On Technical Debt in Software Testing - Observations from Industry

Sigrid Eldh[1,2,3(✉)]

[1] Ericsson AB, Networks Standards and Technology Software Computer Science, Stockholm, Sweden
sigrid.eldh@ericsson.com
[2] Department of Innovation, Design and Evolution, Mälardalen University, Västerås, Sweden
[3] Carleton University, Ottawa, Canada
https://www.es.mdh.se/staff/67-Sigrid_Eldh,
https://carleton.ca/sce/people/eldh

Abstract. Testing large complex systems in an agile way of working was a tough transition for systems having large active legacy and honouring backward compatibility. Transition from manual test to full test execution automation resulted in increased speed and manifested technical debt. The agile way of working in continuous build and test, creates a lot of repetition by execution of the same tests. Overlap between agile teams producing similar test cases, causes a constant growth of the test suites. Despite the obvious improvement of automating millions of test cases, the numbers provide a false sense of security for management on how well the system is tested. The causes of technical debt should be addressed, instead of managing the symptoms. Technical debt in software testing could be addressed by refactoring, supported by known techniques like cloning, similarity analysis, test suite reduction, optimization and reducing known test smells. Increasing the system quality can also be improved by utilizing metrics, e.g. code coverage and mutation score or use one of the many automated test design technologies. Why this is not addressed in the industry has many causes. In this paper we describe observations from several industries, with the focus on large complex systems. The contribution lies in reflecting on observations made in the last decade, and providing a vision which identifies improvements in the area of test automation and technical debt in software test, i.e. test code, test suites, test organisation, strategy and execution. Our conclusion is that many test technologies are now mature enough to be brought into regular use. The main hindrance is skills and incentive to do so for the developer, as well as a lack of well educated testers.

Keywords: Test automation · Test maintenance · Agile development · Technical debt · Industry testing · Test strategies · Quality assurance

Supported by Ericsson AB and Mälardalen University.

T. Margaria and B. Steffen (Eds.): ISoLA 2022, LNCS 13702, pp. 301–323, 2022.
https://doi.org/10.1007/978-3-031-19756-7_17

1 Introduction

Agile development brought many benefits [22,44,60] for most industries, mainly improving speed of delivering new functionality, and a strong uptake of more new and modern tools. Automating the build and test loop in a continuous integration has solved integration testing issues seamlessly. In large complex systems, there are many challenges remaining [24,44]. In an earlier study [71] we could conclude that *"a comparatively high technical debt is accepted in test systems, and that software design organizations in general forget to apply software design principles such as systematization, documentation, and testing, when producing or deploying test automation systems"*. If we revisit the same organisation, change did happen. The major change is that the (test) systems are now looked upon as a production system in itself, and treated in the same way, and some attempt to unify the test tool situation has happened. The four observations claimed in this former paper, are unfortunately still true (for full explanation, please revisit the original paper): Reuse and sharing of test tools brings issues that need to be considered i.e.; Test facility infrastructure is not transparent and may alter the test results if not accounted for; Generalist engineers expect that their tools are easy to use; Accepted development practices for test code are potentially less rigorous than for production code. In addition can add new observations as a result of testing in the agile way of working.

Specifically, we can observe test suites are growing in size to the point that prioritization, selection and reduction approaches are in strong need, instead of the initial agile aim to "run all tests" at every change which is a consequence of larger organisations. Focus then quickly becomes on maintaining test suites to reduce and minimize suites without loss of executions. The overlap of test case creation is an unfortunate consequence of many concurrent working agile teams creating overlapping test cases. The concurrency of teams also cause delays the CI-flow showing up as merge conflicts. Though, a rich overlap in the test suites can be helpful when using technologies like Automatic Program Repair [73]. This is not on people mind, as it is general a reluctance to remove test cases (due to difficulties to identify actual execution patterns) as well as the testers do not want to see their hard work being undone.

The overall problem remains, the abundance of test cases - will not only have a high cost attached to it, but it will create a false sense of security of how well the testing is made. As the wide adoption of poor metrics like test coverage are used, this also reinforces the core issue of poor test design, often stemming from the lack of rigorous specifications, as agile advocates learning by doing and "fail-fast" schemes. In this paper we will discuss technical debt that still remains in the software testing context with a focus on test automation, and provide a vision in the area. This paper is written in the context of a large world-wide organisation, with many research and development organizations. The observations made, are also compared with other large world-wide organisation, where the author have had insight in. Therefore, the observations are made as generic observations, and no liability to or claim of, that the observation stems from Ericsson should be made.

The paper is structured in the following manner. The next chapter is about the background and context in the telecommunication domain, and the third chapter is about test automation in industry. Then Sects. 4–12 are structured lists of observations grouped under common themes, in the attempt to make the readable. This is then followed by Sect. 13 that aim to describe a vision on testing in the future followed by a description of Methods used, Threats of Validity and disclaimers. We then conclude our paper.

2 Background - Telecommunication Domain

The telecommunication domain faces many challenges affecting the software. The area is strongly regulated with standards, and the ability to communicate everywhere - from condensed areas in cities, to remote locations are now a necessity for the society build on mobility. Telecommunication systems have moved from the fixed line 2G to become wireless connections in any device. Most countries are now transforming to 5G with high-speed secure communication enabling the internet of things (IoT) as well as specialized networks, e.g. in mines - to allow for remote automatic and safe mining [2]. As a telecommunication providers, Ericsson is creating end to end network solutions for its customer, the operators. The new incentives to move telecommunication to the cloud, and into the Fog [46] brings not only new competitors, but also new challenges. Issues like the ever increasing need for speed is resolved by redundancy in hardware. Code size and variety of offered solutions creates a massive need for concurrency, leading to timing issues and increased complexity.

Legacy systems partly suffer from monolithic architectures that are costly to refactor. Attempts to transform, slice, and simplify the software through e.g. utilizing micro-services is an ongoing transformation. Variations of product offerings in e.g. radio is in itself an issue resulting in costly tests. Software needs to be secure and perform in high-speed environments. Many historically well made decisions, are now suffering from lacking advanced tools for different analysis and modern testing features. As Agile assumed self-organisation, the variety of solutions, tools and programming languages are strong. Testing these systems is a constant challenge, but gives a researcher ample sources to juxtapose technologies and approaches.

The Continuous Delivery and DevOps approach is a bit more tricky [10] also for the telecommunication sector, as it assumes the operator (the customer) needs to be prepared for automated delivery. Automatically acceptance test and automatically provide feedback and data, the latter which is a legal area, as the delivered solution should be integrated and work in a new environment alongside competitors solutions. No wonder this is an area of much regulations and standards to define operations.

The general management view is that testing is already too costly. Much of the cost comes from simulators and test lab equipment as well as the cost of the entire development infrastructure with tools.

3 Test Automation in Industry

We can conclude that the number of testers has strongly declined in the agile context as quality is the task of everyone, the tester role disappeared within the agile teams.

Automation of test execution in industry has come a long way, thanks to the agile transformation. Industries developing safety-critical or heavy regulated software, are more focused on test automation by modelling the requirements and generating the test than more traditional commercial software. The trend is clear. As modelling tools constantly improve, and more developers are educated in formal approaches reaches industry, the likelihood of using modelling as a natural tool in the requirements phase is obvious. At least for reference models in systematization. Modelling (through tools) as development still has not really grabbed the general industrial developer community. But in one sense, all is a model, and simple state-transition tools have been around for more than three decades (at least within Ericsson). As we state below, there is still a strong development by manual coding and "hacking" culture alive in most industries, as experience of the difficulties with early modelling tools still lingers. Utilizing modelling for test automation has its clear champions in industry. Unfortunately, if you do not have a champion in your agile team, it is very unlikely more formal model approaches spread until tools makes it easy and safe.

Test automation is very much a developers affair in the agile team, where the tendency is to automate test execution for unit tests or low level functionality. As systems are large and complex, the knowledge to test of all aspects of e.g. a telecommunication node and system functionality for a small team update is much to ask. Most developers lack enough insight in detailed execution patterns on system level for their part of functionality, e.g. concerning timing and other system parts executing concurrently. This lack of test environment understanding is a significant source of flakiness in the test suites [55].

When it comes to content and creation of the test cases, little attention is paid to utilizing test design techniques. We can observe that today the test automation approach used upon the first automation of test execution has indeed established itself as a practice: Test cases were automated based on manual use cases tests, where there is a lack of separation of execution and data, with many input-output relations hard-coded in the test case. The lack of common directives of utilizing any specific test technologies to create test cases is unfortunately a fact. Quality Assurance is in principle to enforcing statement code coverage in the gating for CI/CD loops, using secure coding principle, embedded in static analysis tools. The use of static analysis tools like CodeCompass [59] and CodeChecker [1] are now becoming mandatory, but there is still lot of work remaining to bring 20+ programming languages used in a product portfolio into a common quality approach. The quality is (still) much dependent on manual practices like code reviews and even exploratory tests [50], as some people still advocates manual practices despite goals aiming for "complete" automation. The digitalization drives automation, and it is common sense to aim to automate as many aspects

as possible, to gain constant improvements and repeatability. Unfortunately, it is still hard to transfer the knowledge behind solutions made.

4 Test Design Technologies

The largest gap between academic research and industrial practice in testing is that many of the test design techniques described by academics are rarely used, not used systematically or as a regular practice [25,33,36]. This differs between industries, as more regulated industries, e.g. in safety critical systems, pay more attention to some of these. Below we list a more detailed discussion on why this is the case, what techniques we are talking about and an attempt to describe some context. This lists have arbitrary order and weight, depending on what software and system in what industry we are talking about.

1. Test Strategies do not contain any information of **how** things should be tested. This results in a lack of proper systematic approaches of what test design technologies used. Instead test strategies, and even e.g. agile testing strategies, advocates what and when to test as well as e.g. manual testing (exploratory test) [20].
2. The normal case is well tested at all levels of test, demonstrating the main functionality [25].
3. Negative test (testing fault handling and anything outside the normal case, including unspecified behavior), is randomly performed, except for robustness testing at system level. Hardware redundancy acts much as "information/bug hiding" of issues in this area [25].
4. The dominating functional test approach is requirement testing. This is often expressed as use cases. As requirement specifications in agile are "user stories", it allows for not being fully specified up front. A consequence is that the requirements can lack sufficient detail to be used to for sufficient verification. This is especially clear when changes occur that details are lost updating specifications [25].
5. As systems today contain functionality that has been transformed and updated for decades, the historic documentation is not always kept accurate, especially not in regards to tools, versions and patching. This lack of documented detail, prevents modelling methods and more formal methods to be used for testing on a regular basis.
6. There is no lack of attempts to model both source and test code, and success varies. There are many reasons. Champions are often stuck in a team, and modelling needs to be used and taught more widely. Modelling must be a system decision and there is a lack of management back up. The model must meet the code at some point, and here is the difficulty. Maintaining models are costly, and it is harder to optimize auto-generated code for performance. Also, it is also harder to check security and static analysis on generated code - and corrections would be meaningless, hence to fulfill the harsh requirements, many (not all) modelling tools and languages must

have experts in-house that can change and update the modelling environment concurrently. Also intermediate languages or the auto-generated code must be adapted to compilers that sometimes are proprietary, even if most today aim for commonplace front and back-end approaches like e.g. LLVM [47]. Therefore currently, for most industries *code is king* meaning, what is written in code is what is used and triumphs both documentation, and tools. In safety-critical systems *model is king*. Ground truths differs for different industries. Historically, e.g. UML [30] was successful, but is now partly abandoned. ETSI TTCN-3 [3] is also made into an ITU-T standard [6] but does only cover partial test suites within at least the telecommunication industry. It has not kept up in competition with other (newer) languages, and developers are not so keen on it, despite the potential.

7. Requirement test means that test cases are aimed to "cover" the requirement - and manual analysis is performed to check that it is done. A lot relies on this non-scientific measurement called "test coverage" of the requirement. A requirement can need one test case or 30 000 test cases. Hence, talking about 100% test coverage is nonsense. Taking an extreme example, is that if you only had one test case for a system that is passed, you would have 100% test coverage. Therefore, test coverage should always mean code coverage. And as there are so many - you need to be specific of which code coverage.
8. Requirements management quality varies due to the agile transformation. It is more often better described if the specification describes phenomena closer to the user (or something used by the developer). Some requirements in the machine-to-machine communication, layer 1 and layer 2 code in the OSI model [62]), platforms, embedded code, or code closer to hardware, are examples that take much longer to teach developers all about the domain specific aspects. Consequently, there are often a lot to improve, when it comes to supporting verification and validation [16].
9. As requirements seldom contain "what should not work" i.e. fault handling, there are few test cases developed or prioritized in this area.
10. For specific protocol's standards and telecommunication standard aspects, careful modelling through e.g. Mathlab/Simulink [28,43] is regularly used. In Hardware (FPGA's) stricter modelling tools are used, as this can then be transformed to a layout in hardware. When it comes to hardware/software co-design, modelling tools are regularly used to asses what is most efficient. Modelling is often used as reference (reference modelling) for prototypes, but not in the context of fully generated code and/or test cases except samples. This could for example be a set of input defined, often referred to as test vectors, e.g. specific VHDL (Verilog High Definition Language) [52] or ATPG (Automated Test Pattern Generation) both in common use for hardware tests.
11. As data is often in abundance for large complex systems, another road taken is to use constraint technologies, e.g. MiniZinc [53] that can aid in a lot of human complex tasks - keeping track of endless rules of what data can be combined with what. This area shows a lot of promise for the future. It is

easy to hope that one define, they work, but also constraint solvers also have to be tested - making sure rules are created and used correctly.

12. Many test cases became "frozen" in its development, during the agile transformation when automating these (functional) tests, much due to lack of time to refactor test cases but also due to poor test automation and test design know-how. The effort to automate them was challenging (costly) enough. The result is that a lot of manual test cases were not transformed to good automation, utilizing libraries, avoiding repetition etc., and did hard code input and output/verdicts. They were translates as is. This makes many test cases very inflexible. Very few test design techniques were used then, at its best, you find three clones with hard coded boundary values, as an input. As a result, test selection will remove two of these for being too similar, and the boundary check is lost. This is just one simple example. Documentation of why a test case exists are totally lost. At best the test case can be linked to a requirement specification, or deduced in time to "what project" that created it. This is not very helpful. Therefore good and instant search and test case tagging must exists in the tools to identify similarities and understand context of test better. It would also be helpful to have trace information of what the test case actually address. We are far from these features in test tools today. No-one wants to update the information in the tools to make it work either.
13. This lack of systematic testing and use of test design still remains, as each agile team mainly "copy-paste" earlier test patterns. Exceptions do exists. This results in a lot of clones and duplications of test code [39].
14. The more recent prioritization of security has made many testers transfer to security. Checking for vulnerabilities implicitly improves testing, as some limited fuzzing through the available fuzzing tools is now becoming standard practice. Most fuzzing tools does basically perform input variation testing. Books like Zeller et al. [75] also make standard testing practices and design techniques useful by exploiting the security context.
15. Testing for code coverage is often set to 80% statement coverage as a gating for legacy code, if used at all in industry. There is no control mechanism that checks that this coverage is done properly, whereas "creative" avoidance code to pass the gating is occasionally used. Some claim that code coverage is a poor metric to drive quality. The reason is that e.g. statement code coverage can be easily misused (see Marick's comment on this [49]). You can also do poor testing with good coverage (i.e. selecting input that does not invoke faults). This historic view have made many developers reluctant to use it. Some agile teams also test statement code coverage on integrated code modules instead of the unit, which makes it very difficult to reach higher coverage values as paths then exponentially grows. Enforcing strict 100% statement code coverage could be a waste too, as some code are not cost efficient to cover. The simple approach to use coverage as a self-check if the tests created was good is often lost in this debate of code coverage, test coverage or any other creative invention of coverage.

16. There are many agile teams that have abandoned proper unit test, and mainly work to test functional tests, as the test harness and unit test suites are costly to keep up to date with many code changes. It is true that for some functionality it is difficult to test in isolation at lower levels and is easier to test through its context. This leads of course to more difficulties to achieve code coverage targets, but also brings a hidden bug count, as new bugs come into play at code changes. In addition, it is easier to test an item in its real context, instead of creating (and maintaining) stubs, which at first glance seems costly, can be difficult to keep up and time consuming. Using controlled stubs alleviates a lot of unit test issues, makes units easier to understand, test and control. There are tools that allows real code context to "act" as stubs which aid better control. This area should be better explored both by unit test tool vendors and academic research.

5 CI/CD - Build, Test and Regression Testing

The CI pipeline, where the true sense of the agile approach to submit frequent small changes can easily in a large organization become a big-bang integration. Examples range from between 100 to 2000 commits in one hour, making it clear that several pipelines was a solution. As the CI pipelines were conducted as a part of the agile self-organisation, there are a variety of solutions and also on what tools were used [65].

1. The CI framework is used to check that new code integrates, which is the most common fault issue with committing a change.
2. Fixing build faults are currently a manual process, from debugging (fault localization) to patching. The new patch is code reviewed by others in the team and submitted. No extensive test framework exists for most pipelines doing build changes, which can cause delays in the framework.
3. Current attention in one of the organizations tracking and remedy top 10 most unstable (nondeterministic) test cases (also known as flaky tests) are often root caused to poor test case writing, lack of insight in test environment and simulators that also execute with timing discrepancies. Identifying and remedy issues that prevent flow (i.e. flaky tests) is a key industrial practice following agile test automation in the CI flow [11,14,23].
4. Test suites have too long turn around time for some test tools, the test cases should be better sliced and grouped and intermediate feedback should be allowed, to save overall time.
5. The CI pipeline can have too long turn around time, often depending on the vast number of test cases (and not sufficiently equipped test environment, as these are costly)
6. Regression Test Selection (RTS) could be improved in some of the pipelines to speed up order of execution.
7. Test suite can easily be in a wait state - (appearing to be hanging) as it is waiting for input (from e.g. test environment) - as the test resources takes time to set up and use, etc. This causes unnecessary flakiness issues [48].

8. When testing against real hardware - there are many issues of test equipment failures and timing - causing the pipeline to be hand, stop, or be in wait state [48].
9. If testing is using simulators - the simulators are often "too good" in the sense that they do not simulate hardware issues or bugs, but definitely not concurrency and timing sufficiently. A good simulator should be possible to put in "faulty" mode. And controlled faulty is much easier to debug than when it happens in the field.
10. The cost to update simulators to more accurately test fault hardware behavior and timing issues (concurrency) is steep, and difficult to make happen if commercial simulators are used.
11. Therefore software solutions, e.g. mutation testing tools, are so successful to induce bugs [45,56]. The issue is that mutation test generate "too many" faults at "too low level" - and many claim that many of these generated faults are "too simple" and therefore not representative. It is definitely worth investigating what typical faults are, for a specific aspects of the system. Especially to aid fault localization and support fail handling. To make it worth embarking on mutation testing, tools must improve speed, accuracy, and usability as well as users should start with sufficiently high code coverage.

6 System and Non-functional Testing

Most advanced and intense testing happens in non-functional testing (or at system test level), such as performance test and load test, stability test, availability test, robustness and resilience test, to mention a few. There exists dedicated specialized testers in this area.

1. KPI's (Key Process Indicators i.e. Key metrics) are essential for the products commercial value.
2. These areas of system test are instead of being a "one off" in the end, now with agile constantly measuring and performing automatic evaluation, running the latest versions of the software and deploying intelligent machine learning tools to support fault and issue localization [37].
3. The automation also contains automatic transitions between simulators and the real hardware, which enables applications to run seamlessly. A consequence can be that not all measures are on "real" hardware, and this can cause a false sense of quality, but has the advantage of removing specific test lab equipment issues.
4. As the system test is constantly measuring, it has strong similarities with monitoring - and making the DevOps model fulfillment in this regard, a much easier transition between internal testing and operations [57].
5. As non-functional level is fully automated relying on functional tests, there are still many possibilities to improve e.g. visualization.

6. Here, AI/ML supported test can aid in finding faults, producing more intelligent metrics trends, swapping between simulation tools and real networks and find strange anomalies among millions of metrics. Network resilience, robustness, stability and reliability is measured, as well as performance testing.
7. In general, telecommunication sector is well equipped to test at network level, with a combination of real utility equipment (UE's) and simulations tools, making systems well tested.
8. Performance, load, overload and similar aspects are key metrics, and is well evaluated on all levels of testing also regulated e.g. in the telecommunication standards through ITU-T.
9. Security testing (incl. Penetration testing, fuzzing etc.) is a non-functional test approach, but can be done at all levels of testing, and is not typical system test matter, rather associated with level code, static analysis tool and so on [12].
10. Not all non-functional tests described in e.g. ISO/IEC std. 25010 Series [5,29] are done at system level, and that is maintainability. This is not considered well a well measured or clear metric as is. as some aspects are disregarded - and that is maintainability of the test suites. It is not particularly measured, or any specific attempts made to improve this aspect of neither source code nor test code.
11. The testers at this level are well acquainted with the system, and have expertise in both measuring and troubleshooting the system. As most KPI's have high commercial value, this area has management attention, and the existing expertise make sure that the technical debt in the area remains relative low.
12. The area suffers from normal test design issues and a lot of of faults show up that should have been take care of in earlier stages.
13. Concurrency and redundancy at system level can hide bugs at this level, making fault-finding difficult and time-consuming.

7 Test Maintenance

1. Test suites are seldom refactored and improved in general. For maintained code, fault test cases are sometimes made passive (silent) instead of root caused and fixed. This means over time test suites are becoming less powerful.
2. As a result of not keeping test suites improved and updated, test cases grow old [32]. Some test cases that do not find any faults are down-prioritized to execute seldom or simply removed. As there is a lack of tool support to safely remove test cases - as sometimes there are hidden dependencies in the order of execution, few testers do anything about this problem - as time is too scarce anyway to just write sufficiently new amount of test cases. Agile team's that do most of the functionality test automation, are not measured at all on any test quality, but on functionality delivered.

3. If any refactoring is done on test code, it is basically done during the construction of the test case e.g. look at how test lags the source code writing [74]. Many (non software based) industries do not do obvious code fixes, like refactoring of e.g. test smells [68]. These smells should be easy fixes. Instead time is spent on bug fixing (after the fact) or if a test fails. It would be so easy to focus on the top worst code pieces and make them fault free. See e.g. Software Quality Rank Model [25] that uses simple targeting of worst culprits, and then uses basic quality approaches to refactor to at least have these *fault free*.
4. Test design is mainly created through copying earlier test cases and changing them. As a result we have concluded that 15–20% identical duplication (measured by SonarCube [9] (i.e. Type I cloning) exists in the test suites. On average 30–50% code cloning overlap (measured by NiCad [21]) (Type I–Type IV) exist in the test suites. These duplication's and different types of overlap exists at all levels - even if it is easiest to identify them at unit level, as the tool has language restrictions. Note that some test suites had as much as 80% code cloning duplication when Type III clones were used.
5. Hasanain et al. [39] did a thorough clone analysis with NiCad on large part of Ericsson's Test code, from specific sub-systems functional test code, to unit test code packages in a large variety of software. Despite the high cloning results, that could, with some effort be removed, no investment or action was taken. Even if the results of such investment would not only reduce test code footprint by large, and also speed up the test suites in the pipelines which implicitly would lower energy costs, this was met with cold shoulders, as the understanding of testing and test suite impact and gain is low. Another factor that makes this hinders to perform this refactoring is the lack of skill to write "good" test cases among test managers, test guardians and test architects, and developers per se. This is because most testers were good because of their domain skills and manual testing skills, follow suit of test managers, test guardians and test architects, with a few exceptions. Developers who implicitly would have sufficient coding and refactoring skills, are simply bad in test design techniques.
6. Even if clones exists (Type I–Type IV) [61] only some of Type I code is removed. This is rather daunting as e.g. Van Bladel et al. [17] showed that there is much value pursuing also Clones of type IV. In general test cases are made of copy-paste, therefore there is no incentive to remove or refactor test code [39].
7. Too simplistic tools are used like SonarCube, [9], and at best [21], NiCad has been used, even if Van Bladel et al. insist on combining several clone coding tools and shows that they can target different types of clones [18]. The cost to integrate a set of tools in production is still to steep and the incentive to improve on this typical test smell is surprisingly low.

8 Fault-Fixing Loops

1. The majority of bugs are found in the first early commit code in the CI flow. This indicates that testing at the lowest unit level is not very robust (as unit tests should be performed before the check-in, if specifications were correct or existed), as many bugs are found after commit. An observation is that many developers trust that the integration and functional tests will find all important bugs instead of the more time-consuming unit tests. Meaning how a small changed piece of code works in its real context will always reveal bugs. Instead the contrary is true - many code changes passes through the test flow completely untested (other than at the unit level), as no particular test targeting the change as been made.
2. External faults (Anomaly Reports, AR) found are given a high priority, as well as faults from system test are since the 2012 in automatic routing (e.g. triaging) to the right organisation/team, and through various machine learning and Bayesian approaches, now more perfected [40–42]. Unfortunately, the fault localization has not reached exactly the lines of code but is at best remaining on file level - but work to improve this algorithm is constant.

9 Test Tools and Test Environments

Historically, most test tools were proprietary, as there was a need to work with specific protocols, but TTCN-3 [35] and efforts in ETSI [3] has brought through tools like TITAN [66] and testing approaches like UML modelling [30] very much in use. As development of software has progressed (with Agile methods) the change to other types of software development methods, embracing open source software, new tools like GitHub [4], Jenkins [7] and Maven [8] have been examples of concepts that is embraced and sometimes used alongside proprietary adaptations of legacy test tools.

1. As many unit tests are dependent on adequate test tools for the language at questions it is often the case that unit test code is written in something completely different than the source language. The debate what is best here is still not straight forward.
2. There is a lack of good tool support, e.g. test tools, static analyzers and fuzzer's for new languages. Such a simple thing as creating CFG's (Control Flow Graphs) does not exists out of the box for all procedural languages.
3. Low-level code close to hardware, e.g. embedded and/or proprietary code, will be unique as it needs tools to be developed in-house. This is often a hidden cost and a hindrance for unified test approaches.
4. Test systems are generally not aligned and integrated in their way of working, and a large diversity of tools, approaches and concepts are still existing with a high technical debt to merge [58].
5. Tests written in old tools are very reluctantly changed to new tools. Therefore many old tools still linger in the system. In addition, as self-organization allows for some aspect of low-level decisions, the temptation to use new and

more modern tools is high, resulting in an abundance of tools from the last decades.
6. There is an interest to consolidate tools to lower licensing costs, and treating them as a production system in itself. Despite this, tool changes always come second, as the first goal is always to deliver new functionality, giving organisations reluctant to change, a way to avoid tool change.
7. With an abundance of tools, it is hard to please all, as functionality is different in different tools. Consolidating them is inherently hard.
8. As tools are now the new security threat in themselves, the future will probably look a lot different when it comes to using tools from outside the company firewall- this will also be a hindrance to explore new researcher tools.

10 Quality Assurance

There is a lack of control and checking mechanisms that testing is done properly, e.g. measurements for coverage or test criteria are seldom used. I.e. quality assurance control is diminished in the agile context to attempting a set of test cases being passed. As the software product is constantly in flux, being updated, corrected and changed, most (larger) industry software needs to have code freeze on a branch before a delivery - instead of being able to deliver the entire software at "any moment", as the original intent of agile was. Sometimes tolerance to submitting code with failed test cases are allowed to circumvent this issue. Hence, there is seldom a state where "all test cases are passed" in a large system. This obfuscation creates a blunted view on quality in the form of test cases as control, and a lot of management effort is put on justification of being "good enough" quality for submission. Sometimes even test cases are removed that are difficult to pass, but mostly - measurements that indicates these issues (of poor quality) have instead been removed.

1. Developers are only fulfilling mandatory gating requirements. If they are set low (e.g. 80% statement code coverage) this is where testing stops. Test adequacy criteria an coverage is important, see Zhu et al. [76].
2. As Agile team's goal is to deliver functionality, and no quality measurements or checking of the test quality is made (other than code review- that mainly focus on the source code), the quality of testing is in principle poor.
3. Mutation testing [54], could be a way to check the quality of the test. Unfortunately - mutation test operates mainly on the code level and in the context of unit tests, and as such - it misses all functional and non-functional (as well as integrated) executions of the software.
4. Testing is considered a cost to be reduced as its value is poorly understood, i.e.testing is "a necessary evil".
5. Most industry delivery plans do not take refactoring into account, especially not for testing.
6. The need for speed to deliver new functionality often compromises quality up front.

7. Test maintenance is considered a "non-issue" as maintenance is often exported to "low cost" sites. This does not teach developers to have a "quality mind" - and learn from their own mistakes, and is probably (with lack of incentive for developers) the main reason for poor quality.
8. Test Coverage is measured, but this is not a clearly defined metric that provides any scientifically or statistical value of how well a system is tested, hence progress and accomplishments sounds better than reality is (see earlier discussion on this).
9. Bug reports (anomaly reports) are one of the few tools to evaluate the final quality of the product, based on the customer. As it is costly for customers (time, effort) to also report bugs, only a few are reported. Another confounding factor hiding quality issues is as mention the redundancy in the product. This makes many software bugs go undetected. Concurrency (parallelism) in the product also make many bugs difficult to reproduce.

11 Knowledge and Skills in Testing

There is a lack of proper education on testing for developers and engineers (from university, and within the company)

1. Developers in agile teams are rarely trained in testing [34].
2. As 7 of 8 test managers cannot or do not read code, hence, checking how e.g. coverage is fulfilled, and test cases are constructed, is not managed and supervised sufficiently.
3. Some internal courses and leaders are also advocating manual test or "exploratory test", despite the company's goal to automate as much as possible and automate testing approaches. Manual testing is good for learning, but should not be advocated, as the regression test cost advantage is lost, as well as the possibility to improve the testing in both accuracy, speed and quality.
4. Most commercially available courses are too simple, and not geared to teach advanced automated testing that is needed (but a few exceptions).
5. There is no incentive of becoming a good tester, as they are rarely valued, do not have a clear carrier staging, and have lower status and salary compared to being a good developer that delivers functionality fast.
6. As a good tester finds more bugs, they are instead viewed as hindrance (to delivery) instead of helpers, which limits the incentive to learn new techniques, become better and add quality to the system [34].
7. Preventive techniques, such as modelling and constraints fall under "system work" and is therefore a better quality approach. These systems must also be tested.

12 Management

The main caveat with observations from management in industry, is that there is a lack of research in the area and from industry. Most information around it can only be found i grey literature.

1. Testing in general is seen as a cost that do not contribute to the product. Most of the cost is due to expensive test equipment and usage of commercial simulators.
2. As the organisation moves software to low-cost for maintenance, there is little incentive to improve your code from the start.
3. The reward system for developers is not geared towards rewarding good testing. It is geared towards a) saving the day (when difficult bugs exists) b) to deliver functionality and changes fast (and pass the CI loop). As fewer detected bugs let the code pass the CI loops faster, the current way of working actually rewards less testing.
4. Management prefer system approaches rather than proper test, which means that modelling could be a way forward. Against this, is our large experience of how modelling can be very costly, as at some point the model must meet reality, e.g. translate into hardware instructions. Many development managers still keeps the memory of this costly issue that historically has locked them into specific vendors and tools.
5. The abundance of test cases (in the millions) creates a false sense of security that the system is well tested. Especially when considering the high cloning percentages.
6. There is a belief that a tools can fix all problems, which implicitly makes it easier to invest in e.g. a tool than to actually test properly. The belief is that AI and Machine Learning will solve everything, which of course is not true (yet). The main caveat is that one must at some point - both for any learning system or modelling to generate code or whatever approach or technique you use - actually know what is correct behavior - or expected behavior. Until we can express this fine-grained enough we will not produce trustworthy solutions.
7. The promise of APR (Automatic Program Repair) has been strong with management since Ericsson's first trial 2011–2012 in cooperation with Weimer and DiLorenzo [63,70]. It became clear that this technique relies on a lot of unit tests but primarily identifies single fault issues. As half of the true software faults being of single fault nature, shown on typical telecommunications middleware code [27], singe fault failures can be prevented by mutation testing. There are still a lot to be exploited in this area.
8. As much work is a result of proprietary and self-organized way of working, the lack of unified set of tools, approaches, languages used, etc. causes a very diverse environment (due to historic decisions). This makes transformation difficult to newer methods, and changes in behavior - as tools and way of working, as well as attitudes remain the same. Testers and testing is considered "old-fashioned".

13 Visions on Test in the Future

Despite a somewhat gloomy set of observations above of test debt in industrial systems, the future is very promising on many accounts.

First and foremost, formal tools are improving on performance, usability and can better scale to larger applications. These factors together with more educated computer scientist coming to the job market, provides a base to use more formal, modelling and model-checking tools in industries that are not used to do so (i.e. industries that have not are not working with safety-critical aspects). Not being a safety-critical company, has also been used as an excuse for many commercial software companies to avoid investing in any of these "costly" technologies. Here is the incentive - improved requirement, need for automated support, and more easy to use and powerful tools, minimizing the transition to the code. This gap needs to be reduced to make "normal" commercial software use these technologies.

Secondly, AI (ML)- support will inevitably drive improved software, as code auto-completion is now supporting some aspects of source code creation (suggesting e.g. next character or line of code), there is no reason why we cannot train ML models to write better test code and test patterns, to guide developers in describing better testing. As with any technology - it will not be better than the people training the models. The key is, again, to know what is correct - the source code or the test, and know what is a "better" test approach. This also leads back to better tools and models used. Positive news also comes mainly from the webscalers and the occasional leap happening for the larger more mature industries, often with researchers driving it. Webscalers put serious money in their testing frameworks, as the business loss of poor quality poses a security risk. This creates more advancements for industry and new collaborative research on testing. A recent comprehensive literature study in the use of AI and ML in software testing can be found in Battina et al. [15].

Security concerns being strongly related to software testing are pushing quality higher on the agenda. Examples are e.g. Zeller et al. with the interactive Fuzzing book [75] and the survey by Felderer et al. [31]. Many technologies well known for academics, but new for industry use, are being matured and adapted for use. Examples are metamorphic testing [19,64], mutation testing [54,56], evolutionary approaches [13,38,51,69] and many new analysis tools combined with learning technologies. As Industry has the data, context and commercial incentive, collaboration with academics to perfect tools and technology will be the way forward. It is probable that the intelligence in the tools will be hidden completely from the user in tools, much as learning tools behave. Novel approaches even attempt to make testing into a game, where the user gets awarded to create tests, clear is that it should be pleasurable [67,72].

The issue is of course for many companies to tackle old fashioned organisational structures that do not promote software engineering. Most management are not educated in a software driven world nor understands where quality comes from in software. If companies cannot provide quality at a high level and low cost, they will have an issue to survive in the long run, as quality is expected. Every developer must know testing basics, and new ways of working, e.g. utilizing improved modelling and increasing the abstraction levels. Testing must be taken into account, as even a constraint model needs to be tested.

Automation of test will also become a strong drive, as still surprisingly many aspects of software development are still manual. The approach of automation everywhere in the entire R&D process is currently scrutinized, as is the aim to simplify. What can be done to create reliable, robust and well suited systems, that handle faults "automatically" so they magically disappear? The digitalization will also drive more automation, integration of tools, frameworks and systems will lead to changed focus. Monitoring and testing will be more aligned. We are already letting the customer test more, through targeted pilots, A/B test approaches and canary testing. Decision support will be increasingly deployed by different AI and machine learning bots as trustworthiness in the system increases.

The TAIM model - Test Automation Improvement Model [26], describes how automation growth, also implies that robustness, trustworthy data handling and fault localization leading to automatic program repair are issues to be solved, all suffering from the same issues, what is correct. In the future we can imagine that systems also take examples and probes to learn (with some guidance) what solutions are. The drivers are definitely based on how to monetize on quality, which is not currently clear. Maybe a future is very much assuming that you pay for quality solutions, and other "free" solutions are inherently relying on you as a user to debug them (implicitly) by claiming faults and sharing your preferences. Clearly the TAIM 2.0 model is a vision, where only some aspects of test management seems to have gained ground, where e.g. the need to progress report testing is very much historical, as a dashboard can keep continuous updates on the progress - as well as provide fault history and give quality assurance overviews with a simple drill down visualisation of issues. Most are working on avoiding to look at this data at all, and have machine learning only highlight what is important. Driving this machinery are assumed to be an abundance of "good" tests. This is the main caveat. Just because the number of tests are many they have high overlap (cloning) and are is not necessarily exercising other than variants of the normal case. Here new technologies like metamorphic test, great simulation tools for the test environment, that can also mimic faulty hardware, can make strong contributions. The security issues, which forms one sub-category of faults and issues will also drive much stronger technologies e.g. block-chains. Because of changing difficulties in these technologies, the aim to make sure they are solid in themselves also drives more formal modelling approaches. Unfortunately, the need for these technologies to be established, and the secrecy surrounding best practices, are not fruitful for exchanging best practices.

What drives automation is not only the goal to save cost, increase speed and reliability, but also to remove tedious repetitive ways of working where the interaction with the software is key. There is still much to be learned about bots, how they interact between them, and with humans, and what the best way to drive things. It is easy to hand over the responsibility of "what is right" to the computer (bot, AI decision system), and humans need to stay active and tell what is correct. So supporting in the right way might be the trick to success.

14 Method Used, Threats of Validity and Disclaimers

The method used to gather this data, is through observations, documentation existing, presentations and interviews in an action research setting. The sample of observations is a convenience sample. Therefore no statistics have been exposed or tables made. Despite being observations, there is undoubtedly a researcher bias in the selection of the observations and how the data is described. Effort has been made to make observations as factual and general as possible, seeking support in academic publications where available. Due to page limit, not an exhaustive reference list have been made for all items, as this paper aim to provide a research agenda and vision to inspire future systematic studies. As with all large world-wide development organisations, what is true for a majority (within one company), is not true for all, and there are exceptions to these observations. Examples comes from geographical diverse set of R&D centers in Asia, North America and Europe, and have been compared with industries in a large variety of domains, within mainly Europe, but also to some extent in Asia and North America. Disclaimer: As this paper is based on an invited talk, there should be put no blame or liability to Ericsson AB, as all observations and analysis expressed, are the made by the researcher. We also thank Ericsson AB and Mälardalen University for the support in writing this paper.

In addition, the lists presented have no weight in the order intended, and are to be viewed as an arbitrary enumeration, solely to create aid reading. It could be suggested as an easy exercise for any industry to walk them trough and create a priority order suitable for their specific environment to remedy, or for an academic to attempt to establish solutions, provide metrics and establish such an order.

15 Conclusions

In this paper we discussed a series of observations on technical debt of software testing and test automation in industry. As most software development has transformed to Agile and DevOps processes, the testing is still surprisingly manual even if automation is happening. Test automation means in industry automating the CI pipeline, and particularly test execution, not automatic test generation. The focus is often on the lower level tests, as this functionality is well understood by the agile teams (developers) as there is a loss of professional testers following the agile transformation. Non-functional tests require expertise, and here testers remain active, resulting in lower test debt and better acceptance for the cost of testing, as these testers produce metrics (KPI's) used in sales.

The focus should be on alleviating overlap of test clones and duplications through test code refactoring with the aim to reduce the cost of constant testing in the CI-flow. This is not happening, as test code refactoring is "too costly". Instead prioritization and reduction through e.g. similarity analysis removes poorly created test cases, at a loss of data (input) variation that can remove the original important test case intent, e.g. boundary value checks. The drive to

remove technical debt is low and pushed to the future. Refactoring test smells, reducing test code clones and utilizing e.g. mutation test approaches seems to be obvious steps to improve, but has a harsh reality to face, as code coverage are not considered mandatory (or only at statement level), and is often obfuscated by poor so called "test coverage" measurements.

Security is a strong driver for improved quality. There is an unwillingness to pay for what might be the real cause of test ignorance, e.g. lack of skills and poor test management awareness. Skills in software testing (and test design techniques) are still remaining low, with low status in industries, especially if software is not the main product. There is though a new hope that by introducing more gaming qualities into testing tools and more "intelligence" hidden in the tools fueled by AI/ML approaches, will diminish some of the of costly manual debugging and need for thorough testing. The drive to achieve automatic program repair will push the industry to be more aware of the new methodologies and utilize technologies that can change us enough to want to change how we solve software quality issues.

References

1. CodeChecker at github. https://github.com/Ericsson/codechecker. Accessed 05 May 2022
2. Ericsson Smart Mining web-page and report. https://www.ericsson.com/en/enterprise/reports/connected-mining. Accessed 08 Aug 2022
3. ETSI, European Standard. https://www.etsi.org. Accessed 05 May 2022
4. Github tool. https://github.com. Accessed 05 May 2022
5. ISO: ISO/IEC 25000: 2014, Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - Guide to SQuaRE webpage. https://www.iso.org/standard/35733.html. Accessed 08 Aug 2022
6. ITU-T TTCN-3 Z-series Z.161-Z.169. https://www.itu.int/rec/T-REC-Z/en. Accessed 08 Aug 2022
7. Jenkins tool. https://www.jenkins.io/. Accessed 05 May 2022
8. Maven tool. https://maven.apache.org. Accessed 05 June 2022
9. SonarCube tool. Accessed 08 May 2022
10. Agarwal, A., Gupta, S., Choudhury, T.: Continuous and integrated software development using DevOps. In: 2018 International Conference on Advances in Computing and Communication Engineering (ICACCE), pp. 290–293. IEEE (2018)
11. Ahmad, A., Leifler, O., Sandahl, K.: Empirical analysis of factors and their effect on test flakiness-practitioners' perceptions. arXiv preprint arXiv:1906.00673 (2019)
12. Al-Ahmad, A.S., Kahtan, H., Hujainah, F., Jalab, H.A.: Systematic literature review on penetration testing for mobile cloud computing applications. IEEE Access **7**, 173524–173540 (2019)
13. Ali, S., Briand, L.C., Hemmati, H., Panesar-Walawege, R.K.: A systematic review of the application and empirical investigation of search-based test case generation. IEEE Trans. Software Eng. **36**(6), 742–762 (2009)
14. Barboni, M., Bertolino, A., De Angelis, G.: What we talk about when we talk about software test flakiness. In: Paiva, A.C.R., Cavalli, A.R., Ventura Martins, P., Pérez-Castillo, R. (eds.) QUATIC 2021. CCIS, vol. 1439, pp. 29–39. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-85347-1_3

15. Battina, D.S.: Artificial intelligence in software test automation: a systematic literature review. Int. J. Emerging Technol. Innov. Res. (2019). https://www.jetir.org. UGC and ISSN Approved. ISSN 2349-5162
16. Bjarnason, E., et al.: Challenges and practices in aligning requirements with verification and validation: a case study of six companies. Empir. Softw. Eng. **19**(6), 1809–1855 (2014)
17. van Bladel, B., Demeyer, S.: A novel approach for detecting type-IV clones in test code. In: 2019 IEEE 13th International Workshop on Software Clones (IWSC), pp. 8–12. IEEE (2019)
18. van Bladel, B., Demeyer, S.: Clone detection in test code: an empirical evaluation. In: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 492–500. IEEE (2020)
19. Chen, T.Y., et al.: Metamorphic testing: a review of challenges and opportunities. ACM Comput. Surv. (CSUR) **51**(1), 1–27 (2018)
20. Collins, E., Dias-Neto, A., de Lucena, V.F., Jr.: Strategies for agile software testing automation: an industrial experience. In: 2012 IEEE 36th Annual Computer Software and Applications Conference Workshops, pp. 440–445. IEEE (2012)
21. Cordy, J.R., Roy, C.K.: The NiCad clone detector. In: 2011 IEEE 19th International Conference on Program Comprehension, pp. 219–220. IEEE (2011)
22. Diebold, P., Mayer, U.: On the usage and benefits of agile methods & practices. In: Baumeister, H., Lichter, H., Riebisch, M. (eds.) XP 2017. LNBIP, vol. 283, pp. 243–250. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57633-6_16
23. Dietrich, J., Rasheed, S., Tahir, A.: Flaky test sanitisation via on-the-fly assumption inference for tests with network dependencies. arXiv preprint arXiv:2208.01106 (2022)
24. Dikert, K., Paasivaara, M., Lassenius, C.: Challenges and success factors for large-scale agile transformations: a systematic literature review. J. Syst. Softw. **119**, 87–108 (2016)
25. Eldh, S.: On test design. Ph.D. thesis, Mälardalen University (2011)
26. Eldh, S.: Test automation improvement model-TAIM 2.0. In: 2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 334–337. IEEE (2020)
27. Eldh, S., Punnekkat, S., Hansson, H., Jönsson, P.: Component testing is not enough - a study of software faults in telecom middleware. In: Petrenko, A., Veanes, M., Tretmans, J., Grieskamp, W. (eds.) FATES/TestCom -2007. LNCS, vol. 4581, pp. 74–89. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73066-8_6
28. Engelman, C.: MATHLAB: a program for on-line machine assistance in symbolic computations. In: Proceedings of the November 30–December 1, 1965, Fall Joint Computer Conference, Part II: Computers: Their Impact on Society, pp. 117–126 (1965)
29. Estdale, J., Georgiadou, E.: Applying the ISO/IEC 25010 quality models to software product. In: Larrucea, X., Santamaria, I., O'Connor, R.V., Messnarz, R. (eds.) EuroSPI 2018. CCIS, vol. 896, pp. 492–503. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-97925-0_42
30. ETSI: Methods for Testing and Specification (MTS); UML 2.0 action syntax feasibility study TR 102 205 v1.1.1 (2003–2005)
31. Felderer, M., Büchler, M., Johns, M., Brucker, A.D., Breu, R., Pretschner, A.: Security testing: a survey. Adv. Comput. **101**, 1–51 (2016)
32. Feldt, R.: Do system test cases grow old? In: 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation, pp. 343–352. IEEE (2014)

33. Florea, R., Stray, V.: A global view on the hard skills and testing tools in software testing. In: 2019 ACM/IEEE 14th International Conference on Global Software Engineering (ICGSE), pp. 143–151. IEEE (2019)
34. Garousi, V., Zhi, J.: A survey of software testing practices in Canada. J. Syst. Softw. **86**(5), 1354–1376 (2013)
35. Grabowski, J., Hogrefe, D., Réthy, G., Schieferdecker, I., Wiles, A., Willcock, C.: An introduction to the testing and test control notation (TTCN-3). Comput. Netw. **42**(3), 375–403 (2003)
36. Grindal, M., Offutt, J., Mellin, J.: On the testing maturity of software producing organizations. In: Testing: Academic & Industrial Conference-Practice and Research Techniques (TAIC PART 2006), pp. 171–180. IEEE (2006)
37. Haindl, P., Plösch, R.: Towards continuous quality: measuring and evaluating feature-dependent non-functional requirements in DevOps. In: 2019 IEEE International Conference on Software Architecture Companion (ICSA-C), pp. 91–94. IEEE (2019)
38. Harman, M., McMinn, P., de Souza, J.T., Yoo, S.: Search based software engineering: techniques, taxonomy, tutorial. In: Meyer, B., Nordio, M. (eds.) LASER 2008-2010. LNCS, vol. 7007, pp. 1–59. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-25231-0_1
39. Hasanain, W., Labiche, Y., Eldh, S.: An analysis of complex industrial test code using clone analysis. In: 2018 IEEE International Conference on Software Quality, Reliability and Security (QRS), pp. 482–489. IEEE (2018)
40. Jonsson, L.: Machine Learning-Based Bug Handling in Large-Scale Software Development, vol. 1936. Linköping University Electronic Press (2018)
41. Jonsson, L., Borg, M., Broman, D., Sandahl, K., Eldh, S., Runeson, P.: Automated bug assignment: ensemble-based machine learning in large scale industrial contexts. Empir. Softw. Eng. **21**(4), 1533–1578 (2016)
42. Jonsson, L., Broman, D., Sandahl, K., Eldh, S.: Towards automated anomaly report assignment in large complex systems using stacked generalization. In: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, pp. 437–446. IEEE (2012)
43. Karris, S.T.: Introduction to Simulink with Engineering Applications. Orchard Publications (2006)
44. Kaur, K., Jajoo, A., et al.: Applying agile methodologies in industry projects: benefits and challenges. In: 2015 International Conference on Computing Communication Control and Automation, pp. 832–836. IEEE (2015)
45. Kintis, M., Papadakis, M., Malevris, N.: Evaluating mutation testing alternatives: a collateral experiment. In: 2010 Asia Pacific Software Engineering Conference, pp. 300–309. IEEE (2010)
46. Kitanov, S., Monteiro, E., Janevski, T.: 5G and the fog-survey of related technologies and research directions. In: 2016 18th Mediterranean Electrotechnical Conference (MELECON), pp. 1–6. IEEE (2016)
47. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization, CGO 2004, pp. 75–86. IEEE (2004)
48. Malm, J., Causevic, A., Lisper, B., Eldh, S.: Automated analysis of flakiness-mitigating delays. In: Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test, pp. 81–84 (2020)
49. Marick, B.: How to misuse code coverage. https://www.exampler.com/testing-com/writings/coverage.pdf. Accessed 05 May 2022

50. Mårtensson, T., Ståhl, D., Bosch, J.: Exploratory testing of large-scale systems – testing in the continuous integration and delivery pipeline. In: Felderer, M., Méndez Fernández, D., Turhan, B., Kalinowski, M., Sarro, F., Winkler, D. (eds.) PROFES 2017. LNCS, vol. 10611, pp. 368–384. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-69926-4_26
51. McMinn, P.: Search-based software test data generation: a survey. Softw. Test. Verif. Reliab. **14**(2), 105–156 (2004)
52. Navabi, Z.: VHDL: Analysis and Modeling of Digital Systems, vol. 2. McGraw-Hill, New York (1993)
53. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: towards a standard CP modelling language. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 529–543. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74970-7_38
54. Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Le Traon, Y., Harman, M.: Mutation testing advances: an analysis and survey. Adv. Comput. **112**, 275–378 (2019)
55. Parry, O., Kapfhammer, G.M., Hilton, M., McMinn, P.: A survey of flaky tests. ACM Trans. Softw. Eng. Methodol. (TOSEM) **31**(1), 1–74 (2021)
56. Petrović, G., Ivanković, M., Fraser, G., Just, R.: Does mutation testing improve testing practices? In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp. 910–921. IEEE (2021)
57. Pietrantuono, R., Bertolino, A., De Angelis, G., Miranda, B., Russo, S.: Towards continuous software reliability testing in DevOps. In: 2019 IEEE/ACM 14th International Workshop on Automation of Software Test (AST), pp. 21–27. IEEE (2019)
58. Planning, S.: The economic impacts of inadequate infrastructure for software testing. National Institute of Standards and Technology, p. 1 (2002)
59. Porkoláb, Z., Brunner, T.: The codecompass comprehension framework. In: Proceedings of the 26th Conference on Program Comprehension, pp. 393–396 (2018)
60. Rodríguez, P., Markkula, J., Oivo, M., Turula, K.: Survey on agile and lean usage in Finnish software industry. In: Proceedings of the 2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, pp. 139–148. IEEE (2012)
61. Roy, C.K., Cordy, J.R.: A survey on software clone detection research. Queen's Sch. Comput. TR **541**(115), 64–68 (2007)
62. Saxena, P.: OSI reference model - a seven layered architecture of OSI model. Int. J. Res. **1**(10), 1145–1156 (2014)
63. Schulte, E., DiLorenzo, J., Weimer, W., Forrest, S.: Automated repair of binary and assembly programs for cooperating embedded devices. ACM SIGARCH Comput. Archit. News **41**(1), 317–328 (2013)
64. Segura, S., Fraser, G., Sanchez, A.B., Ruiz-Cortés, A.: A survey on metamorphic testing. IEEE Trans. Software Eng. **42**(9), 805–824 (2016)
65. Shahin, M., Babar, M.A., Zhu, L.: Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. IEEE Access **5**, 3909–3943 (2017)
66. Szabó, J.Z., Csöndes, T.: Titan, TTCN-3 test execution environment. Infocommun. J. **62**(1), 27–31 (2007)
67. Tillmann, N., De Halleux, J., Xie, T., Gulwani, S., Bishop, J.: Teaching and learning programming and software engineering via interactive gaming. In: 2013 35th International Conference on Software Engineering (ICSE), pp. 1117–1126. IEEE (2013)

68. Van Deursen, A., Moonen, L., Van Den Bergh, A., Kok, G.: Refactoring test code. In: Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001), pp. 92–95. Citeseer (2001)
69. Wegener, J., Baresel, A., Sthamer, H.: Evolutionary test environment for automatic structural testing. Inf. Softw. Technol. **43**(14), 841–854 (2001)
70. Weimer, W., Forrest, S., Le Goues, C., Nguyen, T.: Automatic program repair with evolutionary computation. Commun. ACM **53**(5), 109–116 (2010)
71. Wiklund, K., Eldh, S., Sundmark, D., Lundqvist, K.: Technical debt in test automation. In: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, pp. 887–892. IEEE (2012)
72. Xie, T., Tillmann, N., De Halleux, J.: Educational software engineering: where software engineering, education, and gaming meet. In: 2013 3rd International Workshop on Games and Software Engineering: Engineering Computer Games to Enable Positive, Progressive Change (GAS), pp. 36–39. IEEE (2013)
73. Yang, J., Zhikhartsev, A., Liu, Y., Tan, L.: Better test cases for better automated program repair. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pp. 831–841 (2017)
74. Zaidman, A., Van Rompaey, B., Demeyer, S., Van Deursen, A.: Mining software repositories to study co-evolution of production & test code. In: 2008 1st International Conference on Software Testing, Verification, and Validation, pp. 220–229. IEEE (2008)
75. Zeller, A., Gopinath, R., Böhme, M., Fraser, G., Holler, C.: The fuzzing book (2019)
76. Zhu, H., Hall, P.A., May, J.H.: Software unit test coverage and adequacy. ACM Comput. Surv. (CSUR) **29**(4), 366–427 (1997)