduce model. Nonetheless, Hadoop has become a de facto standard for parallel batch-processing of large amounts of data used by Yahoo and Facebook.

5.2 Storm [19]

Although Hadoop is widely used for a distributed batch-processing of data sets, it has the mentioned drawback of lacking a solution for real-time application scenarios. For this reason, a subsequently introduced solution named Storm was developed as a distributed real-time computation system, to compute on continuous streams of data. Now it becomes an essential part of many enterprises such as Spotify, Twitter, and Yahoo.
Derived from Hadoop, Storm follows the master-slave model to handle topologies (the counterpart of jobs in Hadoop). Equivalent to the JobTracker, a "master-node" called Nimbus organizes the distribution of the code, assigns tasks, and monitors failures. Additionally, all other slave nodes themselves are organized by supervisor daemons and connected to the nimbus node via Zookeper, which is a centralized service for managing distributed computations. The actual processing of data is done via two main components, i.e. Spouts and Bolts.
Spouts are sources of one or more streams, whereas Bolts can be regarded as processing units receiving streams from Spouts, apply some functionality and optionally hand over their output to further Bolts. Both of these component types also form a topology similar to directed graphs, where the edges between Spouts and Bolts define the process flow. In particular, each Spout reads tuples from an external source and creates streams of tuples that are sent over to Bolts.
This can either be done either reliably or unreliably, such that a Spout temporarily saves the emitted information in case of further processing fails. To handle all sorts of streams it is possible to define serializer objects for each type within a tuple. After receiving a stream from one or more Spouts, Bolts perform stream transformations, which can be filtering, accumulating or other forms of operations.

5.3 Spark [23]

Spark is a framework for large-scale data processing, which is developed by Apache. Besides general features of MapReduce-based processing, it allows for efficient in-memory computation. Due to the excessive usage of persistent storage in most iterative implementations of Hadoop, it was invented to serve low-latency applications.
The main distinguishing aspect of Spark is its fault-tolerance and Resilient Distributed Datasets (RDDs) [24]. RDDs allow programmers to explicitly persist data in main memory in combination with associated operators for in-memory computations. In contrast to Hadoop, where for each job iteration data is read and write in secondary storages, the same operations can be performed using main memory up to 100 times faster.
To manage data within the main-memory, Spark uses RDDs as distributed memory abstractions. In addition to accessibility features, they allow persistent data storage and recovery via some Spark programming interfaces. In case the initially transferred data exceeds the available memory, the programmer can define alternative behaviors, for example supplementary data can still be processed from distributed secondary storages. For this reason, Spark is able to interact with existing file systems like HDFS and other types of data sources (HBase, Cassandra, etc.).
Apache has also developed a whole set of distributed data processing tools and libraries, similar to Hadoop's third-party tools presented before. In addition to interaction with existing tools like Hive, Spark also provides its own optimized query engine. Furthermore, a stream engine has been provided in this suite to be used by continuous data analytics in real-time.
In summary, Spark sounds like an all-in-one solution that is able to perform all kind of tasks efficiently. However, it only shows its superior power, if it is used in iterative jobs. In typical scenarios more stable and mature solutions are preferable, since it might be costly to create a shared-memory cluster with huge capacity.

6. SELF-SERVICE BI TOOLS

In contrast to remaining challenges of Big Data, the effective decision-making processes are the keys of the companies' success in this highly competitive business environment today. Therefore, companies use Business Intelligence (BI) systems to assist decision makers. In fact, BI systems provide enterprises with timely and accurate information, which allows them to make suitable decisions. As they can react more quickly on customer needs and market changes, they will gain more profit.
BI tools actually complement Big Data analytics to unlock business value from enterprise information. Companies that augment BI through Big Data techniques gain more holistic views of business and more realistic insights. These can help them addressing customer needs, taking more responsive decisions on incoming risks, and identifying performance opportunities.
In order to ease the usage of BI systems for end-users such as executives, managers and operational staffs, a new kind

of BI tools has been emerged recently. They are called Self-service BI tools and enable business users to become more self-reliant and less dependent on the technical issues [9]. Self-service BI tools can also be called Do-It-Yourself BI (DIY-BI) tools, which apparently shows their aims for reducing IT dependency and favoring end-users involvement. Furthermore, this kind of tools has four key essential characteristics with respect to their underlying frameworks [9]:

- Make easier to access source data
- Make easier to use
- Make easier to consume and enhance the results
- Maker easier to deploy and manage

The evaluation of different Self-service BI tools according to the Forrester 2015 report [5] uncovered a market in which:

- IBM, Microsoft, SAP, SAS, Tibco Software, and Microstrategy lead the market: These vendors demonstrate significant capabilities and a good balance of self-service BI features across many requirements.
- Other vendors that compete with the leading vendors to get stronger position in the market: Information Builders, Tableau Software, Actuate, Oracle, QlikTech, and Panorama Software are strong performers as Self- service BI platforms.

7. DISCUSSION ON FRAMEWORKS

In the previous chapters, multiple frameworks and high-level tools for Big Data analytics were presented. Depending on the actual scenario, each of those tools has its own benefits or drawbacks. In this section we briefly review some remarks of them in order to make an evaluation.
Starting from the introduced frameworks, the major distinction is determined by the way of processing data. While batch-wise processing tools like Hadoop perform certain computations in parallel on large fixed sets of data, stream processing tools like Storm works on continuous data.
This difference consequently reflects its nature on analytic techniques. For example, real-time classification algorithms [7] should be built upon stream processing tools, whereas computations on static complex data sets requiring precise results should generally be done via batch-wise solutions.
Another important criteria to be considered is the available computation time. Hadoop has the advantage of ETL-processing (Extract, Transform, Load), since it was designed for efficient interactions with parallel databases and thus cares for load-balancing and preprocessing. Although it can be used for initialization of greater jobs, however the assigned tasks can rarely be done under some minutes. As a consequence, it should be utilized for complex operations on data sets of tera and petabytes, which makes it an inappropriate choice in many real-time query scenarios.
Furthermore, Even though utilities such as Hive can speed up ad-hoc queries, intermediate phases of the MapReduce model, specially shuffle phase, result in a slowdown. As a result, it is generally a disadvantage compared to stream processing tools.
The rigidity of the MapReduce model also affects performance of the system depending on the input data. If data cannot easily be processed as key-value pairs or split up using the Divide&Conquer paradigm, other variants may be more beneficial.
Hopefully over these advantages and disadvantages, more generic frameworks like Spark are on the rise, where both concepts are combined in a common framework. This in turn also allows for a mixed usage of both techniques, where real-time queries can be combined with more complex computations on huge data sets.
All in all, choosing one framework over the others depends on actual scenarios. Each framework has its own advantages and disadvantage. As a result, in next chapter we are going to compare these frameworks for a concrete scenario, i.e. technology homogeneity.

8. TECHNOLOGY HOMOGENEITY

According to the survey of technical university of Munich [12], one of the most important interests among renowned enterprises is technology homogeneity. By technology homogeneity in this context, we mean which methodologies would be helpful for analyzing and controlling the conformity of used technologies in application landscape. Consequently, the main concerns of the topic are [12]:

- How is an architectural blueprint or architectural solution created?
- In which parts of the enterprise are architectural blueprints or architectural standards used? Where are those standards breached?
- How can we improve conformance to architectural standards?
- Which applications or technologies can be used to achieve more conformity?

The relation between an architectural solution and an architectural blueprint is similar to the relation between a concrete class and its direct abstract parent class in programming paradigm. The desired technologies are suggested by an architectural blueprint in an abstract way. Hence, an architectural solution is a concretization of the architectural blueprint created by selecting specific technologies. For example, existence of a MySQL database in an architectural solution leads to the existence of a relational database in the corresponding architectural blueprint.
A Big Data framework in its essence resides on a large cluster of nodes. To access all the data in the enterprise, the framework needs to establish an underlying distributed file system to cover all the nodes. In other words, any data stored in the distributed file system can be questioned by the framework. These stored data can be ranged from single values in different file types to all tables in different databases and even data warehouses. Additionally, the framework provides some facilities to interact with external data stored outside its file system.
In this respect, Big Data technologies can be helpful in two different manners:

1) **Increasing flexibility:** Since a framework supports different kinds of technologies, it can be used to facilitate the interaction between them. For example, data stored in a relational database can be transformed and inserted in another non-relational database. Similarly, from an optimistic viewpoint, a single query can be forwarded to

all files and databases to gather all the results from different sources. Therefore, Big Data frameworks can be used to increase technology homogeneity of an enterprise.

2) **Analyzing the complexity:** Big Data frameworks should rely on programming languages for creating distributed file systems. These distributed file systems support different programming languages for their computations. Similar to Java, these programming languages generally have an ability to ask about constituent elements of the framework. In other words, Big Data frameworks have an ability to do self-questioning. The result of this inquiry can reflect the architectures and technologies used in the deployed framework. Therefore, the frameworks can be used to analyze the technology homogeneity. To be more precise, two methodology patterns (M-2 and M-4) and six viewpoint patterns (V-5, V-6, V-23, V-39, V-66, V-67) of EAM Pattern Catalog [12] can be supported in this way. All these patterns are interested to grasp holistic visions of the used technologies and their relations.

By comparing the Big Data frameworks noted in this article, we have noticed that Hadoop is a better choice in both ways. Architectures and technologies used in enterprises are not too dynamic. Most of their components are also stored in the file system with long lifetime. As a result, in an analyzing approach, it is better to do a more comprehensive investigation which could take some hours to some days.
Storm is not practical in this approach since it is generally used for analyzing too dynamic data on the fly. The main focus of this framework is real-time analytics. Analyzing data with short lifetimes like stock exchange rates is a more appropriate use case for this framework. In addition, it is not supported by large sets of marginal technologies to interact with other components in the enterprise architecture.
Spark does not seem a better alternative to Hadoop in this scenario. Although this framework is capable of doing a comprehensive analysis, it is not appropriate in two ways: cost and time. High speed memories are much more expensive than storage devices. Compared with Hadoop, comprehensive analysis takes more time, if the data is not present in nodes' memories (RDDs). Keeping architectural data in memory is not a reasonable approach, since we do not need them time to time. Additionally, this framework is newer than Hadoop; it does not have as many marginal technologies as Hadoop.
Therefore, Hadoop is a more suitable choice in this use case. It is more affordable than the others, since it can be deployed on cheap commodity hardware. Hadoop is also more scalable than Spark and Storm. Additionally, its own storage layer, HDFS, can be used by others. This feature enables enterprises to complement their needs with other computational frameworks such as Spark and Storm. Hence, it can be integrated seamlessly with both.
Traditionally in enterprises, more valuable data are stored on higher level of data sources such as relational databases, data warehouses, and non-relational databases. Such data sources keep data in a more structured way. Hadoop has a de facto standard technology, called Sqoop, to work with legacy RDBMS. It also has its own set of technologies to support data warehousing (Hive), data mining (Mahout) and metadata sharing (HCatalog) in a distributed manner. There are also some renowned technologies to store and query data in a non-relational form such as HBase (non-relational DBMS) and Pig (SQL-like query language).
Hadoop is an open-source framework maintained by Apache foundation, but there are also some third party distributions. These distributions also have their set of technologies to enrich the framework in different use cases. All these huge set of technologies can provide a solid foundation for enterprises to increase their technology homogeneity, either by getting more flexibility or better analysis.
Finally, providing lossless interactions between different Big Data technologies is still in progress. Practically speaking, Big Data frameworks and technologies are not a complete alternation for our legacy systems. For example, most enterprises still need RDBMS to handle relational data. However, the existence of such a database can be detected by using Big Data technologies. Even more if we use them to analyze the RDBMS log files, we can grasp some probable hidden information. As are result, we think it is more confident to stick the second approach, i.e. analyzing the complexity of the enterprise application landscape.

9. CONCLUSION

Big Data is that magical word spreading its usage to all aspects of our modern society. However, most of the times, we do not use these facilities in a greenfield approach. There are lots of enterprises with established structures interested in integrating Big Data technologies. Since we found the pattern-based approach as an appropriate solution for EAM, we decided to identify some patterns helping us with this integration.
Then, after clarifying our definition of Big Data and its difficulties, we moved on to identify renowned frameworks. Hadoop, Spark and Storm were distinguished as the three dominant open-source frameworks. Since their internal structures were opened, we have compared them regarding their natures.
Additionally, we have briefly looked at some tools to abstract the details of these frameworks. Considering, the importance of the extracted data value, deploying Self-Service BI applications seems to be a promising approach. Recently, this kind of tool gained increasing reputations within the spectrum of users with less IT-understanding.
Finally, after comparing our designated frameworks in an abstract way, we have restricted our use case to do a concrete comparison. The use case was about technology homogeneity. We identified two different approaches in this context. Then, we explained our reasons to prefer Hadoop rather than the others.

10. FUTURE WORKS

According to the experiment done by technical university of Munich, most management experts of enterprise architectures tend to express their ideas with examples. These examples are usually depicted as diagrams. That is why they provide viewpoint patterns in the EAM pattern catalog. The main focus of these patterns is to show statistical issues in a clear manner.
In Big Data context, there are also a huge sets of visualization tools which can help us to present statistical issues. These tools are getting more attention during the last years. That is why Big Data experts tend to divide the whole deployed solution in three different layers: storage, computation and visualization. Since the first two layers were briefly

discussed in this article, we think investigating the third layer will be a good candidate to pursue this article.

11. REFERENCES

[1] S. Buckl, A. M. Ernst, J. Lankes, K. Schneider, and C. M. Schweda. A pattern based approach for constructing enterprise architecture management information models. *Wirtschaftinformatik Proceedings 2007*, page 65, 2007.

[2] S. Buckl, A. M. Ernst, F. Matthes, and C. M. Schweda. Enterprise architecture management patterns for enterprise architecture visioning. In *EuroPLoP*, 2009.

[3] P. Chandarana and M. Vijayalakshmi. Big data analytics frameworks. In *Circuits, Systems, Communication and Information Technology Applications (CSCITA), 2014 International Conference on*, pages 430–434. IEEE, 2014.

[4] A. M. Ernst. *A pattern-based approach to enterprise architecture management.* PhD thesis, Citeseer, 2010.

[5] B. Evelson. The forrester wave: Agile business intelligence platforms, q3 2015, 2015.

[6] M. Ferguson. Architecting a big data platform for analytics. *A Whitepaper Prepared for IBM*, 2012.

[7] J. Han, M. Kamber, and J. Pei. *Data mining: concepts and techniques: concepts and techniques.* Elsevier, 2011.

[8] J. Hurwitz, A. Nugent, F. Halper, and M. Kaufman. *Big data for dummies.* John Wiley & Sons, 2013.

[9] C. Imhoff and C. White. Self-service business intelligence-empowering users to generate insights. *TDWI best practices report. On the TDWI site: www. tdwi. org*, 2011.

[10] A. Katal, M. Wazid, and R. Goudar. Big data: Issues, challenges, tools and good practices. In *Contemporary Computing (IC3), 2013 Sixth International Conference on*, pages 404–409. IEEE, 2013.

[11] D. Lyon. Surveillance, snowden, and big data: capacities, consequences, critique. *Big Data & Society*, 1(2), 2014.

[12] F. Matthes. Enterprise architecture management pattern catalog. *Technische Universität München, München*, 2008.

[13] V. Mayer-Schönberger and K. Cukier. *Big data: A revolution that will transform how we live, work, and think.* Houghton Mifflin Harcourt, 2013.

[14] M. Mestyán, T. Yasseri, and J. Kertész. Early prediction of movie box office success based on wikipedia activity big data. *PloS one*, 8(8):e71226, 2013.

[15] R. P. Padhy. Big data processing with hadoop-mapreduce in cloud systems. *International Journal of Cloud Computing and Services Science (IJ-CLOSER)*, 2(1):16–27, 2012.

[16] T. Version. 9, the open group architecture framework (togaf). *The Open Group*, 1, 2009.

[17] T. White. *Hadoop: The Definitive Guide, 4th Edition.* O'Reilly Media, Inc., 2015.

[18] K. Winter, S. Buckl, F. Matthes, and C. M. Schweda. Investigating the state-of-the-art in enterprise architecture management methods in literature and practice. *MCIS*, 90, 2010.

[19] W. Yang, X. Liu, L. Zhang, and L. T. Yang. Big data real-time processing based on storm. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2013 12th IEEE International Conference on*, pages 1784–1787. IEEE, 2013.

[20] J. Zachman et al. A framework for information systems architecture. *IBM systems journal*, 26(3):276–292, 1987.

[21] J. A. Zachman. Concepts of the framework for enterprise architecture. *Los Angels, CA*, 1996.

[22] J. A. Zachman. John zachman's concise definition of the zachman framework. *Zachman International*, 2008.

[23] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. Mccauley, M. Franklin, S. Shenker, and I. Stoica. Fast and interactive analytics over hadoop data with spark. *USENIX; login*, 37(4):45–51, 2012.

[24] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

Towards A Modularity-Based Technical Debt Prioritization Approach

Peter Sommerhoff
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
peter.sommerhoff@rwth-aachen.de

Muhammad Firdaus Harun
RWTH Aachen University
Software Construction
Ahornstr. 55
52074 Aachen, Germany
firdaus.harun@swc.rwth-aachen.de

ABSTRACT

Technical debt (TD) refers to aspects in software development that can have short-term benefits (e.g., faster time-to-market) but may be detrimental in the future (e.g., due to decreased software modifiability). TD management (TDM) deals with activities to control TD, deciding which debt to repay and which to defer, and monitor the effect of the incurred TD on business goals and productivity. Effective TDM requires prioritization of TD items in order to repay the debt with the strongest negative effects on business priorities and software quality first. This paper gives an overview of existing approaches for TD prioritization and aims to analyze commonalities and differences in order to extract prioritization rules and metrics. We focus on architecture, design and code debt that negatively impacts modularity and propose a prioritization approach for modularity-related TD.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.9 [**Software Engineering**]: Management—*cost estimation, time estimation, productivity*

Keywords

technical debt, technical debt prioritization, design debt, modularity

1. INTRODUCTION

Since Ward Cunningham coined the term technical debt (TD) in 1992, its usefulness to manage software projects more proactively and communicate to non-technical stakeholders has been widely appreciated [3, 13]. The metaphor refers to shortcuts taken during development which speed up development time in the short-run but hamper productivity and software quality in the long-run [12]. Typical reasons for intentionally incurred TD include business needs, tight deadlines, strict budgets, or customer requirements [13]. However, TD may also be incurred unintentionally, e.g., by a development team lacking the skills to implement a better solution [3], by acquisition of another company, or even due to indifference towards technical debt [12]. Several notions are associated with the TD metaphor, most importantly principal and interest. Principal refers to the total cost involved in repaying (i.e. fixing) a particular debt. In contrast, interest denotes the cost incurred over time due to the debt through decreased software quality and productivity [12]. Typically, principal and interest are measured for each TD item and then accumulated for a broader picture. A TD item is a particular debt such as a missing test case, an interface without proper documentation, or a code duplication.

Explicitly managing TD has several benefits: it is an effective way to manage software projects proactively [9], streamline refactoring activities [19], and monitor development progress [9]. Proactive management is supported by monitoring trends and acting upon them. For example, when a development team fails to write sufficient software tests, thereby incurring test debt, this problem can be identified and addressed early [9, 10]. Streamlined refactoring decisions are supported by TD prioritization methods which facilitate identifying high-impact debt [19]. Lastly, active tracking of TD over time allows to evaluate if TD is handled successfully or incurred uncontrollably, so that corrective action can be taken if needed [9].

Li et al. identified 8 TDM activities, one of which is TD prioritization. TD prioritization means ranking identified TD items according to a set of predefined rules, e.g., using a cost-benefit approach [12]. For example, in an object-oriented project, one TD item may be a class violating the Single Responsibility Principle. Another one may be lacking documentation for an interface. The first one may take more time to fix but probably also provide a bigger positive impact on the software quality – this depends on the organization's or the project's priorities [12].

Activities related to TDM which typically take place before prioritization are TD identification to identify intentional and unintentional debt, and TD measurement to estimate cost and benefit of TD items in one form or another. Further activities include TD monitoring to track changes in cost and benefit for TD items, TD repayment to eventually pay off selected TD items, and TD communication to raise awareness for TD among stakeholders [12].

TD prioritization is an essential part of the TDM process because it can be implemented in a way that it reflects not only technical, but also business priorities, and these ultimately take precedence in practice [3]. Several frameworks for TD prioritization have been introduced by researchers [9, 6]; however, Li et al. recently identified several open challenges concerning TD prioritization. These include lacking tool support, ways to prioritize TD to maximize benefit, and which factors to take into account for prioritization. [12].

Several approaches for TD prioritization have been introduced, most of which rely on source code and/or design metric measurements to prioritize TD [19, 9, 6]. All approaches use some way to measure the severity of a particular TD item (e.g., its interest) and the effort involved in fixing it (i.e. its principal). These values can then be used to prioritize the TD items. We will explain some approaches in more detail later.

The modularity-based approach presented in this paper is mainly concerned with design debt reflected in source code. In this regard, it is complementary to the SQALE method. To our knowledge, there has been no previous research on TD related to modularity in particular. We address this gap and present a framework to prioritize this type of debt.

The paper is structured as follows. In section 2, we provide the necessary background and analyze previous approaches to identify commonalities, differences, and limitations. Section 3 presents the modularity metrics identified in this paper and provides a brief rationale for each. In section 4, we demonstrate a TD prioritization approach based on the introduced metrics. Section 5 concludes this research by mentioning limitations of our approach and future work.

2. BACKGROUND

For the purposes of this paper, we will focus on architectural, design, and code debt. These correspond to three of the ten major types of TD identified by Li et al. in their recent mapping study [12]. We also note that we consider intentional and unintentional debt equally since both could be equally detrimental to software modularity.

TDM relies heavily on TD prioritization to streamline maintenance activities. Yet, of the 29 tools evaluated by Li et al., only the SQALE plugin for SonarQube and the Sonar TD plugin supported TD prioritization [12]. We will expand upon this area by addressing modularity as a subtopic and hope to encourage similar research in other niche topics which can serve as a basis for a more comprehensive framework in the future.

2.1 Existing TD Prioritization Approaches

Previous research has developed several TD prioritization approaches which exhibit many commonalities but also some foundational differences.

2.1.1 Finance-based Approach

Guo and Seaman proposed a portfolio approach for TD prioritization [6]. This approach offers a new perspective on TDM by reusing established portfolio management strategies from the finance domain. Here, TD is considered an investment and TD items are considered assets of a portfolio. Portfolio management aims to reduce risk and maximize return on investment (with TD being the investment). This maps nicely to the issue of TD prioritization. For this approach, important characteristics of each TD item (such as principal, interest amount, interest standard deviation, and correlations between TD items) must be estimated and documented. After that, a portfolio management model such as the Modern Portfolio Theory model is used to extract the best assets (i.e., TD items). The result is a subset of the assets which reduces risk and maximizes return. Thus, TD items which are not part of this portfolio should be fixed first [6]. Guo and Seaman note that there are several incompatibilities between financial assets and TD items which need to be considered: In finance, assets are divisible, and their cost and interest are known in advance – both of which are not the case with TD. The approach suggests using quantitative methods for the estimations if possible, and qualitative methods otherwise. Also, the performance of this approach in practice was not tested [6].

2.1.2 Design Quality Prioritization Approach

Another prioritization approach which focuses on design debt in particular was presented by Zazworka et al. [19]. More specifically, their paper focuses on god classes. They present a cost-benefit approach to decide which god class to refactor first, based on three metrics: a) weighted method count; b) tight class cohesion; and c) access to foreign data. For each metric, an acceptable threshold is defined. Zazworka et al. note that one can assume the cost of refactoring to increase with the distance of the class from the threshold. For instance, a class with 500 methods will typically take more effort to refactor than one with 50 methods. This argument is made for each of the three metrics. Change history data are used to estimate change likelihood as well as defect likelihood for each class. The distance of a class from the thresholds then defines the cost whereas the change and defect likelihoods define the benefit of refactoring the god class. A cost-benefit matrix can be used in order to give an effective overview on which god classes to prioritize [19].

2.1.3 Metrics-based Approach

The SQALE method is a metrics-based approach which addresses eight so-called characteristics which are divided into sub-characteristics and eventually source code requirements. The used characteristics are ordered by the time they become important in the file lifecycle. They are, in order, testability, reliability, changeability, efficiency, security, maintainability, portability, and reusability [9]. Requirements represent the definition of "right code", e.g., no commented out code, no interfaces without documentation, or no public class attributes. Note that the modularity characteristic is related to changeability, maintainability and reusability. In fact, Li et al. categorized modularity as a sub-attribute of maintainability, adhering to ISO 25010 [12]. However, the SQALE approach does not explicitly address modularity and is restricted to only code-related debt [9, 10]. In order to estimate the amount of TD, SQALE requires a remediation function for each requirement. This represents the estimated time needed to fix a TD item (the principal), e.g., 30 minutes to add documentation to an interface. Similarly, each TD item must be associated with a non-remediation function – the estimated cost for not repaying the debt [10]. These are simply accumulated to represent the TD for modules, subsystems or the whole system. The SQALE method also contains ways to visualize this TD, such as the Rating Grid, the SQALE Pyramid, and the Debt Map [9, 10].

2.2 Analysis of Existing Approaches

We will now evaluate commonalities, differences, and limitations of the aforementioned prioritization approaches.

2.2.1 *Commonalities*

All the approaches **exhibit some definition of cost and benefit** which is used for prioritization. In the SQALE approach, the remediation and non-remediation functions correspond to principal and interest, respectively. Thus they both represent costs. However, the non-remediation functions also represent the benefit achieved when repaying the debt. In the design debt prioritization approach introduced by Zazworka et al., the distance of a class from the threshold can be seen as a cost (the principal for fixing the item) while change and defect likelihood act as a measure of benefit. In the portfolio approach, interest cost is explicitly defined as estimated interest amount and interest standard deviation as part of the TD items. Principal is also an explicit part of each TD item (equivalent to the remediation function in SQALE) [6]. Benefit is not documented explicitly but can be derived from the interest measures.

Another commonality between the SQALE method and the design debt prioritization method is that **both rely on metrics as a basis for measuring and prioritizing TD**. In SQALE, the requirements defined by the organization rely on metrics in order to evaluate adherence to these requirements. For example, a requirement like "method should have no more than 100 LOC" would rely on a metric such as "lines of code in method". SonarQube is made to provide such metrics which allows the SQALE method to be implemented effectively in SonarQube. Similarly, the god class prioritization is explicitly based on the metrics weighted method count, tight class cohesion, and access to foreign data to identify and measure the severity of each god class. Defect likelihood and change likelihood are two more metrics used to estimate the relevance and severity of the debt.

Additionally, the SQALE method and the god class prioritization approach both include **visual representations to aid the decision process**. The portfolio approach could be extended accordingly; however, the approach has yet to be refined and tested in practice. Especially in the SQALE method, visualizations play an important role. Various visualizations have been introduced, including the Rating Grid, the SQALE Pyramid, and the Debt Map [9, 10]. Accordingly, the SonarQube plugin implements such visualizations. One can assume that these support both understanding and decision-making.

2.2.2 *Differences*

The unique feature of the portfolio approach is that it reuses existing knowledge from the finance domain. This is potentially an important advantage because portfolio approaches from finance are well-understood and established. Other approaches usually rely on previous research in fields such as software design, software quality measurement, and coding best practices. For instance, the design debt prioritization approach relies heavily on previous research on god classes which is a well-known issue in software design and software evolution.

The SQALE method is unique in that it has already been widely adopted in the industry through its implementation in SonarQube. To our knowledge, no other tool for TDM is used as much in practice. In fact, there is no research on the real-world implementation of the portfolio approach.

The presented comparison also highlights the fact that, while most approaches make use of predefined metrics, other approaches such as the portfolio approach are also conceivable. These may pose a good opportunity for research since previous research has focused on metrics-based methods. The approach presented in this paper will rely on metrics as well.

2.2.3 *Limitations*

The design debt prioritization approach introduced by Zazworka et al. is obviously limited to god classes in the form presented in the paper [19]. However, the principles may be reused for similar design flaws. For example, a similar approach for improper inheritance structures may use metrics such as "number of child classes", "depth of inheritance tree", or "composition not preferred over inheritance". Another limitation of the approach as presented in the paper is the fact that it relies on historical data to estimate change and defect likelihood. Thus, the quality of the estimate depends on the amount of available history, making it less applicable for newer projects.

With their portfolio approach, Guo and Seaman provide a new perspective on TDM. However, the implementation of this method in practice, its assumptions, conditions, and applicability remain to be evaluated when applied in the context of TD. They also mention some general guidelines based on finance which need further evaluation. For example, Guo and Seaman propose to prefer many small TD items over one big TD item – this diversification promises to decrease risk [6]. Similarly, one should prefer TD items with low positive correlations. Guo and Seaman proposed further studies for empirical evidence [6].

The SQALE method is limited to code-related debt which is only one of the ten types of TD [12]. However, other types of TD such as design debt or test debt are typically associated with the code debt. Thus, fixing code debt can also mitigate other types of debt. Also, we identified that the SQALE method lacks an explicit measurement of modularity. The characteristics, sub-characteristics, and requirements of the method provide an opportunity to tweak and/or extend it. Thus, the method could by extended by either adding modularity as a characteristic or as a sub-characteristic of maintainability, as in ISO 25010 [12]. The results of this paper can be used to add modularity requirements and are insofar complementary to the SQALE approach.

3. MODULARITY METRICS

The modularity of a software system refers to the capability of its modules and subsystems to function as autonomous modules and provide their services outside the original system [1]. The major benefit of such modularity is the option to substitute system components if a superior implementation becomes available. Since this option is available but not obligatory, and potentially improves system design, it provides a positive net value [17].

3.1 Existing Modularity Metrics

In order to measure and evaluate modularity, various researchers have gathered a catalog of modularity metrics. In 2007, Sant et al. presented 11 architectural metrics to measure modularity [15]. They are based on the principles that a single concern should typically be realized by a single component, that shared data and state between components should be minimized, and that the complexity of components should be reasonable. Additionally, Li and Henry gathered maintainability metrics, some of which can be applied for modularity concerns [11]. These include depth of inheritance tree (DIT), lack of cohesion of methods (LCOM), and number of child classes (NOC). A rationale for each is given later.

Another way to discover modularity requirements is by looking at research on modularity violations. Wong et al. presented the `CLIO` tool which detects such violations by comparing which components should change together and which did change together according to version control history [18]. This indicates the concept that architectural and design metrics should be mapped to measurable code metrics if possible in order to evaluate consistency with the architecture and design.

Metrics for modularity can be mapped to source code requirements, equivalent to those introduced in the SQALE method. By identifying and prioritizing such metrics, we will present a prioritization approach for modularity-related TD which can be integrated into the SQALE method if desired.

3.2 A Catalog of Modularity Metrics

The modularity debt prioritization approach introduced in this paper, like many others, is based on metrics. We present a catalog of metrics in Table 1. Typically, architectural metrics imply design metrics which in turn imply code metrics. Note that the derivation of metrics ends on the design level if they are measurable on that level already; however, most of our metrics map directly to source code requirements. In order to discover the modularity metrics, we relied on previous research on architectural best practices, modularity issues, and previously presented metrics. Most predominantly, we derived design metrics from architectural practices and then derived corresponding source code metrics. In most cases, this yielded well-known metrics from previous research.

Note that there are two trends here. First, the derived metrics often correspond to good design or coding practices. Second, architectural metrics tend to imply several design metrics which in turn tend to imply several code metrics. This makes the methodology very fruitful to derive a range of source code metrics.

3.3 Rationale

Table 1 shows all metrics identified in this paper and how they are derived from each other. We will give a brief rationale for each to explain in which ways they support modularity. References to other work are given where applicable.

3.3.1 Low Coupling Between Modules

Low coupling between modules is a major architectural requirement for modularity and reusability [1]. Based on this principle, we derived several design requirements and metrics. First, components should communicate via well-defined interfaces. Thus, on code level, developers should always refer to the most general type possible [2] in order to abstract from the concrete subtype and implementation. Next, associations and, more strongly, compositions introduce dependencies between components and should therefore be minimized. On code level, this can be measured by the number of imported types. Also, the message passing coupling can be measured by the number of method calls on other classes [11]. Next, the use of intermediaries decouples components by adding a layer of abstraction for communication. Such patterns include Facade, Mediator, Proxy, Strategy, Factory, Publish-Subscribe, and Blackboard [1]. Note that these are architectural and design-level patterns, yet could be measured semi-reliably on source code level by relying on naming conventions or stereotypes. A strong form of coupling is inheritance [11] which implies two design-level metrics, the number of parent classes (i.e., the depth in the inheritance tree in single inheritance languages), and the number of child classes (NOC). Arguably, inheritance couples components stronger than association [2].

3.3.2 Proper Distribution of Functionality

Second, proper distribution of functionality is a well-known challenge for software architects [1]. Design-level requirements derived from this include tight class cohesion (TCC) [14] and proper reuse of functionality. Tight class cohesion can be measured on code level by clusters of methods that share no common data at all (LCOM) [11]; the overall number of lines of code (LOC) may also be indicative of the cohesion since very large classes tend to handle various concerns [19]. The proper reuse of functionality is reflected by the lines of duplicated code, corresponding to the don't-repeat-yourself principle of object-oriented design.

3.3.3 Information-Hiding Interfaces

Third, we chose the use of information-hiding interfaces [17] as a separate architectural requirement since it yields several design and coding guidelines. One metric is the number of interfaces in comparison to the number of classes to estimate how widely information hiding is employed. This is related to "communication via interfaces" above. Another important metric is the number of public class attributes [4, 19] (excluding explicit class constants) since these violate encapsulation. More generally, we propose to measure the percentages of private, protected, and public attributes in a class (the naming convention is based on Java and similar languages). This allows to estimate the strength of encapsulation in more detail. For instance, private attributes can reduce inheritance coupling because even subtypes cannot access private attributes.

4. MODULARITY-BASED TD PRIORITIZATION APPROACH

4.1 Overview

Based on the presented catalog of modularity metrics, different prioritization strategies may be defined. In the following, we present a cost-benefit approach which balances principal against interest amount and probability – similar to remediation and non-remediation functions respectively. We note that various other prioritization strategies may be defined based on these same metrics.

In order to derive concrete TD items from the metrics, we must define a threshold based on our definition of "right code". For example, we may specify the threshold for the depth of a class in the inheritance tree to be lower than five. Then, in order to prioritize the TD items, we assign principal, estimated interest amount (EIA), and estimated interest

Architecture level	**Design level**	**Code level**
Low coupling between modules	Communication via interfaces [2]	Number of type references replaceable by more general type [2]
	Number of associations to other classes (FAN OUT [4])	Number of imported types [19]
		Number of method calls on other classes [11]
	Number of associations from other classes to this class (FAN IN) [7]	Number of imports of this class in other classes
		Number of classes calling methods from this class
	Number of intermediaries	Number of usages of Facade, Mediator, Proxy, Strategy, Factory, Publish-Subscribe, Blackboard and similar patterns [1]
	Number of parent classes / Inheritance depth (DIT) [11]	
	Number of child classes (NOC) [11]	
Proper distribution of functionality	Tight class cohesion (TCC) [19, 4]	Number of clusters of methods without a shared variable (LCOM) [11]
		Source lines of codes in class (LOC) [11, 4]
		Weighted method count (WMC) [11, 19]
	Proper reuse of functionality [19]	Lines of duplicated code (SEC) [8]
Components have information-hiding interfaces	Percentage of interfaces vs. classes	Number of classes
		Number of interfaces
	Number of public class attributes (excluding constants) [4, 19]	Number of private attributes
		Number of protected attributes
		Number of public attributes (NOPA) [4]

Table 1: A catalog of modularity requirements and metrics.

probability (EIP). This is not a trivial task because the estimations depend on organizational and technical factors [4], including developer skills, interdependencies between TD items, and projected future changes. Therefore, the estimations must be performed by each organization and project individually. Principal, EIA, and EIP have all been used in previous research [6, 16]. The principal defines the cost for fixing the TD item and thus corresponds to a remediation function in the SQALE approach. Similarly, interest amount and probability define the cost for not fixing the TD items and thus correspond a non-remediation function. The rationale behind the probability is that modules which will likely never be changed in the future should have accordingly low priority [5].

4.2 Cost-Benefit Formula

Once we have assigned principal, EIA, and EIP, we can use these to prioritize the TD items. For the purposes of this paper, we will assign to each TD item I the priority $P(I)$ calculated as

$$P(I) = \frac{\text{EIA} \cdot \text{EIP}}{\text{Principal}}.$$

This is a very simple way to assign a useful priority score which represents a cost-benefit approach – dividing benefit by cost. Thus, the TD items I with the highest priority $P(I)$ should be repaid first.

4.3 Procedure

To use this approach, several steps and guidelines should be followed. A prerequisite is the ability to measure code-related debt. Ideally, design debt can be measured as well. The procedure is as follows:

1. Assign a threshold to each metric to derive TD items.
2. Estimate principal, EIA, and EIP for each TD item.
3. Calculate $P(I)$ for each item.

For step 1, Fontana et al. provide some guidance in a recent study [4]. However, thresholds must be defined and evaluated by the organization based on what works in their context. For step 2, you should take into account your development team's skills, organizational constraints, and historical data to improve the estimations [16, 19]. As mentioned above, this is a complex task due to many influencing factors. Since the quality of the approach depends directly on the quality of the estimations, we recommend monitoring

TD Item	Metric	Threshold	Principal	EIA	EIP	P(I)	Rank
3 public attributes in `SampleClass`	NOPA	0	5h	3h	20%	0.12	2
4 clusters in `Another-Class`	LCOM	2	20h	10h	50%	0.25	1
Depth of `SampleClass` is 7	DIT	5	25h	5h	25%	0.05	3

Table 2: List of sample TD items with threshold, principal, and interest.

and adjusting the estimations. To identify the TD items and measure their related metrics, you may use an analysis tool such as SonarQube.

4.4 Applying the Approach

To illustrate the presented approach and make it more tangible, we present a brief example. We assume we have derived the TD items listed in Table 2.

The example is set up in a way that `AnotherClass` is changed much more frequently than `SomeClass`, thus the higher EIP. We can see that the debt related to `AnotherClass` should be repaid first because $P(I)$ is higher than that of any other TD item. This originates from the fact that `AnotherClass` is changed often and thus has high interest. The next TD item to fix would be the first one because its principal is low compared to that of the third item.

5. CONCLUSION & FUTURE WORK

In this paper, we have introduced a modularity-based TD prioritization approach based on several metrics we extracted or derived from previous research.

In its current state, the presented approach lacks a way to systematically estimate and assign values such as principal, estimated interest amount (EIA), and estimated interest probability (EIP) which is not a trivial issue in TDM. Like other current approaches, we rely on user input for these estimations [9, 6]. Tools to guide the estimation of such values remain future work. Such tools should also consider the severity of violations, e.g., by measuring the distance from the defined thresholds [19] or other user-defined metrics. We also note that we do not consider the presented metrics fixed and expect future work to extend or refine the catalog. In addition, future research focusing on niches other than modularity may provide further insights which can pave the way to a comprehensive TDM approach. Also, other priority calculations may be defined based on the provided metrics and resulting TD items. The implementation of the presented approach in practice remains to be evaluated to generate empirical data.

6. REFERENCES

[1] L. Bass. *Software architecture in practice.* Pearson Education India, 2007.

[2] J. Bloch. *Effective Java.* Java Series. Pearson Education, 2008.

[3] F. Buschmann. To pay or not to pay technical debt. *Software, IEEE*, 28(6):29–31, 2011.

[4] F. A. Fontana, V. Ferme, M. Zanoni, and R. Roveda. Towards a prioritization of code debt: A code smell intensity index. In *Managing Technical Debt (MTD), 2015 IEEE 7th International Workshop on*, pages 16–24. IEEE, 2015.

[5] G. Technical debt: Strategies & tactics for avoiding & removing it. http://blogs.ripple-rock.com/SteveGarnett/2013/03/05/TechnicalDebtStrategiesTacticsForAvoidingRemovingIt.aspx. Accessed: 2015-12-01.

[6] Y. Guo and C. Seaman. A portfolio approach to technical debt management. In *Proceedings of the 2nd Workshop on Managing Technical Debt*, pages 31–34. ACM, 2011.

[7] S. Henry and D. Kafura. Software structure metrics based on information flow. *Software Engineering, IEEE Transactions on*, SE-7(5):510–518, Sept 1981.

[8] M. Lanza and R. Marinescu. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems.* Springer Science & Business Media, 2007.

[9] J.-L. Letouzey. The sqale method for evaluating technical debt. In *Proceedings of the Third International Workshop on Managing Technical Debt*, pages 31–36. IEEE Press, 2012.

[10] J.-L. Letouzey and M. Ilkiewicz. Managing technical debt with the sqale method. *IEEE Software*, 29(6):44–51, 2012.

[11] W. Li and S. Henry. Object-oriented metrics that predict maintainability. *Journal of systems and software*, 23(2):111–122, 1993.

[12] Z. Li, P. Avgeriou, and P. Liang. A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, 101:193–220, 2015.

[13] E. Lim, N. Taksande, and C. Seaman. A balancing act: what software practitioners have to say about technical debt. *Software, IEEE*, 29(6):22–27, 2012.

[14] R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 350–359. IEEE, 2004.

[15] C. Sant Anna, E. Figueiredo, A. Garcia, and C. J. Lucena. On the modularity of software architectures: A concern-driven measurement framework. In *Software Architecture*, pages 207–224. Springer, 2007.

[16] C. Seaman and Y. Guo. Measuring and monitoring technical debt. *Advances in Computers*, 82:25–46, 2011.

[17] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen. The structure and value of modularity in software design. *SIGSOFT Softw. Eng. Notes*, 26(5):99–108, Sept. 2001.

[18] S. Wong, Y. Cai, M. Kim, and M. Dalton. Detecting software modularity violations. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 411–420. ACM, 2011.

[19] N. Zazworka, C. Seaman, and F. Shull. Prioritizing design debt investment opportunities. In *Proceedings of the 2nd Workshop on Managing Technical Debt*, pages 39–42. ACM, 2011.

The Impact of Context on Continuous Delivery

Sebastian Rabenhorst
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
sebastian.rabenhorst@rwth-aachen.de

Andreas Steffens
RWTH Aachen University
Software Construction
Ahornstr. 55
52074 Aachen, Germany
steffens@swc.rwth-aachen.de

ABSTRACT

This paper will evaluate how the properties of production environment and software, which is continuously delivered, have influence on the implementation of *Continuous Delivery*. The evaluation is based on three case studies from different software development domains. The first case study deals with the way the software engineers at *Etsy* use *Continuous Integration* for the delivery of their *App*. The second example is about *Box's* decision to introduce *Continuous Deployment* in order to continuously deploy their desktop software *Box Sync* to its customers. The last example is about the *Hewlett-Packard LaserJet Firmware Team* which implemented *Continuous Delivery* with great success.
These case studies will show that UI (user interface) complexity, the lack of control over the production environment and the quality of software simulators, which simulate the production environments, are properties or derived properties which have impact on the implementation of *Continuous Delivery*.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

Keywords

Continuous Delivery, Production Environment

1. INTRODUCTION

Continuous Delivery is a software development discipline which enables "Reliable Software Releases through Build, Test, and Deployment Automation"[9].
This paper will evaluate the impact of context on *Continuous Delivery* implementation for software development domains which are different from the classical domains of none UI heavy backend and web applications. Software and production environment properties present the context evaluated in this paper. The evaluation is based on three different case studies.
The structure of the paper is as follows. Section 2 provides all important definitions and a short introduction to *Continuous Delivery*. Section 3 covers the three case studies, which are first described and then analyzed. The first case study is about the way *Etsy*, "often trotted out as a poster child for Devops" [3], introduced *Continuous Delivery* for their *Apps* (mobile applications). The second one comes from the company *Box* which recently implemented *Continuous Deployment* for their desktop software *Box Sync*. The last example is the very well documented case of the *HP LaserJet* Firmware Team, which increased their productivity dramatically by implementing *Continuous Delivery*.
After the analysis and description of all three examples they are compared to each other in section 4 in order to identify similarities of properties which impact the *Continuous Delivery* implementation. The last section 5 will conclude the paper and provide ideas for future work.

2. TERM DEFINITIONS

In this section the most important terms are briefly introduced. More detailed information can be found in the cited sources.

Continuous Delivery and the Deployment Pipeline

The three terms *Continuous Integration*, *Continuous Delivery* and *Continuous Deployment* and their coherences are easily confused and will therefore be defined in this section. The process of *Continuous Delivery* starts with continuously integrating code. **Continuous Integration** "is a software development practice where members of a team integrate their work frequently, [...]. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible." **Continuous Delivery** goes one step further and ensures that "software is build in such a way that the software can be released to production at any time" and the software is deployable throughout its whole life cycle. The last step towards complete automation is **Continuous Deployment**. "*Continuous Deployment* means that every change goes through the pipeline and automatically gets put into production, resulting in many production deployments every day." [12]
The Core of a *Continuous Delivery* implementation is the **Deployment Pipeline**, which models the process of getting "software from version control into the hands of your users" [7]. As figure 1, which illustrates a basic deployment pipeline, shows some stages of a *Deployment Pipeline* are au-

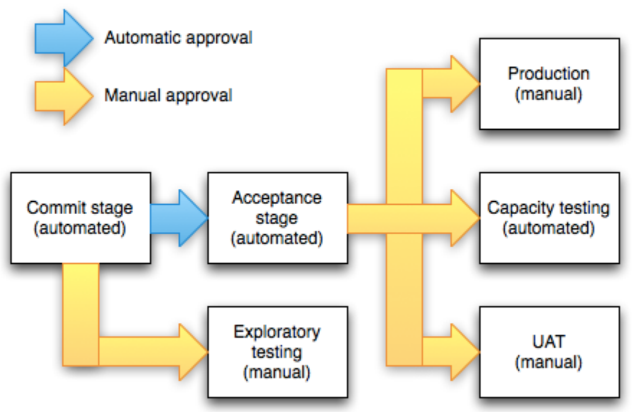


Figure 1: Basic *Deployment Pipeline*[8]

tomated and some need manual interaction. The three case studies in section 3 will show that the level of automation is different for each specific implementation. The first stage is the commit stage. The following two stages, automated acceptance tests and a manual testing stage, can be executed in parallel. The last three stages are parallel again and consist of the UAT (user acceptance tests), capacity tests and going into production stage [8].
The term *classical Continuous Delivery* refers to *Continuous Delivery* for web or backend applications with none or very little UI.

Environment

An **environment** in the context of software development is one specific combination of hardware properties and software properties which build a platform to run software on. Through the *Continuous Delivery* process the following environments may occur.
In traditional software development there are four different environments for a software from development to production. The first one is the **development environment**, which represents the working environment of the developer. After the development environment comes the **integration environment** where the code changes from all developers are combined and integrated. For smaller projects the first two environments could be the same. The **staging environment** should be as similar as possible, ideally identical, to the production environment. It is used to simulate production. The **production environment** is the environment the software was developed for [14].

3. CASE STUDIES

This section covers the three case studies on which the evaluation in section 4 is based on. Each case is briefly described and followed by an analysis. The goal of the analysis is to find out which properties of the production environment and the software had most impact on the specific implementation of *Continuous Delivery*. The analysis part itself will focus on the following two questions:

- Which properties of the production environment and the software have influence on *Continuous Delivery*?
- How is this reflected in the implementation of *Continuous Delivery*?

The results of the analyses are compared to each other in section 4 to identify similarities and determine what had the biggest impact regarding difficulties and unresolved problems.

3.1 Mobile Application: Etsy

The first case study is taken from an article which describes how *Continuous Delivery* is implemented for an *App* at *Etsy*. "*Etsy* is a marketplace where people around the world connect, both online and offline, to make, sell and buy unique goods." [4] The example was chosen since it points out the challenges of *Continuous Delivery* in the context of an *App*. The article mainly focuses on the *iOS Continuous Delivery* stack because at the time of writing the *Android* stack wasn't as well developed as its *iOS* counterpart [13].

Description

Etsy decided to use *Continuous Integration* since "through *Continuous Integration*, they can detect and fix major defects in the development and validation phase of the project, before they negatively impact user experience".
Automated *Continuous Delivery* at Etsy for mobile apps can be summarized in one sentence: "Every commit builds the mainline on special integration machines". So after every commit by a developer an integration server (*Jenkins* [10]) executes a build plan which consists of more than 15 jobs by using the integration machines and notifies the developers in case of a failure. There is also a simple homegrown dashboard which "communicates the current test status across all configurations" [15]. The whole *CI* infrastructure from *Etsy* is illustrated in figure 2.
The biggest challenge was to setup an integration and test environment which covers all important devices. For iOS every build has to be tested on "seven different iPads, five iPhones and a few iPods" [15]. But for *Android* it is even worse because because the number of *Android* devices to cover by tests is overwhelming. As integration and test environment there is a "fleet of *Mac minis*" which are all nearly fully automatically provisioned. Additionally real devices in the cloud from *AWS device farm* [1] are used for testing. The setup of the integration machines could not be fully automated because of the "inability to automate the installation of some software dependencies". Especially the installation and setup of the *iOS* IDE *Xcode* still needs some manual interaction.
With these integration machines the code is build and tested after each push to the repository to get immediate feedback. Regression test are run nightly on a broader range of real devices [15].
Since "most of the core logic of *Etsy's Apps* relies on the UI layer" the software engineers at Etsy focus on functional testing which mimics the steps of an actual user. The test include actions like "searching for listings and shops", "registering new accounts" and "purchasing an item with a credit card or a gift card". One example for a concrete functional tests is the *checkout test*. For this test a buyer and seller test account is created and a real credit card is used [15]. The test is as follows:

1. "Signing in to the app with a test buyer account." [15]
2. "Searching for an item (in the seller test account shop)." [15]

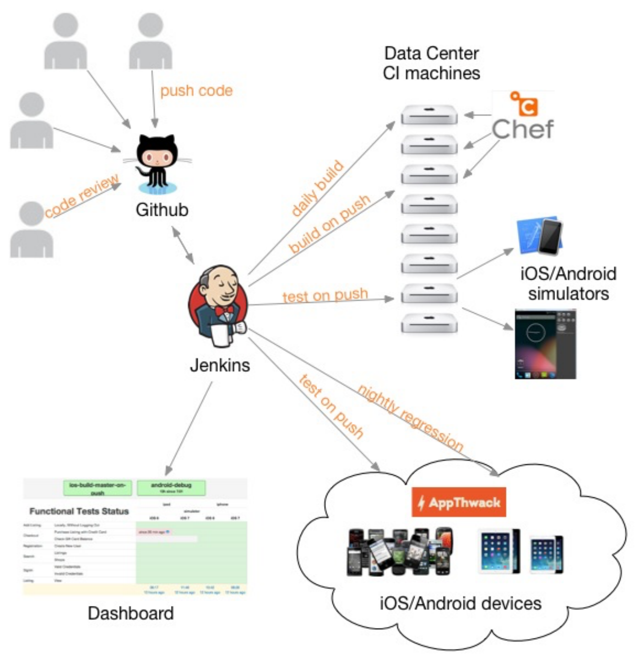


Figure 2: Etsy's *CI* infrastructure overview[15]

3. "Adding it to the cart." [15]

4. "Paying for the item using the prepaid credit card." [15]

These tests run on simulators and real devices. Integration tests, executed after every push, are run in simulators on the integration machines of *Etsy*. Nightly Regression tests are outsourced to *AWS device farm* [1] (previously known as *Appthwack*), which provides the possibility to test *Apps* on a broad range of real mobile devices [15]. Since it is nearly impossible to test on all possible device configurations, devices are chosen based on *Google Analytics* data. Since the integration happened only recently there were still some problems related to testing on physical devices and the challenges of aggregating and reporting test status from all the devices when the article was published [13].

In addition to the automatic integration testing there are layers of manual QA (quality assurance). An internal build is released daily which Etsy employees are encouraged to install on their devices [15]. Another manual test is called "app rotations": "Eight volunteers gather in a room, accompanied by a QA facilitator and a mix of devices. The goal is to find as many bugs as possible in a predefined timebox." [13]
After all automated and manual tests are passed the *App* is submitted to the *App Store* for approval which will take around five days [15]. So if a bug slips through all the tests and is discovered while the *App* is already running on the users devices, it takes a minimum five days to get an update to the users.

Analysis

The goal of the software engineers at *Etsy* was to implement fully automated *Continuous Delivery* for their *Apps*. They automated the process as far as possible from pushing the code into the repository to submitting the *App* to the *App Store*. During the implementation of *Continuous Delivery* they were faced with two major challenges.
The first challenge was the setup of the environments for integration and testing. For testing the *Android* and *iOS Apps*, either a software simulators or real devices is necessary to run the *Apps* on. Simulators for *Android* and *iOS* don't model real devices closely enough and proved to be insufficient [17], thus simulators are only used for integration tests and real devices have to be used for regression tests. So the first property with influence on the *Continuous Delivery* implementation is the inability to properly model mobile devices with software simulators.
The need to use real devices for tests results in the next problem. Which devices should be used for the tests? The possible production environments are all *iOS* and *Android* devices with an *App Store* or *Google Play Store* installed. While for *iOS* there is a limited number of different devices and versions, the number of *Android* devices and versions is far to big to run tests on all of them, because that would lead to unmanageable number of devices to run tests on. *Opensignal* reports there are over "24,000 distinct *Android* devices seen in 2015" [16]. So it's impossible to cover all existing devices with tests and a specific set of *iOS* and *Android* has to be chosen for tests. Thus the second property with influence on the implementation of *Continuous Delivery* is a to big variety of possible production environments which have to be covered with tests.
The second challenge was the automated testing of the UI. The functional tests used for this purpose will only discover basic bugs and *App* crashes. Additionally the aggregation and evaluation of the test data from the *AWS device farm* devices used for regression testing is still an unsolved problem, which is again a result of the variety of possible production environments. These problems made it necessary to add the described manual QA stages. So one identified property of the production environment which has impact on *Continuous Delivery* is UI complexity. It results in the inability to fully automate tests, which makes manual quality assurance necessary.
Another property is control over the deployment process. The five days approval process of the *App Store* was one reason for an additional manual test stage. So the inability to deploy a hotfix immediately results in even more accurate testing.

3.2 Desktop Application: Box

Box is a company which provides "secure content and online file sharing for businesses" [2]. One part of their product is their desktop software *Box Sync* which syncs their customers desktop computers with *Box*'s online services. "In an effort to maintain the agility of our startup days and deliver the best software possible, Box has been moving towards *Continuous Deployment*". Since the software engineers at *Box* had huge success with *Continuous Deployment* and web development they decided to use their experience and knowledge and adapt it for their desktop software [18].

Description

"In order to do *Continous Deployment* you must be doing *Continous Delivery*" [12]. Therefore the team at *Box* implemented "automated acceptance testing" in a first step. Basic functionality of *Box Sync*, syncing files from one computer to another, is easy to test since network and file system could easily be simulated [18]. They used standard best practice for web development:

1. "Every time a developer pushes a new commit, the application is built in its entirety ("continuous integration") and the full suite of tests is run." [18]
2. "If a test fails, the build cannot be deployed and further commits are rejected until the test failure is fixed ("stop the line")." [18]

Since the *Box Sync* software "is light on UI and its basic job-ensuring two sets of files in two different places match-is very easy for a computer to verify" [18] there is full code coverage through unit tests. They have three types of automatic integration tests:

1. Full code coverage via unit tests.[18]
2. "The main syncing algorithm is covered by integration-style tests which simulate the network and file system called B to Y (the local file system is A, and the network is Z)." [18]
3. "Full-scale integration tests that launch the built version of Sync (the full .app or .exe, depending on platform), play with files on the local hard drive or on Box, and verify the right things end up in the right place at the end. They call this "chimp"." [18]

All of the described tests are run on each supported platform and operating system. Since the *Box Sync* software relies heavily on the *Box* web API, another suit of integration tests called "chimp-staging" is run to ensure compatibility. But "deploying client software is completely unlike deploying a web app, so their first goal was to make the process as consistent as possible, while respecting the different domain requirements and maintaining high user experience standards" [18].
As a result of *Continuous Deployment* of *Box Sync* all updates had to be backward compatible to a lot of prior versions since the production environment of *Box Sync* are desktop computers which might be offline for days or weeks. The risk of rendering a client useless with a failed update is too high and therefore older versions of *Box Sync* are manually updated with consecutive updates before the release of a new version [18].
To make sure that there are no problems during the automated deployment the *Box Sync* clients are monitored remotely and bad things like "exceptions, errors, or warnings the clients encounter" are reported. But also things like uploads, downloads, and authentication session renewals are monitored to assure that the clients don't stop working completely. All the data is aggregated by the client and send to the servers in a bandwidth saving manor to prevent *Denial-of-service attacks* by the own client [18].
But there are still three problems to be solved for real *Continuous Deployment*:

1. "Shipping a complete copy of the application multiple times a day would saturate bandwidth. Differential updates could solve this problem." [18]
2. The UI elements of *Box Sync* are still checked manually for each platform [18].
3. The reading of feedback from the clients is not automated [18].

The author of the article summarizes the implementation of *Continuous Deployment* for *Box Sync* as follows: "One of the things we learned while building Box Sync is that even if we cannot reach true continuous deployment for technical reasons, having it as a goal makes a strong, positive impact on our culture and development practices." [18]

Analysis

The *Box* team had a lot of experience with *Continuous Deployment* for web applications and tried to apply their knowledge to the delivery and deployment of their desktop software *Box Sync*. This worked out very well for the *Continuous Integration* stages of their pipeline because the core functionality of *Box Sync* was easy to verify and the implementation was very similar to an implementation for classical *Continuous Integration*.
The regression test stage, which was executed on "real computers", however could only be partly automated. The core functionality was again easy to test since there was no complex UI and the result of an test could easily be verified automatically on real computers. UI tests in contrast were too complex for the *Box Sync* team to implement and therefore the UI is tested manually before each release. So again the UI couldn't be tested fully automated.
But the major challenge for the *Box Sync* team was to keep each release backward compatible to prior releases. This problem was a result of no control over the production environment since it's an decision of the customer when the client is online and can update itself. This is a big difference to web servers, the target environment for classical *Continuous Delivery*, which are fully owned and are mostly incrementally updated. This problem couldn't be solved with automated tests instead they had to add a manual approval stage.

3.3 Embedded System: HP Printer Firmware

The last case study is about the *HP LaserJet* Firmware Team which made their way out of a crisis and increased productivity by implementing *Continuous Delivery*. The whole process is very well documented in the book "A Practical Approach to Large-Scale Agile Development" [6] by the project leader Gary Gruver, which I recommend for further details. This case study was selected since it's completely different from the other two case studies and shows that fully automated *Continuous Delivery* is possible for software development domains different from web and backend.
When Gary Gruver joined the *HP LaserJet* Firmware Team they spent only 5 % of their resources on developing new features and the average time of one regression test cycle was six weeks. This is why they decided to implement *Continuous Delivery* and changed the architecture of their software. We will focus on the implementation of *Continuous Integration* as described in chapter 6 of Gruver's book [6].

Description

Before they implemented *Continuous Delivery* they had to change the structure of the code first. They reorganized their code base and changed from multiple branches, one

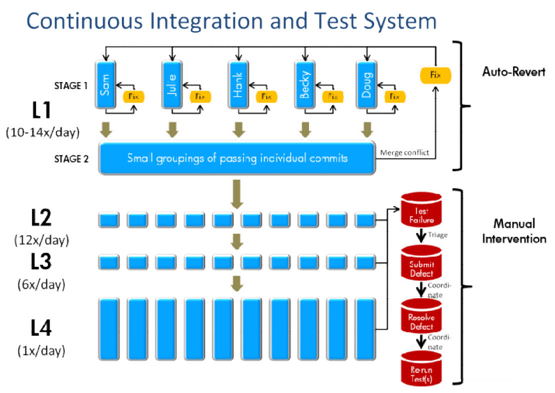


Figure 3: *Continuous Delivery* system at HP [6]

for each printer model, to one single branch. Instead of defining the specific capabilities for each printer with a *C #ifdef* directive [5] they used XML configuration files for the definition of the capabilities. For the integration tests they developed their own printer simulators and deployed them on 2000 virtual servers. For the later test stages they used hardware emulators to get more accurate results [11]. Figure 3 shows the *Continuous Integration* and test setup of the *HP LaserJet* Firmware Team. The system has four different levels of testing [6]:

- **L1**: Is executed after each commit. If there is a failure it will be automatically reverted.
- **L2**: More detailed tests, which run every 2 hours and use last working commit from *L1*. If a test fails an email with everything needed to replicate the failure is sent to the developer who committed the code.
- **L3**: Same as *L2* but runs on dedicated emulator hardware every 4 hours.
- **L4**: All automated tests are combined to a regression test suite and run daily around midnight. "Provides a complete view of the quality of the system" and is an indicator for the release readiness of the firmware.

With the introduction of *Continuous Delivery* the *HP LaserJet* Firmware Team could strikingly decrease the time and resources needed for code integration and tests. While the team spent 10 % of their resources for code integration before the changes now it's only 2 %. They could decrease the resources needed for testing from 15 % to 5 %. This allowed them to spend 40 % instead of 5 % on new features and innovations [6].

Analysis

The HP team also had the challenge of covering multiple production environments with tests. But in contrast to the other two case studies, their models of the simulators are very good and they have full control over each possible production environment. This way they could reach full coverage of all possible production environments. Additionally, since there was no complex UI, all the test data could be evaluated automatically and for some tests there was also automated feedback to the developers. But there was also the problem of software simulators not being good enough and therefore they used hardware simulators for the regression test stage.

4. EVALUATION

This section sums up and evaluates the results of the analysis parts in Section 3.

UI complexity

The first two case studies showed that the level of UI complexity of the software has a big influence on the degree of manual test stages required for *Continuous Delivery.* To reach full test coverage for an UI heavy software all possible input paths have to be covered and each result has to be verified. User Input can be simulated with the help of scripts. The problem is the automated aggregation and evaluation of the test data from all devices. The software engineers at *Etsy* can only detect crashes and low level bugs with automated UI tests. The UI of *Box Sync* is tested manually because implementing tests would be too complex. The case study from *HP*, in contrast, is a good example for software with very little UI and UI interaction of the user. As a result they could completely automate their tests.
Therefore UI complexity is one property of the software which has an impact on the implementation of *Continuous delivery.*

Lack of control over the production environment

The impact of lack of control over the production environment showed itself in three different variants.
The first one comes from the *Etsy* case study which showed that if there are no constraints on the configuration of the production environments, that could lead to a fragmentation of the production environment. This might result in an unmanageable number of possible production environments. This again results in the problem how to implement the integration and test environments, since it is impossible to run tests on all possible production environments. The *HP* case study in contrast shows that it is possible to cover all production environments with tests, even if there is a big number of them.
The second variant about lack of control over the production environments is the lack of control over when and how updates are deployed to the production environment. The *Box Sync* case shows that if you continuously deploy your software into production there might be problems because some clients skip updates and therefore updates have to be compatible to all prior versions. So the lack of control over the production environment could lead to an additional manual approval stage
The last one is a result of no control over the deployment process. Bugs that slip into production can't be immediately fixed with a hotfix. This makes it necessary to test several nightly builds manually before every release.
So the lack of control over the production environments has a lot of impact on all stages which are connected to tests.

Quality of software simulators

The *HP* and *Etsy* case studies showed that quality of software simulators, which simulate the production environment, have impact on the test stages of *Continuous Delivery*. Tests with simulators are mostly not sufficient since simulators are unable to properly imitate some properties of the production environment. Therefore the software running on simulators won't show the same behavior and performance as on the real devices. As a consequence, tests in simulators won't discover all bugs that are found with tests on real devices. For this reason in both case studies from *HP* and *Etsy* simulators are only used for early test stages. But with the use of hardware based simulators or real devices the aggregation and evaluation of test data is more complex. This results in more effort for the implementation of tests or even inability to process the data automatically.
So with decreasing quality of software emulators for an production environment the complexity of tests increases.

5. CONCLUSION & FUTURE WORK

The analysis of the case studies showed some of the possible impacts of production environment and software properties on the implementation of *Continuous Delivery*.
The following three properties were extracted from the analysis of the case studies:

- With increasing **UI complexity** the test data from UI tests can't be processed automatically and manual test stages are necessary.
- The **Lack of control** over parts or the whole of the production environment influences the implementation of the different test stages.
- **Quality of software simulators** is connected to complexity of the test stages.

Since this paper could only evaluate a limited number of case studies examples from other software development domains should be examined to confirm and expand the results.
One of the consequence of the found properties is the need for additional test stages which require manual interaction. Especially the evaluation and feedback for UI tests are done manually for two of the three case studies. In order to solve this problem further investigation of UI testing is necessary to identify the reasons which prevent the full automation.
The lack of control over the production environment combined with an unmanageable number of possible production environments made it impossible to reach full test coverage for them. There are two problems suitable for further investigation. The first one is how to prevent the fragmentation of a production environment which leads to an uncontrollable number of possible production environments. And if it can't be prevented how to maximize the coverage of relevant production environments.
The case studies also showed that if the production environment is fully under control of the team and UI complexity is low it's possible to implement fully automated *Continuous Delivery*.

6. REFERENCES

[1] Amazon. Aws device farm. `http://aws.amazon.com/device-farm/?nc1=f_ls`, 2015. Retrieved December 03, 2015.

[2] box. box.com. `https://www.box.com`, 2015. Retrieved December 06, 2015.

[3] L. Chen. Continuous delivery: Huge benefits, but challenges too. *Software, IEEE*, 32(2):50–54, Mar 2015.

[4] Etsy. About etsy. `https://www.etsy.com/de/about/`, 2015. Retrieved December 06, 2015.

[5] gnu.org. Ifdef, 2015. Retrieved December 16, 2015.

[6] G. Gruver, M. Young, and P. Fulghum. *A Practical Approach to Large-Scale Agile Development: How HP Transformed LaserJet FutureSmart Firmware*. Addison-Wesley Professional, 1st edition, 2012.

[7] J. Humble. Continuous delivery: Anatomy of the deployment pipeline. `http://www.informit.com/articles/article.aspx?p=1621865`, 2010. Retrieved December 11, 2015.

[8] J. Humble. Deployment pipeline anti-patterns. `http://continuousdelivery.com/2010/09/deployment-pipeline-anti-patterns/`, 2010. Retrieved December 12, 2015.

[9] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 1st edition, 2010.

[10] W. Jenkins. Meet jenkins, 2015. Retrieved December 15, 2015.

[11] G. Kim. The amazing devops transformation of the hp laserjet firmware team (gary gruver). `http://itrevolution.com/the-amazing-devops-transformation-of-the-hp-laserjet-firmware-team-gary-gruver/`, 2015. Retrieved December 06, 2015.

[12] F. Martin. Continuousdelivery. `http://martinfowler.com/bliki/ContinuousDelivery.html`, 2013. Retrieved November 22, 2015.

[13] J. Miranda. How etsy does continuous integration for mobile apps. `http://www.infoq.com/news/2014/11/continuous-integration-mobile`, 2014. Retrieved December 03, 2015.

[14] P. Murray. Traditional development/integration/staging/production practice for software development, 2006. Retrieved December 18, 2015.

[15] K. Nassim. Etsy's journey to continuous integration for mobile apps. `https://codeascraft.com/2014/02/28/etsys-journey-to-continuous-integration-for-mobile-apps/`, 2014. Retrieved November 22, 2015.

[16] opensignal. Android fragmentation visualized (august 2015). `http://opensignal.com/reports/2015/08/android-fragmentation/`, 2015. Retrieved December 19, 2015.

[17] M. Poschenrieder. Testing on emulators vs real devices, 2015. Retrieved December 17, 2015.

[18] B. Smith. Continuous deployment in desktop software. `https://www.box.com/blog/continuous-deployment-in-desktop-software/`, 2013. Retrieved December 03, 2015.

Categorization of application layer viewpoints in the EAM

Sinan Durmaz
sinan.durmaz@rwth-aachen.de

Simon Hacks
RWTH Aachen University
Software Construction
Ahornstr. 55
52074 Aachen, Germany
simon.hacks@swc.rwth-aachen.de

ABSTRACT

Since the last century companies are getting bigger and bigger. Thus they have a more complicated structure and are difficult to manage. Enterprise architecture models help to provide better transparency and a clear view for all involved parties. These IT-based systems are good for visualizing business processes of a company. For the representation are among others methods of the software cartography used. In the software cartography there are many different models, which have their own advantages and disadvantages. This paper will focus on the categorization of these different software mapping techniques. It also introduces views from TOGAF. The main goal is to match software maps to TOGAF views, such that these views can be illustrated. This shall be done in order to offer an overview for which use cases a map is patricularly efficient or if there is an other better fitting model available.

1. INTRODUCTION

Nowadays the number of companies are increasing really fast, so does their intern complexity. The complexity is reflected by the high number of information systems and different available technologies. Even in todays medium-sized companies there is much information to be administrated. Enterprise Architecture Management (EAM) is getting more important to handle all the business processes and company-intern structures. Especially from bigger companies it is demanded to provide parts of these information outwards, because investors or other stakeholders want transparency. Therefore all the information has not only to be stored and administrated, but it also has to be visualized. The reason for that is that they, as onlookers can only understand the system from a clear overview. For the visualization of application layers usually software maps are used. There already exist many types of these software maps and each of them has its own features. It depends on the use case and the actual structure of the company, to say which software map is useful. In this paper the different types of software maps will be presented and their advantages as well as disadvanages will be discussed. As an example for an EAM framework the popular TOGAF will be presented. There will also be a categorization for these maps in order to give an overview of which software maps can be matched to which view of TOGAF.

The paper has the following structure: The next section will deal with the related work. The third section is devoted to the software cartography. After the software cartography section, software map types will be discussed. Then the TOGAF and the terms views, viewpoints and stakeholders will be introduced. In the sixth section core views of TOGAF will be presented and they will be matched with specific software map types. At the end, results and possible future work will be broached to round up this paper.

2. RELATED WORK

There are already many papers and theses which thematize software cartography, but they all have a different view on this topic. The ones I used to get information from, are more about the software cartography itself and focused on the aspect of how to represent information on a software map.

Buckl writes in her paper about how to generate visualizations of Enterprise Architectures with the help of model transformations. Thereby she mentions software cartography and techniques to visualize enterprise information content with it.[1] Lankes wrote two papers about software cartography, the first one adressing the visualization of application envirohnments, and the other one with the title "architecture description of application environments". [5] [4] Matthes has a short paper about software cartography in general, whereas Wittenburg has a much more in depth going PhD thesis about this topic. [6] In his PhD thesis Wittenburg goes more into detail by explaining in extensive chapters the theoretical background information and management of application environments.[12]

Therefore I picked the TOGAF as one popular example to examine particular steps of visualizing parts of the EAM. The TOGAF has an excellent documentation, where some key parts of the framework are explained. So, I picked most of the information about TOGAF from the documentation website of The Open Group. [10]

3. SOFTWARE CARTOGRAPHY

Many aspects of the software cartography are adapted from the original principles of cartography. According to the Wikipedia definition cartography is ”Combining science, aesthetics, and technique, cartography builds on the premise that reality can be modeled in ways that communicate spatial information effectively.” By applying the software cartography in conjunction with EAMs, it deals as a bridge to represent the abstract information about structure and processes of a company into a clear graphical view.

The software cartography is not only associated to cartography, but it has its roots also in the computer science and the economic science. Computer Science and economic science have in common, that both heavily work on project management, which is a central part of software cartography. In computer science, especially software systems engineering with its models for respresenting structures and information in form of diagrams (like UML), are important. Also the process management from economic science is relevant for software cartography. Finally cartography completes the

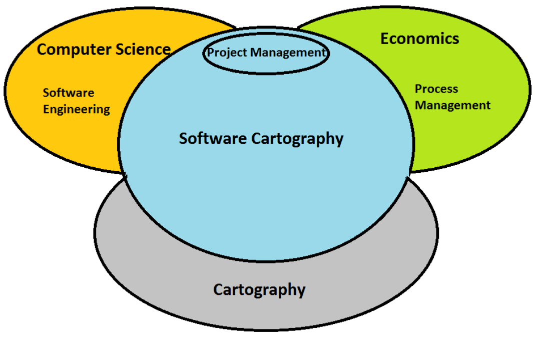


Figure 1: Software Cartography as an Intersection of three Sciences [6]

triple with its methods to create structured maps by utilizing colours, forms etc. Figure 1 shows that the software cartography indeed fits in as an intersection of all of these three different sciences. [6]

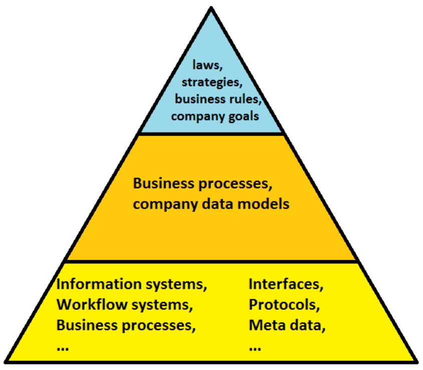

laws, strategies, business rules, company goals
Business processes, company data models
Information systems, Workflow systems, Business processes, ...
Interfaces, Protocols, Meta data, ...

Figure 2: Pyramid of Software Cartography [5]

Each software map has the purpose to give an insight into one part of business processes or parts of the application environment. This shall be achieved by connecting the three important viewing planes of the application environment in order to provide a good representation. These three viewing planes are shown in the pyramid of figure 2. First there is the ground layer, with information systems, protocols and interfaces, which all contain important information about how things work in a company. According to this, the ground layer is built all around the question ”How things are done in the company?”. The layer in the center deals with the business processes and company data models. From an abstract point of view these represent the core actions of a company. Therefore the fitting question to this layer is ”What does the company do?” Last but not least the spire of the pyramid features company goals, strategies, laws as well as business rules. Basically this contains the goals and also reasons for the actions of a company. The content of the spire can be summarized under the question ”Why does the company do something?”. A good designed software map has to have an intuitive representation and it should also connect all of the three shown aspects with each other. [5] [6] [4]

4. SOFTWARE MAP TYPES

In computer science there already exist different models to represent data structures and information systems. These are amoung other: the UML diagrams and the Entity Relationship model. It is possible to use these types of software maps also to model parts of an EAM. Nevertheless most of the time the power of these visualization models is not enough to represent the whole EAM or bigger parts of it. But as it will be discussed later, these known models can be used as an addition to the other existing software map types. In the rest of this section, some of the relevant UML diagrams and the four basic software map types will be introduced. For some of these map types there will also be an example shown.

In general it is hard to fit a whole application environment onto a map. Software cartography makes use of techniques from the cartography itself to visualize information. This is not always as easy as in the cartography due to the fact that methods of cartography are originally designed for geographical maps. Especially the limitation of space is one of the biggest challenges for a software map. In comparison to georgrapahical maps, software maps are not built on a topographical basis, which makes it much more difficult to visualize information. Each software map type has its own way to handle this challenge. One important characteristic to distinguish software map types is, if they have a base map for positioning or not. Having a base base map for positioning means, that the position of certain objects on the map has a semantical meaning, e.g. if two objects nearby belong together or are associated in a way. [12] In figure 3 on the next page the classification is shown. With the exception of Graph Layout maps, all other maps are based on the principle of base map positioning.

4.1 UML Diagrams

The Unified Modeling Language (UML) is collection of diagram types, which allow to visualize software archictectures or software structures. Each of these diagram types visualizes a different aspect of the software. There are three

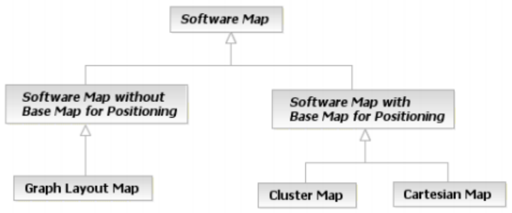

Software Map
Software Map without Base Map for Positioning
Software Map with Base Map for Positioning
Graph Layout Map
Cluster Map
Cartesian Map

Figure 3: Software Map Types [12]

UML diagram types, I will shortly introduce in this section, since the main focus is on the other software map types.[2]

4.1.1 Activity Diagrams

An activity diagram graphically describes a set of activities for a software. They are considered as state diagrams, therefore they begin with an initial state and end on a final state. There are also states in between. Every state has a name that describes the state in which the software can be. The states are connected together by arrows, which are the transitions. These transitions represent the sequential order of performed activities. Within an activity diagram there can also be decisions, where it has to be decided in which direction the control flow shall continue. In short, the activity diagrams represent for a set of actions, every possible state that the software can be in. They also show if a sequence of activities leads to the intended result. This makes them suitable in visualizing the data flow in a system. [11]

4.1.2 Sequence Diagrams

Sequence diagrams are usually created for one chosen scenario. In sequence diagrams each involved object has a class, to which it is assigned to. Every object has its own lifeline belonging to it. These objects perform actions, which can involve one or more objects. An action can be communicating with one or more other objects. But it can also be the case that an object does something on its own. The action in a sequence diagram follows sequentially one control flow. Therefore there are no parallel actions allowed in Sequence diagrams. All in all a sequence diagram has the task to represent in a graphical way, how the objects communicate with each other, such that the data flow between software components can be comprehended. [3]

4.1.3 Use Case Diagrams

This UML diagram type focuses more on the user interaction with the system. It shows what possibilities a user has, to interact with the system. Herefore the users are categorized, e.g. administrators, customers, etc. All of them can perform different actions. Every action the user can perform, is called a use case. Hereby the technical implementation of the actions does not play a role, because it is not relevant for the user interaction. Figure 4 shows an exemplary use case diagram. [8]

4.2 Cluster map

Cluster maps have the property that objects can be grouped together. On a Cluster map, a number of objects of the organization can be placed nearby, such that together they form a logical unit. Most of the time these logical units are also in a frame. Cluster maps make use of it to represent functional units, organization units or even geographical units like e.g. different locations, regions and cities. To hightlight the unity, most of the time the frames are coloured and logical units also have a capture. If one object has to occur more than once within a software map, belonging to two different logical units, then it appears two times on the map, once in each logical unit. An example for a Cluster map can be seen below in figure 5. [1] [6] [12]

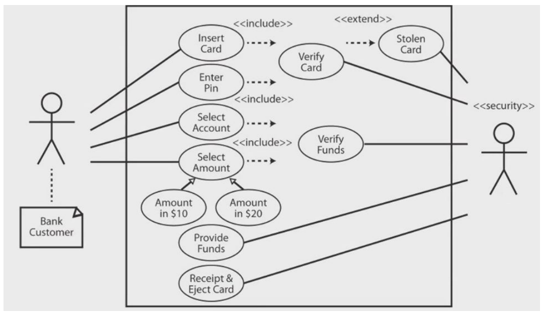


Figure 4: Use Case Diagram [12]

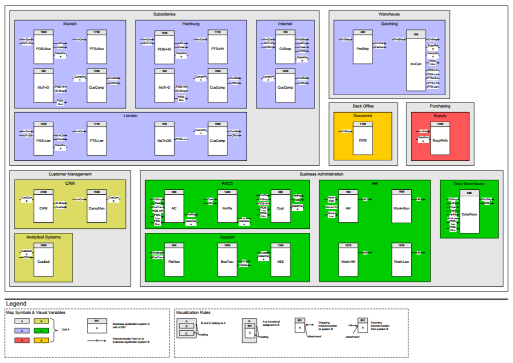


Figure 5: Example of a Cluster map [12]

4.3 Cartesian map

A Cartesian map is simply defined by a sofware map that has a base map with an x- and y-axis. The Cartesian map is abstract and only descirbes the layout principle of a map. How the actual map looks like and further attributes are defined by the one of the two following map types. The most popular representatives of Cartesian maps are the Process Supporting map and the Time Interval map. Both are described in the coming two sections. [1] [6] [12]

4.3.1 Process Supporting map

A Process Supporting map focuses on visualizing linear business processes. It is particularly effective to use a Process Supporting map, when the workload of one business process is distributed onto several organization units. Process Supporting maps have like every Cartesian map an x- and an y-axis. On the x-axis there are different processes stringed together, whereas on the y-axis the organization or executive units are. Because there could be many different business processes in a company, the primary process, respectively the one requiring the most organization units decides about the postion of the units. The y-axis can also just contain locations of the company or even different products the company has to produce. This makes the Process Supporting map very versatile. [1] [4] [6] [12]

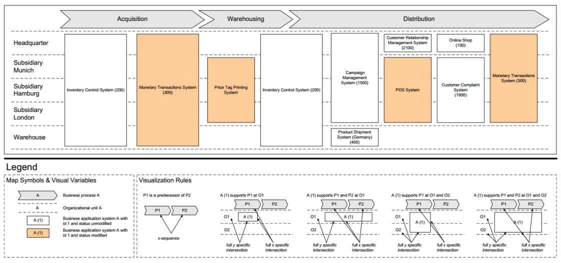


Figure 6: Example of a Process Supporting map [12]

4.3.2 *Time Interval map*

Time Interval maps are very similar to the Process Supporting maps. Like in Process Supporting maps the y-axis serves for displaying organizations or executive units of the company. But while on the x-axis of Process Supporting maps the business processes or particular steps of processes are depicted, a Time Interval map uses time for scaling. [6] [12]

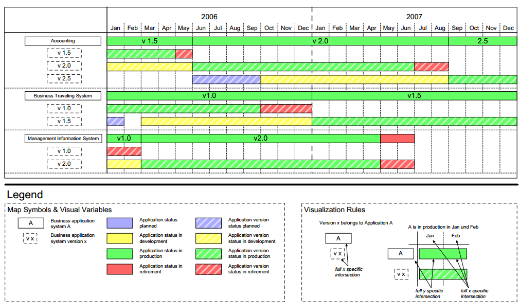

2006
2007
Accounting
Business Traveling System
Management Information System
Legend
Map Symbols & Visual Variables
Visualization Rules

Figure 7: Example of a Time Interval map [12]

4.4 Graph Layout map

In each of the previous presented map types the position of an object on the software map had a semantical meaning. Cluster maps use the space as a resource to express that certain organizations or parts of the company belong together. Cartesian maps, like the Process Supporting map and Time Interval maps are also based on positioning, since they operate on a coordinate system. In Graph Layout maps this is not the case. The position of an object does not have a semantical meaning. This is a great freedom for the architects who are designing a Graph Layout map. With the freedom of positioning they can create a much clearer map, which is more understandable for the viewers. The only disadvantage of this is, that all the freedom is bought by giving up expressing power for having free positioning on the map. Generally a Graph Layout map utilizes nodes and edges to convey information. So, togehter belonging objects have to be connected via association lines. Appropriate examples for a Graph Layout map are e.g. UML diagrams or Entity-Relationship models. [12] [6]

5. VIEWS, VIEWPOINTS AND STAKEHOLDERS IN TOGAF

The The Open Group Architecture Framework (TOGAF) is a free popular architecture framework. With the help of this framework it is possible to efficiently build an IT enterprise architecture for an organization. The architecture is developed by the core of the framework - The Architecture Development Method (ADM). The ADM itself is an iterative method to create the whole architecture. Step by step the architecture can be defined, whereby for each step the area which should be covered has to be chosen. Also the breadth of coverage as well as the level of detail have to be chosen. Below, you can find an image which is showing the inherent Architecture Development cycle.

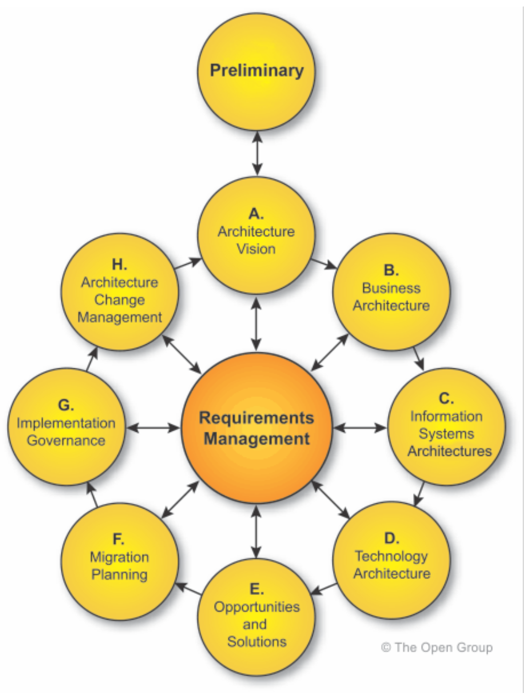

Preliminary
A. Architecture Vision
B. Business Architecture
C. Information Systems Architectures
D. Technology Architecture
E. Opportunities and Solutions
F. Migration Planning
G. Implementation Governance
H. Architecture Change Management
Requirements Management
© The Open Group

Figure 8: Architecture Development Method (ADM)[7]

Since todays companies have a complex structure, where many people and organizations are involved in one company, such frameworks are definetly a must. Within a company there are different business processes, associations and organizations. Accordingly the whole architecture of a company cannot be represented in just one software map. The entire architecture of a company is not interesting for every person which is involved, because different persons or groups within the company have different working environments and so each of them has also a different interest. The different parties in the company are named *stakeholders*.

STAKEHOLDER 1. *Simply expressed, a stakeholder is a person or an organization that is interested in the things another person or organization does.*

Most of the time a stakeholder is affected by the direction a company goes or even able to direct parts of the company himself. This can be people working in the company from "simple workers" up to the managers of a company. But stakeholders can also be extern people, who have interest in success of the company, like e.g. investors on the capital market.

Views 1. *A view is the representation of a context, e.g. a part of the organizations architecture, which can be in a form of a software map.*

Viewpoints 1. *Viewpoints are always refering to a stakeholder. Viewpoints are the point of view of a stakeholder who sees the company from his perspective, including only the aspects of the company which are relevant for his active working environment.*

Stakeholders can also be grouped together if they have similar or same interests, so that they all have a common viewpoint. To clarify these three important terms *views*, *viewpoints* and *stakeholders*, the following example should help: An air traffic controller and a pilot work at the same airport. These two are the stakeholders in our example. Both of them have a view of the system, but neither of them has a view of the whole system, since not every part of the system is relevant for both. In this example the viewpoints are the persepective of which the pilot sees the system and the perspective of the air controller on the system. All in all it can be seen that both stakeholders have a subset view of the whole system and that both views differ from each other. The pilot has a air flight view of the system, whereas the air controller has a air space view of the system. The context between those three definitions and the software maps is the following: Stakeholders have a viewpoint from which they see the part of the system which is relevant for them - the view. And these views have to be depicted visually, so software maps are used to translate the relevant information for the stakeholder into a clear graphic. The next section is about examples for different stakeholders acccording to the TOGAF in a company and which of the software maps can be used to visualize them. [7]

6. SOFTWARE MAPS FOR CORE TOGAF VIEWS

In the TOGAF core there are already standard views available. This section will deal with those views and it will provide suggestions on which kind of software maps could be useful to visualize these views.

6.1 Business Architecture View

At first, there is the Business Architecture View, which focuses at most on the user experience. Also part of the Business Architecture View is the production planning. The biggest issue is a change in the production process, so there have to be different scenarios for the prodution planning. To sum up, the Business Architecture View should be able to represent the production planning and the functionality respectively the usability of the product. In order to archieve this, at least three types of software maps will be needed: First a Cartesian map (Process Supporting or Time Interval) to model the production process scenarios. The remaining software map types have the task to demonstrate features of the product. To show the coarse functional features the Use Case diagram is a good UML diagram type for it, since it is easy to understand for the user. In addition, Activity Diagrams are a clear way to display all possible outcomes by using or operating the product.

6.2 Enterprise Security View

The Enterprise Security View is developed for security engineers, who have to ensure that valuable information cannot get in the hands of unauthorized persons or organizations. So they have to be aware of the data exchange and the connected system units within the product or company. It is very common that distributed systems are used within companies or are even part of a product. Since these systems have different locations Cluster maps are very good to depict them. The monitoring of data exchange and the data flow within the application are ideally modeled by UML diagrams. Activity Diagrams are able to show all possible states in a outcomes, which cases can occur. This is important to know for the security engineers, since they have to be informed about all possible system states. The data exchange between several system units can be modeled by sequence diagrams.

6.3 Software Engineering View

For software engineers it is important to have an overview about the data flow and structure. Therefore this information has to be visualized in a way for them. Software engineers can make use of almost any presented UML type, since UML diagrams were originally designed for software engineers. It often depends on the task of the software engineer, but in general every presented UML diagram type is useful for software engineering. Since most of the time software functionality is complex to represent and the positioning is not that important, Graph Layout maps are fitting for this view.

6.4 System Engineering and Communications Engineering View

System engineers and communication engineers do not have exact same tasks, but if we want to assign maps to these views, it turns out that we can assign almost the same software maps. System engineers are busy with optimizing software and hardware interaction as well as designing computing models for a distributed computation environment. From the communication engineer's point of view, it is important to understand how the communication within the application is handled and how the system communicates with foreign systems. Therefore an overview of the distributed systems in the network and the communication models itself (e.g. OSI Reference Model or similar) is mandatatory for both views. An exemplary communication model can be seen below in figure 9 on the next page. For these models, Cluster maps are a good choice, since they are strong at representing content where the positioning and linking between units are relevant. The communication between system entities can be modeled by the UML sequence diagrams.

6.5 Data Flow View

Controlling and monitoring the whole lifecycle of data witihin the system is the task of a database engineer. This includes data storage, data retrieval, data processing, data archiving and also data security. So for this view it is needed to have an overview about the data structures as well as how the data is managed in the system. The ER model provides an optimal model to depict the database structure. The security and communication of data can be monitored from sequence diagrams like mentioned in previous views. As far

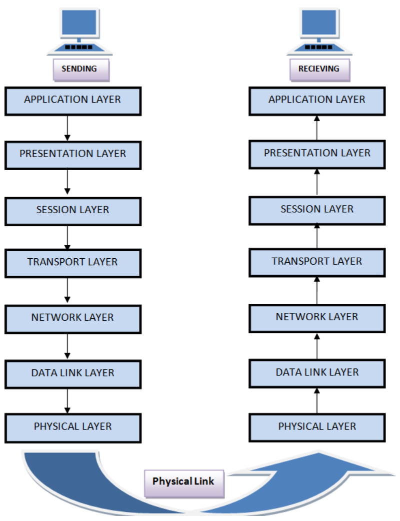


Figure 9: Exemplary Communication Model [9]

as the data processing within the system units goes, Activity diagrams are good to see how actions influence the system state and how the data is affected by this.

6.6 Enterprise Manageability View

Unlike most of the previous views, the Enterprise Manageability View is not related to engineers. Stakeholders of this view are the operations, administration and management personnel of the system. These personnel has to have the ability to oversee the structure the management within the company, as well as planning on future investments in projects by regarding the available budget. Basically most of the high level decisions are made by these personnel. The general structure of the management, executive organizations and also the different locations of the company are important for this personnel. For modeling the connection between the different organizations of a company, one should make use of Cluster maps. It makes sense to use them here, because the positioning and grouping of certain elements is really important. Business processes can brilliantly be depicted in a Cartesian map for the same reasons as mentioned in the Business Architecture View above.

To sum up, we can say that in general the management based views utilize more the Cartesian maps and the technical views which deal with the implementation of the product or system, more rely on UML and Graph Layout maps. What both of these view categories have often in common is, that both make use of the Cluster map. [10]

7. CONCLUSION

The paper began with the goal to first present general information about software cartography and the TOGAF. This has been done by first explaining the software cartography and introducing the different software map types. After that the TOGAF has been suggested as one popular representative of the EAM frameworks. With the framework also viewpoints, views and stakeholders have been described. Finally some of the core views of TOGAF and the most important software map types were matched together. This matching showed for which views, which software map type can help to constitute the needed information efficiently. As for the future work this procedure can be done in a similar way for more views or for other frameworks.

8. REFERENCES

[1] S. Buckl, Alexander, and M. Ernst. Generating visualizations of enterprise architectures using model transformations. 2007.

[2] T. Horn. Uml unified modeling language. `http://www.torsten-horn.de/techdocs/uml.htm`.

[3] IBM. Uml basics: The sequence diagram. `http://www.ibm.com/developerworks/rational/library/3101.html`.

[4] J. Lankes. Architekturbeschreibung von anwendungslandschaften: Softwarekartographie und ieee std 1471-2000. 2005.

[5] J. Lankes. Softwarekartographie: Systematische darstellung von anwendungslandschaften. 2005.

[6] F. Matthes. Softwarekarten zur visualisie rung von anwendungslandschafte n und ihr en asp ekten eine bestandsaufnahme. 2004.

[7] M. Schaefer. Enterprise architecture bebauungsplanung fÃ̃ijr informationssysteme. `http://st.inf.tu-dresden.de/files/teaching/ws11/ring/20111123_Capgemini_Vorlesung_TUDresden.pdf`, 2011.

[8] B. Schaling. Das use-case-diagramm. `http://www.highscore.de/uml/usecasediagramm.htmll`.

[9] Studytonight. Communication model figure. `http://www.studytonight.com/computer-networks/images/Figure25.png`.

[10] TOGAF. Developing architecture views. `http://pubs.opengroup.org/architecture/togaf8-doc/arch/chap31.html`, 2006.

[11] uml diagrams.org. Activity diagrams. `http://www.uml-diagrams.org/activity-diagrams.html`.

[12] A. Wittenburg. *Softwarekartographie Modelle und Methoden zur systematischen Visualisierung von Anwendungslandschaften.* PhD thesis, July 2007.

Technical Debt Calculation And Its Uncertainties

Waqas Ahmed
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
waqas.ahmed@rwth-aachen.de

Firdaus Harun
RWTH Aachen University
Software Construction
Ahornstr. 55
52074 Aachen, Germany
firdaus.harun@swc.rwth-aachen.de

ABSTRACT

We don't know how much we owe (i.e., Technical Debt) unless a calculation is made. A lot of TD measurements have been proposed and they seem promising. There are mainly two different major approaches to calculate TD, one is estimation of TD with interest and the other is estimation without interest. However both of these approaches are not useful unless we understand and consider different uncertainties during calculation of these approaches. In this study, we will investigate (1) what kind of uncertainties are reported in literature related to Technical Debt and (2)why they are important to consider.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.9 [**Software Engineering**]: Management—*productivity, programming teams, software configuration management*

Keywords

Technical Debt, Principle, Interest, Interest Probability, Uncertainty, TD measurements

1. INTRODUCTION

The term Technical Debt (TD) was used for the first time by Ward Cunningham in 1992. It is a metaphor indicating technical compromises that produces short-term benefit but it hurts the long-term health of a software system [10]. Talking in the context of source code technical debt is a poorly written code, requiring extra effort to correct and modify it in future. There are three terms related to technical debt including principal, interest amount and probability.

Principal: Principal of technical debt is defined as the cost of re-factoring to clean code.

Interest: Whereas, interest is defined as the extra costs and effort required by developers to work with messy code / functionality during maintenance or new features addition.

Interest Probability: Interest Probability is defined as the probability of technical debt leading to future problems and costs, if not it is not paid on time.

Based on different issues and causes of technical debt Flower in [6] classified technical debt into different groups as presented in figure 1 above.

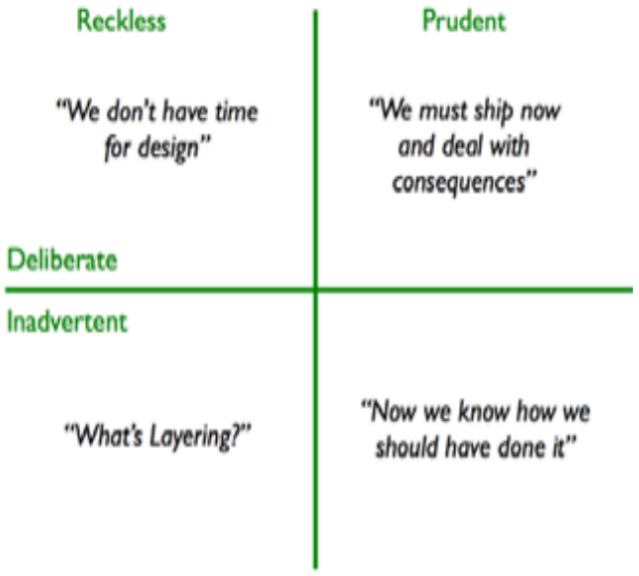

Reckless
Prudent
"We don't have time for design"
"We must ship now and deal with consequences"
Deliberate
Inadvertent
"What's Layering?"
"Now we know how we should have done it"

Figure 1: Types of Technical Debt, [6]

In literature and software industry different terms and properties are attributed to technical debt, Brown et al. discussed some of the properties and major issues of technical debt [2] in detail to provide a vision to better understand technical debt which are are as follows:

- **Visibility:** Due to lack of technical debt visibility, serious management and maintenance issues are caused. Sometimes maintenance task is handled by a third party organization, in this situation maintenance vendors are unaware of any unforeseen technical debt which was introduced by the development team. Moreover maintenance vendor do not get the idea how badly the code is written until maintenance is started. In case of identifying technical debt, it is hard for these maintenance vendors to analyze,that what were the causes of technical debt.
- **Value:** Technical debt is not a negative thing, if it is managed wisely it can add value to software system. Just like, in real life mortgages besides its downsides help people to buy homes.
- **Present Value:** Present value, cost and impact of the technical debt for the same project vary from time

to time. Therefore it is necessary to analyze its impact probability and uncertainties during cost benefit analysis.

- **Debt Accretion:** Managing technical debt is critical as too much debt leads to bad impact on the system. If technical debt is not paid on time, it become difficult to maintain and modify the system according to new requirement .
- **Environment:** Technical debt is highly dependent to its developing environment and context. For example sometimes an organization deliberately introduces some technical debt to meet a short deadline or to overcome lack of cost and resources. Similarly, sometimes technical debt is incurred unintentionally due to lack of expertise of development team.
- **Origin of Debt:** Technical debt can be classified into two broad categories based on its origin including strategic and unintentional debt. The former increases system value if it is managed wisely while the latter is highly discouraged as it badly impacts project value and management.
- **Impact of Debt:** Both in case of strategic or accidental debt, the impact of debt is different varying in scope from local to global to overall software project.
- **Uncertainty:** Estimating technical debt precisely is difficult, as it is dependent on different number of factors, some of them has been discussed above. These factors vary from project to project. Additionally these factors also vary within a project over different time frames.

If TD is incurred wisely it helps to bring business value. However,this TD must be managed otherwise it causes serious issues in system maintenance and evolution [10]. In order to effectively manage TD, identification of TD items and TD measurement is necessary before prioritizing them in order to select which TD item should be paid first. For this purpose different TD estimation models have been proposed. However precision and accuracy of these measurement models varies and is effected by underlying uncertainties. These uncertainties are characterized due to lack of sufficient knowledge of different factors as discussed above, consequences and level of impact of these factors on system health in future.

As uncertainty in measurements is inevitable in software engineering processes [1]. Its critical to consider different uncertainties and errors related to technical debt. The focus of this research work is to provide an overview of these uncertainties. Paper organization is as follows: in section 2, two of the known models for technical debt are discussed followed by different uncertainty factors in detail and errors in section 3 and finally concluding the paper in section 4.

2. TECHNICAL DEBT CALCULATION MODELS

In literature different methods have been proposed to calculate this technical debt. Following is the overview of some of these methods.

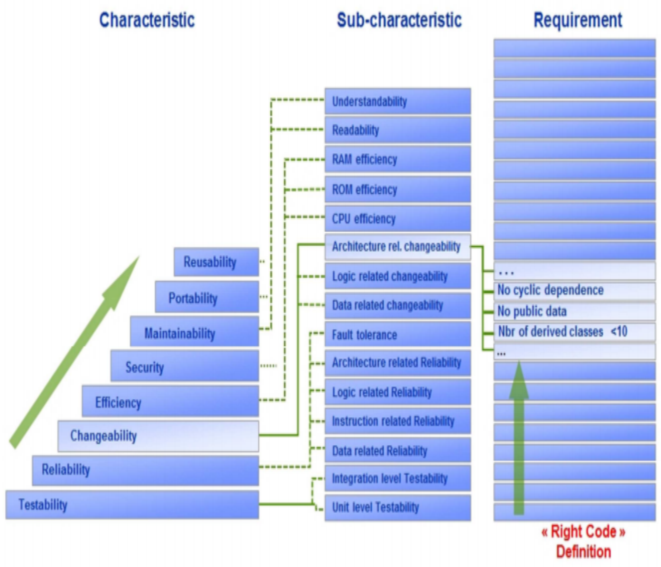


Figure 2: Chronological Order in SQALE, [9]

2.1 SQALE

Software Quality Assessment Based on Life cycle Expectations is a technique applied on source code for its quality evaluations. Version 1.0 of this method provide support for calculating technical debt in an organization. First of all, organization identifies the requirements related to projects, serving as the basis for right code. In SQALE it is called 'Quality model'. Any violation / non-compliance of this quality model creates the code debt. These requirements vary in nature from non-functional requirements to code presentation, naming or design and architectural once. Against each requirement its remediation functions is also listed. These remediation functions provide the basis for calculating remediation cost caused by non-compliance/violation of each requirement. Then, analysis tool is run to calculate the technical debt which is actually sum of all remediation cost incurred by the violation of above set rules.

The SQALE method provides the requirement to be grouped under eight quality characteristics including testability, reliability, changeability, efficiency, security, maintainability, portability and usability in a chronological order. This chronically is important and should not be changed as it badly effects and increase the technical debt if it is changed. Figure 2 provide overview chronologically ordered eight characteristics using by SQALE. If a requirement is related to multiple characteristics it must be associated with the lowest one in chronological order.

Against each of these eight characteristic SQALE provides an indicator index to built a pyramid view and corresponding debt related to each characteristic debt. If the chronology order in the pyramid is not followed it leads to loss of resource and time, thus introducing overhead costs. For example testability should be done before targeting reliability. This pyramid provide the technical perspective of the debt.

As paying whole technical debt is not feasible most of the times. In this case, one need to prioritize within technical debt available time. For this solution SQALE provides another method and different indicators and indexes. Just like remediation function non remediation function is also associated with each requirement. This the non remediation function actualy estimates the penalty that the Product Owner (or someone who represents the Business) may claim as compensation for accepting violations [9] . For ease of use each requirement is classified into different classes of consequence such as "blocking", "critical", "major" "high" "low" and a symbolic cost is assigned to each category. As it is difficult to find out exact consequence of each violation so we broadly classify and associate a symbolic cost with it.

SQALE then defines index for summing up all of the non remediation costs which is called SBII (SQALE Business Impact Index). This SBII provides the business perspective related to technical debt. So instead of just relying on the technical aspect SQALE takes into account business perspective to better utilize and payoff it within limited time and cost constraint. Remediation priority (Non-remediation cost / remediation cost) is displayed in the form of Debt Map Graph which priorities giving with high return can be selected.

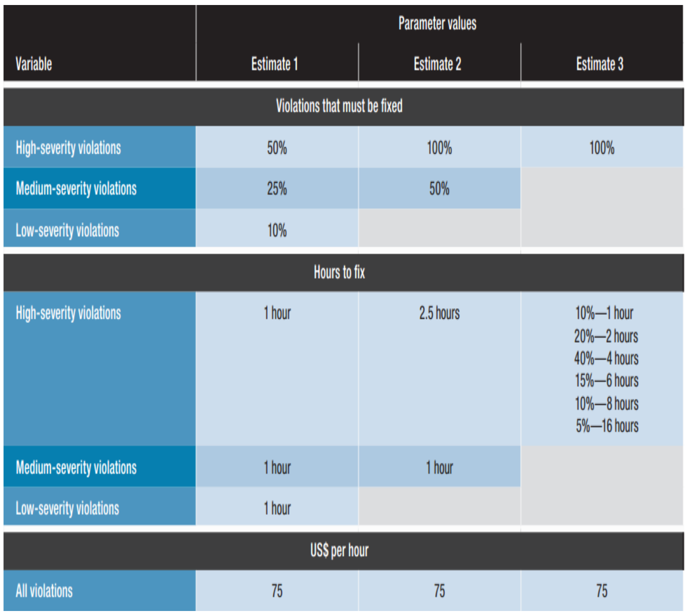
2.2 CAST: Curtis Estimate Models

Curtis et al. [3] presented three different estimate models for calculating technical principle debt. This article was focused on calculating the technical debt as principal only. With the availability of limited resource and time, it is not possible to reduce all of the technical debt. Therefore companies need to prioritize to reduce technical debt based on their available budget. Authors provided the overview for calculating principle technical debt based on following three parameters:

1. No. of should fix violations
2. Hours required to fix all violations
3. Cost of labor

Where hours required to fix all violations can be obtained from historical data of similar projects while cost of labor can be set as the average labor cost in an organization. These parameters can be used under different situations varying from organization to organization such in accordance with the severity of violation such as high, medium and low severity. Based on these three parameters authors defined three estimate methods to calculate the technical debt as shown in Figure 3.

For estimate method 1, constant hours were set to fix different percentages of violations, authors marked it as too conservative approach. For estimate model 2, variant percentage of violations were set to be completed in different time-span. While estimate model 3, considered different hours distribution to fix high level violations. To find the effectiveness of each estimated model, authors tested them using data from Appmarq benchmark repository maintained by CAST Software for 700 different applications containing at least 10 KLOC per application. These applications were



Variable	Parameter values		
	Estimate 1	Estimate 2	Estimate 3
Violations that must be fixed			
High-severity violations	50%	100%	100%
Medium-severity violations	25%	50%	
Low-severity violations	10%		
Hours to fix			
High-severity violations	1 hour	2.5 hours	10%—1 hour 20%—2 hours 40%—4 hours 15%—6 hours 10%—8 hours 5%—16 hours
Medium-severity violations	1 hour	1 hour	
Low-severity violations	1 hour		
US$ per hour			
All violations	75	75	75

Figure 3: Parameter Values For Three Estimates, [9]

analyzed using CAST's Application Intelligence Platform (AIP). AIP is supported with databases of 1200+ violation rules for 28 different languages belonging to architectural and coding.

AIP analyzes and parses source code based on meta-data parsing. Violation score is defined by the probability for a rule being triggered and number of times it is violated based on severity. Then different reports are generated to guide the developers to location where each volitional is done. AIP combines violation scores in under following five categories, robustness, performance efficiency, security, transferability and changeability. Authors presented data for above discussed model by diving and categorizing on the basis of language and then in the perspective of quality measure. Based on the analysis authors presented benchmark stats based on the historical information of 700 applications to define different violation rules specific to each category.

3. UNCERTAINTY IN TECHNICAL DEBT MODELS

There are a large number of TD measurement approaches, models and tools available. Li et al grouped and classified 49 different such studies on TD measurement along with enlisting 8 different tools from research literature [10]. Some of these tools are mentioned as follows: CLIO tool [14] is used for for detecting modular violations, RBML checker is used [12] for calculating deviation between design pattern and actual implementation of that pattern. For detecting code smells, as defined by famous author Fowler[5] there is a tool called CodeVizard [15].

If we calculate technical debt multiple times with different tools, it is obvious to get different results. Because, each tools have different parameters, dependent / independent variables and algorithms to calculate technical debt. As each tool operates on different TD calculation model and there must be some uncertainties associated to each underlying model. Curtis, Sappidi and Szynkarski [3] stated that "there is no exact measure of Technical Debt, since its calculation must be based only on the structural flaws that the organization intends to fix," as different organizations have different goals so they measure and allocate resources to find and fix technical debt differently to with scopes.

With the availability of such large number of TD measurement approaches and tools, an organization often need to make decision for choosing an effective approach / tool. Thus, selecting an appropriate approach / tool without knowing its precision and related uncertainty is a daunting choice.

Uncertainties are inevitable in SE processes and measurement models [1]. Abran et al classified uncertainties of measurement into Experimental standard deviation, Error (of measurement) Deviation, Relative error, Random error, Systematic error, Correction, and Correction factor [1]. While reporting measurement method / models, sufficient information must also be provided regarding its uncertainties.

Reports on technical debt mention such uncertainties for example , Letouzey and Ilkiewicz [9] in the SQALE methodology mentioned it. The SIG (Software Improvement Group) software quality assessment method [7] based on ISO/IEC 9126, also tells that technical debt measurements are not free of uncertainties. So in technical debt measurements it is very important to take into account of these uncertainties and accommodate them in expressing technical debt.

In order to accommodate these uncertainties in technical debt calculation its vital to understand their causes and origins. However, in literature related to TD not much information while discussing different TD measurements approaches and models. Izurieta et al adapted different general uncertainties principles from physics and mapped them to TD models [8]. We will make an attempt to discuss these provided models and elaborate them in more details.

$$measuredvalueofTD_{principal} = (TD_{principal})_{best} \pm \partial_{TD} \tag{1}$$

equation (1) defines the general matrix for TD measurement. This matrix also in addition to considering technical debt principal also take into account margin (denoted by ∂_{TD}) of error or uncertainties of technical debt calculations. In above equation $TD_{principal})_{best}$ indicates the best results reported by subjected tool / model. The uncertainty term ∂_{TD} in above example catering both random and systematic errors.

3.1 Comparing Measures

Results containing single measure are not that useful, scientists usually compare two or more measurements to check relationships between values. Technical debt literature is fairly new and the disadvantage is there is no standard accepted values for technical debt calculation like other scientific fields. Consider a scenario where an organization C purchases software from organization A and assign organization B for its maintenance immediately. Organization C has requested to both the organization to report TD estimates when the purchase and maintenance made. Now both organizations have calculated and reported TD. Lets say two companies A and B have reported their Technical debt principal as follows

$$(A\_TD_{principal})best \pm \partial_{A_TD} \tag{2}$$

$$(B\_TD_{principal})best \pm \partial_{B_TD} \tag{3}$$

then for computing highest probable value for the estimate we can use this equation.

$$(A\_TD_{principal})best - (B\_TD_{principal})best + (\partial_{A\_TD} + \partial_{B\_TD}) \tag{4}$$

and for computing lowest probable value for the estimates equation is

$$(A\_TD_{principal})best - (B\_TD_{principal})best - (\partial_{A\_TD} + \partial_{B\_TD}) \tag{5}$$

If we say both companies have described their uncertainty with measurements using equal number of figures then inconsistency in uncertainty should also be enlisted using same number of figures. If one company is reporting uncertainty with different granularity than the other then the final reports must take that into account and use common measurements in results.

3.2 Propagation of Errors

When calculating complete technical debt calculations, we must take care of how value of uncertainties of technical debt interest and probability propagate. If we calculate the value of technical debt principal, we still need to take care of uncertainties present in average labor hours to fix low quality code which have architecture violations and other issues, the cost of per hour, and average cost in time to fix one violation. Again by using Taylor's [13] rules to estimate the propagation of technical debt uncertainty [8] discussed following equations proposed by Nugroho et al. [11].

3.2.1 Sum And Differences

If several quantities x1 .. xn with their uncertainties are measured with uncertainty then the overall uncertainty of their additions, differences or combinations of both operations are:

$$uncertainity = \partial x_1 + .. + \partial x_n \tag{6}$$

3.2.2 Product And Quotients

The propagation of uncertainty in measured quantities in context of products or quotients can be calculated by using fractional uncertainty notation. Calculation of technical debt as defined in terms of equation 1 then we can define fractional uncertainty in technical debt principal as follows:

$$fractionalUncertainityTD_{principal} = \frac{\partial_{TD}}{|TD_{principal}best|} \tag{7}$$

Nugroho et al. [11] calculated technical debt principal by also considering the Repair Effort (RE) which is required to achieve ideal level software quality. RE can be calculated as

$$RE = RFXRV$$

here RF is Rework Fraction and RV is Rebuild Value. The RV can be calculated for a system by multiplying System Size (SS) again Technology Factor (TF). Number of man-months per statement is TF and System Size (SS) can be calculated either by using lines of code (LOC) or function points. Also uncertainty will be there in above calculations too like function point calculation will have higher degree of uncertainty than using LOC. So Rework Fraction can be calculated like this

$$measuredvalueRF = RF_{best} \pm \partial_{RF} \tag{8}$$

$$measuredvalueRV = RV_{best} \pm \partial_{RV} \tag{9}$$

then uncertainty for the measure value of Repair Effort (RE) is given by

$$\frac{\partial_{RE}}{|RE_{best}|} = \frac{\partial_{RF}}{|RF_{best}|} + \frac{\partial_{RV}}{|RV_{best}|} \tag{10}$$

if several quantities x1 âĂę xn with their corresponding uncertainties then total uncertainty of their products, quotients or both can be calculated like following

$$\frac{Uncertainity}{|measure_{best}|} = \frac{\partial x_1}{|x_{1best}|} + \frac{\partial x_n}{|x_{nbest}|} \tag{11}$$

3.2.3 Power Uncertainty

Nugroho et al. [11] presented technical debt interest amount calculation as the difference between ideal level of Maintenance Effort (ME) needed for a software module and current level of maintenance effort. For calculating Maintenance Effort formula is

$$ME = \frac{MFXRV}{QF} \tag{12}$$

Here $QF = 2^{((qualityLevel-3)/2)}$ where quality level can have values from 1 to 5, so QF values can 0.5, 0.7, 1.0, 1.4 and 2.0. Maintenance Fraction (MF) is the number of lines that will be subjected to change in a year and Rebuild Value (RV) can be calculated as

$$RV = SSx(1+r)^t XTF \tag{13}$$

so in above equation for time t and growth rate r, the RV of a system will increase over time itâĂŹs not taken care of systematically. If the rate r changes as time increases then this equation should take that into account too or uncertainty of the measurement. How it can account for uncertainty we can only focus on $(1+r)^t$ factor of the Rebuild Value (RV) calculation. The uncertainty in multiplication of System Size (SS) and Technology Factor (TF) is carried out using propagation techniques for production and quotients as described before. If r is measured considering uncertainty then the overall uncertainty of Rebuild Value (RV) can be calculated as follows

$$\frac{\partial_{RV}}{|RV_{best}|} = \frac{\partial_{SS}}{|RV_{best}|} + tX\frac{\partial_r}{r}\frac{\partial_{TF}}{|TF_{best}|} \tag{14}$$

3.2.4 Technical Debt Interest Probability

Because probability of interest will vary with respect to time so a time element is necessary to consider in calculations moreover values assigned to interest probability usually classified in ordinal scale based historical values. There is no systematic formula to calculate interest probability calculations, if probabilities are table based and ordinal then further exploration is needed to come up with a formula for calculations.

3.3 Multivariate Uncertainty

Formulas that use quadrature where appropriate are described by Taylor [13] but these need validation in technical debt domain. For ignoring negligible effect of some unlikely error propagation possibilities we can use quadrature, which will help us in having realistic error range when calculation error in multivariate expression. When measurements come from Normal or Gaussian distributions we can use quadrature equations nicely but those distributions also should be independent.

3.4 Technical Debt Interest Uncertainty

Carlous et al. in [4] presented a cost analysis model for estimating technical debt using binary trees. Carlous et al. considered two important parameters which are interest uncertainty and time frames. Interest uncertainty is defined as the probability that no extra cost is derived from technical debt [4]. Technical debt vary from time to time for the same project under maintenance depending on different number of factors. These factors differ in nature from internal such as low code complexity to external such as difference in use of software for varying time length depending upon business activities.

Using maintenance cost as the measuring unit for technical debt during system maintenance and evolution, if a software is not modified over a time period than no interest amount should be paid for this period. Taking in account interest uncertainty factor helps to better estimate cost and benefits of technical debts.

4. CONCLUSION

Technical debt is a popular term used to define technical compromises undertaken during the software development. If managed properly technical debt minimization can bring great value to the product. In order to manage and organize technical debt different calculation models and approaches are used. However precision of these approaches vary greatly due to large number of uncertainties. In order to get the most precise calculations these uncertainties must be taken into account during TD calculations. In this paper we have highlighted the importance and different classes of uncertainties that being used in the literature during TD calculations.

5. REFERENCES

[1] A. Abran, A. Sellami, and W. Suryn. Metrology, measurement and metrics in software engineering. In *Software Metrics Symposium, 2003. Proceedings. Ninth International*, pages 2–11. IEEE, 2003.

[2] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, et al. Managing technical debt in software-reliant systems. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 47–52. ACM, 2010.

[3] B. Curtis, J. Sappidi, and A. Szynkarski. Estimating the principal of an application's technical debt. *IEEE software*, (6):34–42, 2012.

[4] C. Fernández Sánchez, J. Díaz Fernández, J. Garbajosa Sopeña, and J. Pérez Benedí. A cost-benefit analysis model for technical debt management considering uncertainty and time. 2013.

[5] M. Fowler. *Refactoring: improving the design of existing code*. Pearson Education India, 1999.

[6] M. Fowler. Technical debt quadrant. *Bliki [Blog]. Available from: http://www. martinfowler. com/bliki/TechnicalDebtQuadrant. html*, 2009.

[7] I. Heitlager, T. Kuipers, and J. Visser. A practical model for measuring maintainability. In *Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the*, pages 30–39. IEEE, 2007.

[8] C. Izurieta, I. Griffith, D. Reimanis, and R. Luhr. On the uncertainty of technical debt measurements. In *Information Science and Applications (ICISA), 2013 International Conference on*, pages 1–4. IEEE, 2013.

[9] J.-L. Letouzey and M. Ilkiewicz. Managing technical debt with the sqale method. *IEEE software*, (6):44–51, 2012.

[10] Z. Li, P. Avgeriou, and P. Liang. A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, 101:193–220, 2015.

[11] A. Nugroho, J. Visser, and T. Kuipers. An empirical model of technical debt and interest. In *Proceedings of the 2nd Workshop on Managing Technical Debt*, pages 1–8. ACM, 2011.

[12] S. Strasser, C. Frederickson, K. Fenger, and C. Izurieta. An automated software tool for validating design patterns. In *ISCA 24th International Conference on Computer Applications in Industry and Engineering. CAINE*, volume 11, 2011.

[13] J. R. Taylor. An introduction to error analysis: The study of uncertainties in physical measurements, 327 pp. *Univ. Sci. Books, Mill Valley, Calif*, 1982.

[14] S. Wong, Y. Cai, M. Kim, and M. Dalton. Detecting software modularity violations. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 411–420. ACM, 2011.

[15] N. Zazworka and C. Ackermann. Codevizard: a tool to aid the analysis of software evolution. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, page 63. ACM, 2010.