2012 IEEE Fifth International Conference on Software Testing, Verification and Validation

Technical Debt in Test Automation

Kristian Wiklund, Sigrid Eldh
Ericsson AB
SE-164 80 KISTA
Stockholm, Sweden
{kristian.wiklund,sigrid.eldh}@ericsson.com

Daniel Sundmark, Kristina Lundqvist
School of Innovation, Design and Engineering
Mälardalen University
Västerås, Sweden
{daniel.sundmark,kristina.lundqvist}@mdh.se

***Abstract*—Automated test execution is one of the more popular and available strategies to minimize the cost for software testing, and is also becoming one of the central concepts in modern software development as methods such as test-driven development gain popularity. Published studies on test automation indicate that the maintenance and development of test automation tools commonly encounter problems due to unforeseen issues. To further investigate this, we performed a case study on a telecommunication subsystem to seek factors that contribute to inefficiencies in use, maintenance, and development of the automated testing performed within the scope of responsibility of a software design team. A qualitative evaluation of the findings indicates that the main areas of improvement in this case are in the fields of interaction design and general software design principles, as applied to test execution system development.**

***Index Terms*—software testing;test automation;technical debt;case study;empirical study**

I. Introduction

”Technical debt” is a methaphor that has gained popularity in the agile development community to describe the gap between the current state of the system and the ideal state of the system [1], usually in situations where a compromise is made during development to meet demands in one dimension, such as lead time, by sacrificing work in another dimension, such as architecture. The concept was first used by Cunningham [2] to describe situations where long-term code quality is traded for short-term gains in development speed.

The metaphor compares design choices to economics: by not doing something during development, one takes out a loan on the system and pays for that loan by future costs caused by the design choice. One may then choose to accept the cost or, if feasible, pay off the debt through re-work [1].

Much like financial debt, a technical debt can be necessary and well-motivated to be able to move forward with the business at hand.

It should be noted that one of the main differences to financial debt is that while financial debt occurs from a deliberate action of taking a loan, technical debt can be caused by factors outside the control of the development team [1], which in turn may cause the debt to be invisible to all or parts of an organization.

That technical debt needs to be further researched is proposed by Brown *et al.* [1], who are suggesting field studies to determine the relative importance of sources of debt, what sources have the highest impact, and how the debt is handled.

Turning to software testing, it is a time- and resource-consuming activity that has been estimated to use up to 80% of the total development cost [3]. A strategy for reducing this cost is to apply test automation, which potentially provides great savings in the software development process [4], [5]. Test automation is also one of the key components for organizations implementing development methods such as test-driven development [6].

Several studies on test automation [7]–[11] indicate that there could be a general trend of high technical debt present in many test execution automation implementations, which cause problems for the use, extension, and maintenance of those systems.

Our objective is to further the knowledge on the dominant sources of technical debt with respect to test execution automation. In this paper, we will make three contributions: an overview of case studies related to test execution automation, an overview of test process improvement models and how they adress test infrastructure, and finally the results from a case study investigating potential sources for technical debt in test execution automation systems.

To achieve this, we have conducted a series of semi-structured interviews with software designers working on a subsystem in a large complex telecommunication system, leading to four observations on potential contributors to the technical debt and a proposal on further work to deepen the understanding of the area.

The paper is structured as follows: The background of the study and a theory that forms the reason for the work is described in Section II, followed by an overview of related work in Section III. Section IV elaborates on the study design, Section V describes the context of the study, followed by our observations in Section VI, and the conclusions and proposed future work in Section VII.

II. Background

Bertolino [12] identifies 100% test automation as one of the dreams in her roadmap for test research from 2007, and states that ”far-reaching automation is one of the ways to keep quality analysis and testing in line with the growing quantity and complexity of software”. Bertolino touches on the quantity and complexity of *automation itself* by mentioning the large quantity of code needed for unit test result checking and stimuli generation.

978-0-7695-4670-4/12 $26.00 © 2012 IEEE
DOI 10.1109/ICST.2012.192

That issues with testing tools may be common is indicated in a study by Kasurinen *et al.* [10]. In their list of problems "complicated tools cause errors" is stated in three out of five cases, and "commercial tools have limited usablity" is stated in the remaining two cases. As part of the result from their study, they hypothesize that "the selection of testing tools and testing automation should focus on the usability and configurability of possible tools". Parts of these issues were considered by Fewster and Graham [13] in their textbook, where they point out that commercial test tools are complex mechanism that require detailed technical knowledge to be fully useful, and that it is important to consider the actual requirements and constraints of the organization before evaluating and selecting a tool.

Karhu *et al.* [8] note that "automated software testing may reduce costs and improve quality because of more testing in less time, but it causes new costs in, for example, implementation, maintenance, and training". They continue by stating that "automated testing systems consist of hardware and software and suffer from the same issues as any other systems", unfortunately it is not elaborated on what these issues are, except for mentioning unreliability problems.

An investigation into common risks introduced when automating tests was done by Persson and Yilmaztürk [9], and suggests a list of commonly encountered risks for automation deployment. The list spans over a relatively large area of work related to automated testing, and among the included risks, we find lack of specification prior to scripting test cases, insufficient configuration management for test environment and testware, insufficient defect tracking for the test environment and testware, ignoring that test automation is software development, missing functionality in the test tools used, and lack of development guidelines for test environment and testware. Testware is defined by Gelperin *et al.* [14] as "the actual testing procedures that are run, the support software, the data sets that are used to run the tests, and the supporting documentation".

The risk list was identified by a literature study and later evaluated in, and expanded with, findings from two test automation projects executed at the Swedish power systems and industrial automation group ABB. The list contains a lot of potential issues concerning both the use of the test systems, as well as the implementation of the test systems. In the paper, it is reported that even though the risks were known in advance, the studied projects did not manage to mitigate the risks. In fact, only parts of the risk list was actually entered into the formal risk management process.

Berner *et al.* [11] study five projects, and reports issues with test environment usage and test environment design. Among the findings we have lack of documentation, lack of test system architecture, and a general lack of understanding that a test automation organization produce a piece of software that need to be tested itself and maintained.

Considering these studies, as well as anecdotal evidence readily available from practitioners, we observed 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 systemization, documentation, and testing, when producing or deploying test automation systems*.

III. Related Work

Looking for answers on how to remedy the theoretical situation, we turned to the literature once again and encountered the various process improvement models adressing testing issues that are available to the practitioner. As seen in the overview below, these models commonly have test systems and test automation as parts of a much larger scope, leaving the requirements on automation on a relatively high level.

Gelperin and Hayashi [14] state, in a paper describing the "Testability Maturity Model" (TMM), that the primary issue with testware is reuse, and that to enable reuse, one must support the testware by configuration management, modularization, and independence. It is also recommended that the test environment shall be automated, flexible and easy to use.

The importance of test infrastructure is emphasized by Jacobs *et al.* [15] who discusses the Testability Maturity Model and claims that the test system is paramount in testing and must be adressed in any test process improvement model. However, no new requirements are proposed to expand the Testability Maturity model.

In "TIM - A Test Improvement Model" [16], Ericson *et al.* dedicate one of the five key areas to testware. The key area is, apart from a requirement on configuration management, primarily concerned about what the testware does, such as automated result checking, and not as much about how to handle the testware. One additional requirement is added in the review key area where it is recommended that the testware shall be reviewed.

Another test process improvement model that uses a number of checklists to define maturity levels and suggest improvements to the process is "TPI" [17]. TPI promotes reusable testware, centrally managed testware with a testware product owner, configuration management, proper delivery procedures for testware, and testware architecture. Heiskanen *et al.* [18] build on TPI to adress concerns for automated test generation. In the field of testware management, they include evaluation of when to automate, and also propose that a test environment specialist is included in the staff.

TPI and TMM are reviewed by Blaschke *et al.* [19] according to the requirements of the ISO/IEC 15504 standard. This leads to the definition of a new reference model in line with the standard, where they introduce "Test Environment Operation" as one of the four primary life cycle processes of the model with sub-areas in user support and operational use of the test environment. The details are not provided, and the sub-area is present only as a slogan.

Karlström *at al.* [20] propose a test practice framework useful for small emerging organizations, and emphazises the availability of the test environment when it is needed. The paper draws from a number of case-studies that shows the importance of test environment handling. One success factor

that is pointed out is a fast test environment with a known and specified design. Another important observation is that the test environment itself must be tested after setup, and that this practice was the hardest to get in place, as test environments appeared to be taken for granted by the developers.

To summarize, when studying the cited work, we found no clear, detailed, easily used guidelines on how to design, implement, and maintain an automated test execution system in a way that keep the accumulated technical debt on an acceptable level, and provides the maximum benefit to the organization using the system. We also concluded, given the cited case studies on software testing and test automation in Section II, that there is an actual need among software development and software testing practicioners for such guidelines.

IV. Case Study Design

The study consists of a series of interviews combined with inspection of a number of documents, design rules for test scripts, quality guidelines for the studied application, and improvements identified within the studied teams.

A. Objective

The objective of the study was to identify common contributors to the technical debt accumulated in automated testing, and to evaluate the awareness of this debt in organizations using automated test.

B. Research Question

RQ1: What practices in a software design organization are the primary contributors to technical debt present in test execution automation?

C. Case Study Method

Six one-hour semi-structured interviews, using an interview guide that was evolved during the process, were performed with software designers working with the studied application, and test environment engineers from the organization providing the continuous integration framework that is used by the software design team. The subjects were identified both by the researchers and by the responsible manager, who was asked to provide a mix of team leaders, junior and senior engineers. Three software designers and three test environment engineers were interviewed.

The interviews were performed within a period of one week by the principal researcher who also acted as scribe.

After each interview, the notes were checked for readability and any ambiguities were clarified using a pen in different color, while the interview was still fresh in memory of the interviewer. Then, the notes were scanned to pdf format and a preliminary classification of the contents was applied using a pdf annotation tool, with the primary purpose of marking areas of the interview that were relevant for the research question. This preliminary classification was used to evolve and validate the interview guide between interviews, to be able to probe for related information during the subsequent interviews.

When all interviews were completed, the notes were studied again in the context of all interviews, to make sure that no relevant data was left out from the first classification. If any further data of relevance to the research question was identified, it was marked as well, to include all segments that contribute to the analysis with respect to the research question. Following this, all marked areas were transcribed from the notes to one text file per subject. The reason for this transcription was to allow use of the free computer assisted qualitative data analysis software (CAQDAS) application RQDA [21] to simplify the analysis of the collected material. The benefits brought by RQDA is support to perform overlapping labeling, or coding, of text segments in a file, and functions to extract similarily labelled segments from all files in the study.

To analyze the data from the interviews, "thematic coding" as described by Robson [22] was used. This is an iterative method to identify common themes in the data, by classifying behaviors, events, activities or other concepts that are considered to influence the studied phenomenon. The themes are used to group segments of data, and these groups can then be examined to find repetitions, similarities, differences, anomalies, or connections to the theory, that are further analyzed to provide the results from the study.

In doing this, we started with the preliminary codes applied during interview post-processing, then revising and expanding the code-set during the work, to find appropriate matchings between segments and identify themes. During this work, the initial notes were revisited to provide context, when necessary, and the software package was used to provide a view of the themes as they formed.

V. The Case Study Object

A. Overview

The case study object is test execution automation performed to design, test, and integrate a subsystem in a large complex telecom system. The subsystem is part of an embedded system and interacts with its environment through machine-machine interfaces only. Occasionally, manual testing is done to exercise special cases, but the majority of the day-to-day testing is performed via an automated system.

The scope of the study is the testing performed from the moment the programmer writes the code, to the integration tests performed when the application is integrated in its real environment. The conceptual process for this work is shown in Figure 1 and will be further described below.

B. Test Environments

Selection of the case study object was done on the basis of being the first software application in the larger system moving to full responsibility for design- and integration testing. Previously, the test responsibility was divided over two teams, with the design team being responsible for the tests performed in the simulated environment, and the integration team being responsible for integration tests using the real target system. This was considered an impediment for quick feedback and agility, and the test responsibility was shifted to design, while

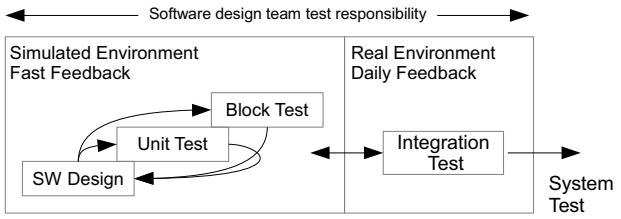

Fig. 1. Conceptual Test Flow

keeping tool responsibility as a support function that produces a common, shared infrastructure for automation of software builds, test execution, and result reporting.

C. Test Process

The conceptual test work-flow is shown in Figure 1, and consists of three activities performed by the design team followed by further system testing that is out of the scope for the design team and not included in this study.

Unit tests, also known as component tests [23], are used to test the individual software components and gain test coverage. The primary purpose of the unit testing is to find problems as early as possible by providing rapid feedback to the programmer. Unit tests typically operate on individual functions, objects, or procedures in the design.

Following successful unit testing, ”block tests” are used to test the interface behavior of several components integrated into larger modules. These modules are defined during subsystem design, and have well-defined interfaces that are accessed by the test system using the same operating system events that are later used in the final integrated product.

The block tests are followed by integration testing, where the entire subsystem is integrated and tested as a complete product in its natural environment, allowing interoperability testing with the peers of the subsystem. This integrated system is tested through the actual interfaces used when it is deployed as a component in a larger telecommunications system and the testing performed is derived from the requirements on the product.

The first two test activities are performed in a so-called ”host test” environment, executing (”hosted”) on the developer’s own work-station.

The unit tests are performed by cross-compiling the application for a unix-like environment where the tests are executed. To simplify the work and provide standardization between developers, a third-party unit test framework is used for writing unit test cases, executing them, and following up the results.

The block test environment uses an operating system simulation library, essentialy a number of stubs, that provide the target operating system application programming interface (API) to the block under test. The library and the block under test are cross-compiled together to execute on a unix-like system, providing a functional behavior as if it was executing on a real system. The purpose of the simulation is to enable functional testing only. Non-functional aspects, such as timing or memory, are not accurately simulated.

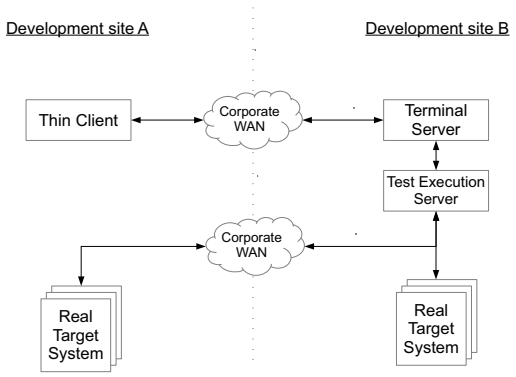

Fig. 2. Overview of Test System Localization on Development Sites

The third test activity uses a real target system and the real delivery candidate of the application. It is exercised by a test system executing test scripts on a unix-like server that is connected to the system under test via the same interfaces as that system is controlled by in the final deployed telecommunications system. Using a real system for the testing allows non-functional properties to be inspected, such as response time or memory use.

The target test system is executing in a test facility that is shared with other products and test activities, and the entire test environment is provided by a test environment team. As the software developers are located on a different site than the shared test facility and sometimes need physical access to interface a target system, a replicated configuration is available on-site for their use. The connections between the sites and placement of equipment is shown in Figure 2.

VI. Observations

Four main observations were identified through the use of thematic coding [22] and are presented in this section where they are discussed together with related work.

A. Observation 1: Reuse and sharing of test tools brings issues that need to be considered

The test environments for the studied product are based on significant reuse of test tools, test scripts, and equipment used by other parts of the company, as sharing test tools between products is considered by the organization to provide several benefits. It is also recommended by Fewster and Graham [13], as well as several of the test process maturity or improvement models described in Section III. Reuse enables, for example, cost sharing by having the procurement, development, and maintenance effort as a support function outside the teams. Having one common tool-set is also believed to simplify staff mobility and workload sharing, as it allows focusing on the

product instead of the development tools when moving to a new team.

Several of the subjects had concerns on restrictions or inflexibility of the shared tool-set, where some tools were not easily adapted to the task at hand. One specific issue, where the third-party block test tool was discovered to not handle the signaling behavior of the product lead to an adjustment of the test strategy to use primarily unit tests, whereas the original intent was to use primarily block tests.

This is related to the finding by Persson and Yilmaztürk [9] in their ”tool evaluation and selection” pitfall that states that it is hard to evaluate tools detailed enough and that there is a risk that one acquire at tool that later is found to be insufficient.

B. Observation 2: Test facility infrastructure is not transparent and may alter the test results if not accounted for

To simplify the product development, the programmers have access to a number of local target systems that can be used when physical access to actual hardware is needed. In the environment used, this leads to unpredictable behavior during test execution when the development moved from local workstations to remote terminal servers on another development site, as shown in Figure 2. The move altered the timing between the local target systems and the now remote test execution server, which in turn caused the test execution system to become instable and unpredictable.

The issue was identified as some test cases started to exhibit an unstable behavior. By comparing test results from test runs using target systems on development site A with test results from test runs using target systems on development site B the problems were determined to be site-specific and inspection of the test cases revealed that they had been designed to be stable when using development site A’s equipment which is local to the development team.

C. Observation 3: Generalist engineers expect that their tools are easy to use

Several tool were reported to be hard to use and providing too much freedom at the expense of ease of use, and was suggested to be improved by wrapping them in a push-button solution for the most common usage scenarios. Another observation from the interviews is that the quality of shared tools is expected to be higher and better communicated than tools that were developed or procured by the team itself. Shared tools were expected to ”just work” and be transparent.

Looking at the concept of interaction design, Donahue *et al.* [24] report that ”people try to avoid using stressful systems”, and that when using them productivity suffers. In ”The Design of Everyday Things” [25], Norman observes that problems seen as trivial will not be reported, as the users tend to blame it on themselves. He continues by stating that if errors are possible, someone will make them.

This suggests that in a ”generalist” approach where all team members perform approximately the same work, from design systemization to testing, both the need and the tolerance for advanced and highly configurable tooling will be lower. Test tasks tend to be mostly routine, and the tooling need to be designed in a way that minimizes lead time to get started. The contribution to the technical debt is in the field of usability, by not being designed to suit the users, an ”interest” will be paid with every use.

D. Observation 4: Accepted development practices for test code are potentially less rigorous than for production code

Test case documentation was reported to be missing in many cases, making maintenance hard. Also, some test scripts had been designed to specifically compensate for the timing issues reported in observation 3. Further investigation revealed that design rules for test code is being produced by the tool team to adress this issue, and were scheduled to be deployed.

When time is short, a higher ”technical debt” [2] is perceived by the subjects to be accepted in the test code than in production code.

As a final checklist prior to delivery, the development team has a ”definition of done” that is followed. According to Schwaber [26], something is ”done” when it is ”complete as mutually agreed to by all parties and conforming to an organization’s standards, conventions, and guidelines”. As part of the definition of done, the studied software development team is using a quality assurance checklist that specifies what actions to do with the software. The actions are ordered in priority order to make sure that the actions considered most important are done first. This priority order puts less priority on test code inspection as compared to production code inspection. This was initially considered a potential problem and probed for in the interviews. In reality, it turned out that all code, including the test scripts, is pair programmed, which means that this likely is a non-issue, as pair-programming outperforms traditional inspection, at least in an experimental setting [27].

Regardless of the actual process used, the priority differences in the *defined process* is in line with Tassey [28] where it is reported that testing suffers when time is short and ”speed of deployment” is identified as one of the driving forces behind problems related to software testing.

Not treating test automation as software development is one of Persson and Yilmaztürk’s risks [9], and according to Jacobs [15], test infrastructure should be controlled and managed. Hoffman [29] tells us that test automation is more than computers launching test program and stresses the need for test system architecture.

That test systems needs to be treated as any software product was also pointed out by Berner *et al.* [11] after observing five different software projects. However, Berner also concludes that some ”fussiness” may be allowed in a test system as problems can be corrected more easily.

VII. Conclusions and Future Work

In this paper, we make three contributions. We have presented a brief overview of case studies related to test execution automation. We have also presented an overview of test process improvement models, and test process maturity models,

investigating each cited model for requirements on test execution automation systems. Finally, we presented the results from a case study investigating test execution automation in the context of a subsystem of a large complex telecommunications system.

In studying the cited work, we found no clear, detailed, guidelines on how to design, implement, and maintain an automated test execution system in a way that keep the accumulated technical debt on an acceptable level, and provides the maximum benefit to the organization using the system. We also concluded from the cited case studies on software testing and test automation, that there is an actual need among software development and software testing practicioners for such guidelines.

From the qualitative analysis of the case study, we contribute four observations:

Observation 1 concerns effects caused by sharing of tools, which in turn is motivated by the perceived positive effects of lower costs and higher mobility of engineers between products. We would like to investigate the validity of this in the general situation and investigate if *tool sharing make a positive contribution to the effort or not?*

Observations 1 and 3 concerns the *usability of the test automation systems* and may form an interesting challenge for the interaction design community, as the systems need to be fast and flexible for the power users, while still being kind to the new employee right out of the university.

Finally, returning to the technical debt we set out to investigate, observations 2 and 4 concerns the *use of software design principles* when producing and maintaining test automation systems. We have seen that there are differences in attitudes towards the test code as compared to the production code, which also has been reported by other studies.

Hence, we would like to perform a deepened investigation to find out *what the underlying reasons for this phenomenon are, how to mitigate those reasons, and what the priority of each preventative action should be.*

ACKNOWLEDGMENTS

The work was performed within the frame of ITS-EASY, an industrial research school in Embedded Software and Systems, affiliated with the School of Innovation, Design and Engineering at Mälardalen University, Sweden. The work was funded by Ericsson AB, Mälardalen University, and the Knowledge Foundation.

REFERENCES

[1] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, R. Sangwan, C. Seaman, K. Sullivan, and N. Zazworka, "Managing technical debt in software-reliant systems," in *Proceedings of the FSE/SDP workshop on Future of software engineering research*. New York, New York, USA: ACM, 2010, pp. 47–52.

[2] W. Cunningham, "Experience Report - The WyCash Portfolio Management System," in *OOPSLA'92*, no. October, 1992.

[3] H. Leung and L. White, "A cost model to compare regression test strategies," in *Software Maintenance, 1991., Proceedings. Conference on*. IEEE, 1991, pp. 201–208.

[4] L. Damm and L. Lundberg, "Results from introducing component-level test automation and test-driven development," *Journal of Systems and Software*, vol. 79, no. 7, pp. 1001–1014, 2006.

[5] T. Wissink and C. Amaro, "Successful Test Automation for Software Maintenance," in *Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on*. IEEE, 2006, pp. 265–266.

[6] D. Janzen and H. Saiedian, "Test-driven development concepts, taxonomy, and future direction," *Computer*, vol. 38, no. 9, pp. 43– 50, Sep. 2005.

[7] J. Kasurinen, O. Taipale, K. Smolander, K. Jussi, T. Ossi, and S. Kari, "Software test automation in practice: empirical observations," *Advances in Software Engineering*, vol. 2010, 2010.

[8] K. Karhu, T. Repo, O. Taipale, and K. Smolander, "Empirical Observations on Software Testing Automation," in *2009 International Conference on Software Testing Verification and Validation*. IEEE, Apr. 2009, pp. 201–209.

[9] C. Persson and N. Yilmazturk, "Establishment of automated regression testing at ABB: industrial experience report on'avoiding the pitfalls'," in *Automated Software Engineering, 2004. Proceedings. 19th International Conference on*. IEEE, Sep. 2004, pp. 112–121.

[10] J. Kasurinen, O. Taipale, and K. Smolander, "Analysis of Problems in Testing Practices," in *16th Asia-Pacific Software Engineering Conference*. Los Alamitos, CA, USA: IEEE Computer Society, 2009, pp. 309–315.

[11] S. Berner, R. Weber, and R. Keller, "Observations and lessons learned from automated testing," in *Proceedings of the 27th international conference on Software engineering*. ACM, 2005, pp. 571–579.

[12] A. Bertolino, "Software Testing Research: Achievements, Challenges, Dreams," in *Future of Software Engineering (FOSE '07)*. Los Alamitos, CA, USA: IEEE Computer Society, 2007, pp. 85–103.

[13] M. Fewster and D. Graham, *Software test automation: effective use of test execution tools*. ACM Press/Addison-Wesley, 1999.

[14] D. Gelperin, "How to support better software testing," *Application Development Trends*, no. May, 1996.

[15] J. Jacobs, J. van Moll, and T. Stokes, "The Process of Test Process Improvement," *XOOTIC Magazine*, vol. 8, no. 2, pp. 23–29, 2000.

[16] T. Ericson, A. Subotic, and S. Ursing, "TIM - A Test Improvement Model," *Software Testing Verification and Reliability*, vol. 7, no. 4, pp. 229–246, 1997.

[17] T. Koomen and M. Pol, *Test process improvement: a practical step-by-step guide to structured testing*. Addison-Wesley Professional, 1999.

[18] H. Heiskanen, M. Maunumaa, and M. Katara, "Test Process Improvement for Automated Test Generation," Tampere University of Technology, Department of Software Systems, Tech. Rep., 2010.

[19] M. Blaschke, M. Philipp, and T. Schweigert, "The Test SPICE Approach."

[20] D. Karlström, P. Runeson, and S. Nordén, "A minimal test practice framework for emerging software organizations," *Software Testing, Verification and Reliability*, vol. 15, no. 3, pp. 145–166, Sep. 2005.

[21] R. Huang, "RQDA: R-based Qualitative Data Analysis. R package version 0.2-1. http://rqda.r-forge.r-project.org/," 2011.

[22] C. Robson, *Real world research, Third Edition*. Wiley Publishing, 2011.

[23] E. van Veenendal, Ed., *Standard glossary of terms used in Software Testing*, 2nd ed. International Software Testing Qualifications Board, 2010, vol. 1.

[24] G. Donahue, S. Weinschenk, and J. Nowicki, "Usability is good business," Compuware Corporation, Tech. Rep., 1999.

[25] D. A. Norman, *The Design of Everyday Things*. New York: Basic Books, 2002.

[26] K. Schwaber, *Agile Project Management with Scrum*. O'Reilly Media, Inc., 2009.

[27] J. Tomayko, "A comparison of pair programming to inspections for software defect reduction," *Computer Science Education*, no. January 2012, pp. 37–41, 2002.

[28] G. Tassey, "The economic impacts of inadequate infrastructure for software testing," National Institute of Standards and Technology, Tech. Rep. 7007, 2002.

[29] D. Hoffman, "Test Automation Architectures: Planning for Test Automation," *Quality Week*, 1999.