Stephan Goericke *Editor*

The Future of Software Quality Assurance

iSQI

Springer Open

The Future of Software Quality Assurance

Stephan Goericke
Editor

The Future of Software Quality Assurance

Springer Open

Editor
Stephan Goericke
iSQI GmbH
Potsdam
Germany

Translated from the Dutch Original book: 'AGILE', © 2018, Rini van Solingen & Management Impact – translation by tolingo GmbH, © 2019, Rini van Solingen

ISBN 978-3-030-29508-0 ISBN 978-3-030-29509-7 (eBook)
https://doi.org/10.1007/978-3-030-29509-7

This Springer imprint is published by the registered company Springer Nature Switzerland AG.
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Foreword

In June 2002 David Parnas, one of the pioneers of software engineering, wrote[1]:

> Software is well known for low reliability and lack of trustworthiness. In part this is attributable to the difficulty of dealing with the complexity of today's software systems, but the inadequate knowledge, skills, and professionalism of many of the practitioners also contributes to this problem. Moreover, we can thank the inadequately qualified people who produced today's software for the unreliability and complexity of the products that serve as support software for new products.

Dave sees the inadequate education of the people involved in software development as a major problem. In the field of software, lifelong education and training is essential to keep up to date with the rapid evolution. iSQI has been dealing with this problem for 15 years now and has certainly contributed in many areas to improving the education of people and thus the quality of the software.

The many contributions in this book clearly reflect the numerous ways software quality assurance can play a critical role in multiple areas.

About 75 years ago, the first electronic computers were used to calculate mathematical problems—also from the military sector. In the following years, computers increasingly took over routine tasks and tasks that required a high degree of precision. Due to technical progress and the associated speed of the systems, large amounts of data can now be processed and evaluated in a short time. The computer now dominates our everyday lives. It has become indispensable, because without it our life today would no longer be possible.

Today we are at a radical turning point. (So-called) artificial intelligence is used in more and more software systems, be it automatic speech processing, autonomous driving or machine learning. In many areas we no longer use software exclusively to make life easier, but also as a tool to help us make decisions or simply delegate decision-making tasks to the software itself.

[1] https://en.wikipedia.org/wiki/David_Parnas Citation from David Parnas's preface to the German book Spillner/Linz "Basiswissen Softwaretest".

So far many of these systems have been used as decisive aids; helping a person to make a more substantiated decision. For example, a physician can have an X-ray image analysed by a system and use the information obtained to make his or her decision. The question arises or will arise: "When is the decision completely left to the system, since the system results show a higher reliability of correctness of the diagnosis than diagnoses of doctors?" In other areas, such as autonomous driving, some decisions have to be made so quickly that they can only be made by the system itself.

In fact, it is not the system that decides, but the person who has programmed the system accordingly. At least that's how it has been so far. When artificial intelligence is used, however, this is no longer definite. With neural networks and deep learning, the results are no longer predictable and therefore no longer depend directly on the programming.

These few examples already show how crucial the quality of the systems is and thus the education of the people responsible for the development and testing of these systems. This is why iSQI will continue to be a sought-after partner for further training and certification in the coming years.

In addition to well-founded specialist knowledge, ethical issues will have to be the focus of attention in the coming years. How do we as a society—even as humankind—want to deal with the fact that autonomous weapons themselves decide which people are being killed?

We must all ask ourselves the question: "Which decisions must still be taken by people and which can we delegate to software systems?"

We, as participants in the development of software systems, have a particularly high responsibility for what the near future will look like and where it will go in the coming years.

I hope that we will take our responsibility and shape the future positively.

GTB, Hochschule Bremen, Bremen, Germany Andreas Spillner
July 2019

Preface

A Question of Quality

Software testing is not a regulated profession. In principle, anybody can become a tester. This was always the case and remains so today. However, there are standards that should apply to all testing professionals and which are a prerequisite for anyone wishing to be taken seriously as a software tester.

An early version of a standardized syllabus was developed in 1998 known as the "Certified Tester". Four years later Austria, Denmark, Finland, Germany, Sweden, Switzerland, the Netherlands and the UK established the International Software Qualification Board (ISTQB®) to define and promote a body of knowledge for the profession of software tester.

It is one matter to create a set of regulations; it is quite another to ensure that the regulations are observed. This is where a certifying body has a crucial role to play, and hence the International Software Quality Institute (iSQI). Exactly 15 years ago a working group known as ASQF (Arbeitskreis Software-Qualität und -Fortbildung e.V.) established iSQI as an independent body to serve as a safety net and guarantee for an uncompromising and consistent level of quality in the training and certification of specialists throughout the world. In the following 15 years the quality of the profession of software tester has improved immensely. Today there are almost 650,000 ISTQB®-certified software testers worldwide.

Quality Standards Are an Imperative

It is the standards maintained by specialists at every level of software production that accelerate improvements in quality. Both functional and non-functional 'quality' can only be achieved when people have a sense of what the term 'quality' really means and they know what they have to do to achieve this. Yet how can they do this when they have to deal with complicated testing in a limited time? And how

does this work when teams are working all over the globe, in different time zones? How can tests be scheduled, if results have to be delivered within an agile working regime or during the development phase?

The testing profession has changed considerably in the last 15 years, and today it faces enormous new challenges.

Automation and the use of artificial intelligence have brought changes to methods and to core areas. Here further improvements must be made in training and continuing professional development to develop general and specialist testing skills and competence.

The Future Is Before Us

Fundamental changes are emerging that will have significant impact on our society in the coming years, and in some cases have already changed it. The response capacity of machines will continue to increase sharply. They will exceed human possibilities many times over. Machines will process data faster, be more mobile and more flexible. Robot axes are already moving autonomously in Phoenix, USA; in Heidelberg, Germany, a trained artificial intelligence system recognizes skin cancer on photographs better than most doctors with average levels of experience.

Information is being processed by ever more powerful devices. The results are becoming ever more complex. How can today's massive data volumes be monitored and controlled by humans? Or, to put it another way, how can a human test a robot that is more 'intelligent' than the tester him/herself?

The need for continuing professional development will continue to increase.

Every day we receive new information on our smartphones about our changing world. About the jobs that are disappearing and about others that are replacing them. About transparent data that offers practical advantages to us in a thousand everyday things. At the same time we hear (and see) policy-makers debating these very things.

New Ethics

Artificial intelligence will not take away the necessity for us to think. On the contrary, certainly we will make use of artificial intelligence to control the ever-more complex functions of machines. We must nevertheless concentrate on what we can do as humans—and on what we can pick up and learn to do. For all the questions about artificial intelligence, about digitization and automation, the human will remain the decisive factor.

After 15 years of working intensively in the IT training sector, iSQI has shown with complete clarity that without specialists, automation is nothing. And artificial intelligence is nothing without people who specify to it what is 'intelligent'. The more complex the systems become, the more important will be training and also

ongoing professional development. We do not need to compete with machines! Rather, we need to control them. How we do this is above all a question of attitude of mind, of ethics. What is it that we want, what is the focus of our desires, how should we act? We should follow commonly agreed ethical principles. We need a code of conduct for the future and our work in service to society. The good of humanity is our first objective, and not the development of machines. All products, software and devices are subject to the highest possible quality demands, always with regard to their benefits. We undertake to maintain fair relations with our colleagues, we take pride in the image of our profession and strive to continue to learn for the entire time that we are in the world of work.

***The* formula for the future:**

Digitization is qualification plus automation—upskilling leads digitalization.

One matter is for sure. The more accurately one may assess the developments, the better we can prepare ourselves for tomorrow. For us, the workers, developers, testers, marketers, employers and employees of the future, this means one thing above all: education. Being able to prepare ourselves for the new tasks. The Canadian creators of the careers2030.cst.org website, for example, list the jobs that will be in demand in 2030 with defined job descriptions. This makes for exciting reading, while allowing some to look into the future with relief. There will not be fewer jobs then than now. There will be designers then, though they may be called recyclable design specialists; there will be farmers, though their profession will be based on microclimates and the question of how quickly solar energy and water reach the plant. Their job title might then be 'agroecologist'. Marketers could become e-media makers and arts managers for big data-driven video, text and game content. And IT experts? They will develop ever more into communicators that enable the different computers to communicate with each other. Software testers will have a much stronger presence in the project planning stage. They will concentrate more on human-machine communication. Things won't be simpler. But the opportunities will outweigh the risks.

This book, issued to mark the 15th anniversary of iSQI, is intended to make a further contribution to raising the profile of the profession of software testing. How is the profession of tester changing? What must a tester prepare for in the coming years, and what skills will the next generation of software testers need? What opportunities are available for further training today? What will testing look like in an agile world that is user-centred and fast-moving? What tasks will remain to the tester once the most important processes are performed automatically? These are questions that we will have to answer for ourselves.

CEO iSQI Group, Potsdam, Germany Stephan Goericke

Contents

Editor and Contributors

About the Editor

Stephan Goericke has been the managing director of the International Software Quality Institute iSQI GmbH (iSQI) since 2005. Furthermore, he is the CEO of the ASQF (Arbeitskreis für Software-Qualität und -Fortbildung e.V). Under his leadership, iSQI expanded worldwide, became the "most world widely known certifier" and now has branches and offices on every continent in the world.

Stephan has worked for many years with professionals, trainers and institutions in the field of education and is a certifier standing in the midst of digital transformation. He is convinced that international standards and upskilling are one of the most important factors for success. He defines this as a formula for the digital world: "Digitalization is Qualification plus Automation—Upskilling leads Digitalization."

Stephan is a sought-after keynote speaker on the subjects of standardization, certification and software quality. Furthermore, he is the editor of the "SQ Magazine" and the international publication "SQmag"—journals for software development and quality management, aimed at experts from the field of software development, quality assurance and management.

Contributors

Sven Amann is a consultant of CQSE GmbH for software quality. He studied computer science at the Technische Universität Darmstadt (Germany) and the Pontifícia Universidade Católica do Rio de Janeiro (Brazil). He received his PhD in software technology from Technische Universität Darmstadt.

Graham Bath is a principal consultant at T-Systems in the division of Digital Integration and Agile Testing and benefits from over 30 years of testing experience to support customers with consultancy, training and test process improvements. Graham is the ISTQB Working Group chair for the Specialist Level Certified Tester qualifications and co-authored the new syllabus on Usability Testing. Graham is also a member of the German Testing Board and is a frequent presenter and tutorial lecturer at conferences around the world. He co-authored the book "The Software Test Engineer's Handbook".

Frank Faber is a DevOps and Test consultant at Alten Nederland. Via Alten, he has been seconded to several companies where he worked in Agile and DevOps teams. With them, Frank has experienced the transition to DevOps. He helps his teams understand what DevOps is, and how you work Agile and DevOps in practice. In addition to his consultancy work, Frank provides training and lectures on DevOps and on testing in an Agile environment. And in his role as an ambassador for the DevOps Agile Skills Association (DASA), Frank wants to bring the fields of DevOps and Testing together, to help discover in which ways these two fields can learn and benefit from each other.

Paul Gerrard is a consultant, teacher, author, webmaster, programmer, tester, conference speaker, rowing coach and publisher. He has conducted consulting assignments in all aspects of software testing and quality assurance, specializing in test assurance. He has presented keynote talks and tutorials at testing conferences across Europe, the USA, Australia, South Africa and occasionally won awards for them.

Educated at the universities of Oxford and Imperial College London, he is a Principal of Gerrard Consulting Limited, the host of the Assurance Leadership Forum and a business coach for Enterprising Macclesfield. He was the Programme Chair for the 2014 EuroSTAR conference in Dublin and for several other conferences in the last decade. In 2010 he won the EuroSTAR Testing Excellence Award, in 2013 the inaugural TESTA Lifetime Achievement Award and in 2018 the ISTQB Testing Excellence Award. He is currently working with an Irish government agency to create a future skills framework for software testers.

Elmar Juergens is founder of CQSE GmbH and consultant for software quality. He studied computer science at the Technische Universität München and Universidad Carlos III de Madrid and received a PhD in software engineering.

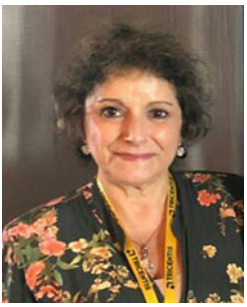
Tilo Linz is board member and co-founder of imbus AG, a leading solution provider for software testing, and has been active in the field of software quality and software testing for more than 25 years. As founder and chairman of the German Testing Board and founding member of ASQF and ISTQB, he has played a major role in shaping and advancing education and training in this field on a national and international level. Tilo is author of “Testing in Scrum” and co-author of “Software Testing Foundations”.

Rik Marselis is a testing expert at Sogeti in the Netherlands. He is a well-appreciated presenter, trainer, author, consultant and coach, who supported many organizations and people in improving their testing practice by providing useful tools and checklists, practical support and in-depth discussions. He is also an accredited trainer for ISTQB, TMap- and TPI-certification training courses, and has additionally created and delivered many bespoke workshops and training courses, e.g. on Intelligent Machines and DevOps testing.

Rik is a fellow of Sogeti's R&D network SogetiLabs. The R&D activities result in white-papers, articles and blogs about IT in general and testing in particular. He has contributed to 21 books in the period from 1998 until today. In 2018, his last book as main-author *Testing in the Digital Age: AI Makes the Difference* was published.

Mitko Mitev With more than 25 years of experience in the software quality assurance and more than 20 years' experience as Project and Test Manager, Mitko Mitev is one of the leading software test experts in South East Europe. Mitko is a part of the International Software Testing Qualifications Board (ISTQB) and President of the South East European Testing board (SEETB), and also the Chief Editor of the Quality Matters Magazine, distributed internationally with a focus on the best leading practices and new trends in the area. His commitment to the promotion of the Software Quality is also demonstrated by taking the role of the Chair of the SEETEST Conferences and participation as a keynote speaker in many software quality assurance conferences.

In the past few years he has focused mainly on writing course materials, books and articles that will enable everybody to learn more about the field of software testing. Along with that, he is the owner and CEO of an outsourcing and consultancy company in Bulgaria—Quality House—the market leader in offering highly professional testing services.



Alfonsina Morgavi is partner at QActions System SRL. For more than 20 years she has been exclusively dedicated to software quality assurance and quality control.

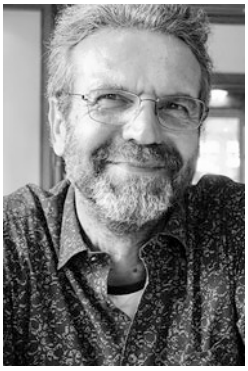
She is the representative of Argentina in HASTQB (Hispanic America Software Qualification Board), a frequent speaker in national and international events and has written several articles about related topics. She actively promotes the professionalization of the activity, ISTQB certifications, and the use of structured techniques and effective tools to support the achievement of maturity in functional, automated, performance and security tests. During her more than 20 years of practice she has trained many testing teams not only in Argentina but also in other countries in the region, and created and organized a considerable number of productive testing teams, for continuous improvement of their technical skills.

Zornitsa Nikolova is a co-founder and managing partner at Leanify Ltd.—a company focused on Lean and Agile training, mentoring, and coaching. Zornitsa has extensive experience in software product development both in large enterprises and startups. Currently, she works as a trainer and a coach with companies and individuals who are looking to apply agile approaches to product development. In addition to her consulting practice, Zornitsa engages with startups as a co-founder and mentor. She also teaches classes in agile software development and product management at the New Bulgarian University (NBU) and the VUZF University in Bulgaria. Before her career as a trainer and coach, she worked at SAP as a development manager and product owner. Zornitsa holds an MA in International Relations, EMBA from Cotrugli Business School, and she is a Certified Scrum ProductOwner (with Scrum Alliance), a Professional Scrum ProductOwner (with Scrum.org), and an Associated Certified Coach (with the International Coach Federation).

Gerard Numan has been a tester since 1998. He has international experience in various contexts and roles and is an experienced test trainer. He has published a book on E2E-testing and has spoken at conferences around the world on subjects such as E2E-testing, test process improvement, risks, critical thinking for testers, the sociology of testing and testing AI. Gerard works for Polteq Test Services B.V. in the Netherlands.

Ina Schieferdecker is Director General of the Department on Research for Digitalization and Innovation of Federal Ministry of Education and Research. Before, she has been the Director of Fraunhofer FOKUS, Berlin and Professor for Quality Engineering of Open Distributed Systems at the Technische Universität Berlin. Her research interests include urban data platforms, critical infrastructures and conformity, interoperability, security, reliability and certification of software-based systems. She is President of the Association for Software Quality and Education (ASQF), member of the German National Academy of Science and Engineering (acatech), and of Münchner Kreis e.V. She has been member of the German Advisory Council on Global Change (WBGU) as well as steering committee member of the Science Platform Sustainability 2030. She was also Director of the Weizenbaum Institute for the Networked Society, the German Internet Institute in Berlin and member of the "Hightech Forum 2025" of the Federal Ministry of Education and Research.

Adam Leon Smith is a specialist in software quality and emerging technologies. He has held senior technology roles at multinational companies including Barclays and Deutsche Bank, and delivered large complex transformation projects. Adam is a regular speaker on multiple topics, including testing and artificial intelligence technology. He manages an AI product development team that has developed an AI product to support the management of development and testing activities, as well as specializing in testing AI-based systems. He is also actively working with IEEE, ISO, and BSI on standards development in the AI and quality domain.



Andreas Spillner has been working in the field of software development and testing in practice and research for 40 years. He studied computer science at the Technical University Berlin and received his doctorate at the University of Bremen. Until 2017 he was Professor of Computer Science at the University of Applied Sciences Bremen. He teaches software engineering with a focus on quality assurance and programming.

Andreas is a Fellow of the "Gesellschaft für Informatik e.V.", a founding member of the "German Testing Board e.V." (honorary member since 2010)

and was founder and spokesman of the special interest group "Test, Analysis and Verification of Software". He is also a member of the advisory board of the ASQF ("Arbeitskreis Software-Qualität und -Fortbildung e.V."). He is author or co-author of several German-language books (e.g. *Basiswissen Softwaretest*, *Lean Testing für C++-Programmierer—Angemessen statt aufwendig testen*) and magazines. In addition, he has authored numerous publications in journals and presentations at international and national conferences as well as seminars.

Mark Tannian is a consulting associate and professional development provider for RBCS Inc., an adjunct faculty member of St. John's University in New York City and Executive Director of Education for a New York City-based security professionals organization (i.e. $(ISC)^2$ NY Metro Chapter). In 2018, he developed the A4Q Design Thinking Foundation course and certification. He pursues innovative approaches to business and security challenges through his consulting, research and teaching.

Mark holds professional certifications in information security (i.e. CISSP) and project management (i.e. PMP). In 2013, he received his PhD in computer engineering from Iowa State University, USA.

Ignacio Trejos-Zelaya is an Associate Professor of the Tecnológico de Costa Rica (TEC) since 1984. He is also a Co-Founder & Professor at the University Cenfotec & Cenfotec since the year 2000. He has directed more than 35 MSc theses and published over 40 technical articles, as well as over 240 opinion articles.

Ignacio has been a speaker on nearly 200 conferences, panel sessions and technical talks. He is a professional member of the Costa Rican Association of Computing Professionals; IEEE Computer Society; Association for Computing Machinery; American Society for Quality, Software Division and the Hispanic America Software Testing Qualifications Board.

Kaspar van Dam With approximately 15 years of experience in IT, Kaspar van Dam advises colleagues and clients on matters concerning testing and/or collaboration and communication within (agile) teams, projects and organizations. He has published a number of articles on test automation, agile ways of work and continuous communication. He also trains and coaches people on these subjects and has been a speaker on related items at events.

Hans van Loenhoud graduated as a biologist and worked in ecological research at the University of Amsterdam. In 1980 he switched to IT as a Cobol programmer. Later, he specialized in consultancy on information and quality management. Around Y2K Hans entered the field of software testing. For many years, he was a board member of TestNet, the Dutch association of professional software testers. As a tester, he took interest in requirements engineering, because he is convinced that good requirements are a prerequisite for professional testing. He is committed to build bridges between these disciplines, as a writer, speaker and trainer, and is a lecturer on related topics at the Amsterdam University of Applied Sciences. Hans is second chair of IREB and active there in the Foundation Level and the Advanced Level Elicitation working groups.

Rini van Solingen is a speaker, author, professor, and entrepreneur. Each year he gives over 100 lectures and workshops. His expertise lies in the speed and agility of people and organizations. Rini makes complex matters simple and can explain things in understandable and humorous ways. His strongest focus is on empowering his audience. He helps people see why things are going the way they are and what they can do about it themselves. In addition, Rini is a part-time full professor at Delft University of Technology and also regularly gives lectures at Nyenrode Business University in master classes and MBAs. He is also CTO at Prowareness We-On, where, as a strategic consultant, he helps clients render their organizations fast and agile. Rini is the author of a number of management books, including The Power of Scrum (2011—with Jeff Sutherland and Eelco Rustenburg), Scrum for Managers (2015—with Rob van Lanen)

and the management novel: The Beeshepperd—How to Lead Self-Managing Teams (2016). He can be reached via his website: www.rinivansolingen.com

Erik van Veenendahl (www.erikvanveenendaal.nl) is a leading international consultant and trainer, and a recognized expert in the area of software testing and requirements engineering. He is the author of a number of books and papers within the profession, one of the core developers of the TMap testing methodology and the TMMi test improvement model, and currently the CEO of the TMMi Foundation. Erik is a frequent keynote and tutorial speaker at international testing and quality conferences, e.g. at SEETEST. For his major contribution to the field of testing, Erik received the European Testing Excellence Award (2007) and the ISTQB International Testing Excellence Award (2015). You can follow Erik on Twitter via @ErikvVeenendaal

Keith Yorkston After a circuitous route into IT, Keith has been involved with risk and testing for over 20 years, specializing in non-functional testing. Currently working for Expleo Group as a consultant and trainer, he runs the Expleo UK Academy. He very much believes in the overall view of security surrounding technology, process and people, as many of today's attacks involve all three areas.

Change-Driven Testing

Sven Amann and Elmar Jürgens

Abstract Today, testers have to test ever larger amounts of software in ever smaller periods of time. This makes it infeasible to simply execute entire test suites for every change. Also it has become impractical—if it ever was—to manually ensure that the tests cover all changes. In response to this challenge, we propose Change-Driven Testing. Change-Driven Testing uses Test-Impact Analysis to automatically find the relevant tests for any given code change and sort them in a way that increases the chance of catching mistakes early on. This makes testing more efficient, catching over 90% of mistakes in only 2% testing time. Furthermore, Change-Driven Testing uses Test-Gap Analysis to automatically identify test gaps, i.e., code changes that lack tests. This enables us to make conscious decisions about where to direct our limited testing resource to improve our testing effectiveness and notifies us about where we are missing regression tests.

Keywords Software testing · Test automation · Test intelligence · Regression-test selection · Test prioritization · Test-resource management · Risk management

1 A Vicious Circle

Today, testers have to test ever larger amounts of software in ever smaller periods of time. This is not only because software systems grow ever larger and more complex, but also because development processes changed fundamentally. Ten years ago, software was commonly released at most once or twice a year and only after an extensive test period of the overall system. Today, we see consecutive feature-driven releases within a few months, weeks, or even days. To enable this, development happens on parallel feature branches, and testing, consequently, needs to happen on each such branch as well as on the overall system.

S. Amann · E. Jürgens
CQSE GmbH, München, Germany

S. Goericke (ed.), *The Future of Software Quality Assurance*,
https://doi.org/10.1007/978-3-030-29509-7_1

In response to this fast-growing demand for testing, companies invest in test automation and continuous integration (CI) to speed up test execution. However, we increasingly see that even automated test suites run for multiple hours or even days, especially on larger systems. As a result, such long-running test suites are typically excluded from CI and executed only nightly, on weekends, or even less frequent. As a result, the time between the introduction of a mistake in the code and its detection grows. This has severe consequences:

- Large amounts of changes pile up between two consecutive test runs, such that it becomes difficult to identify which particular change contains the mistake.
- Developers get feedback from tests only long after they did the changes that introduced the mistake, again making it more difficult for them to identify the mistake.
- The effects of multiple mistakes may overlap, such that one mistake may only be detected after another is fixed.

To make things worse, test automation addresses only half the problem: While it improves the efficiency of test execution, it does nothing to ensure that the testing is effective. In practice, we see that about half of the changes may escape even rigorous testing processes [1, 2]. And as testers strive to make test suites more comprehensive by adding additional tests, they also add to the runtime of the suites, thus, jeopardizing the benefits of automated tests and CI.

So how can we break this vicious circle and make our testing processes efficient and effective at the same time? The answer is surprisingly simple: *We align our testing efforts with the changes.* With an increasing number of changes to be tested, it has become infeasible to simply execute all tests after every change and it has become impractical—if it ever was—to manually ensure that all changes are tested adequately. But instead of resolving to test only rarely, we should keep testing frequently with focus on the changes, i.e., the code that might contain new mistakes. We call the idea of aligning our testing with the changes in application code *Change-Driven Testing*. Change-Driven Testing identifies 90% of the mistakes that our entire test suite may find in only 2% of the suite's runtime [3] and informs us about any change to the code that we did not test [1, 2].

2 Test Intelligence

To achieve high-quality testing, we commonly need to answer questions such as which test we need to run, what else we need to test, or whether our test suite contains redundant tests. Since it is difficult to correctly answer such questions manually, our goal is to automatically answer them using existing data from the software development process. This approach is similar to the approach of Business Intelligence, which analyzes readily available data to gain insights on business processes. Therefore, we refer to our approach as *Test Intelligence*. Figure 1 illustrates it.

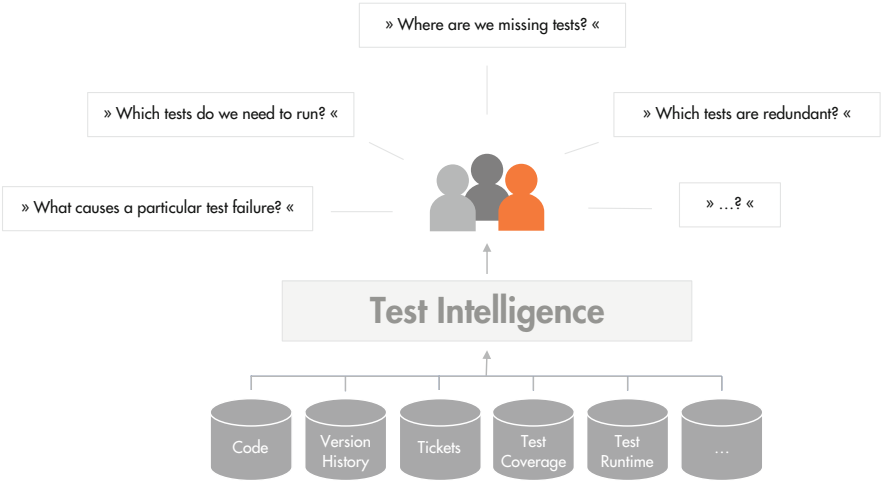


Fig. 1 Test intelligence: combining readily available data from various data sources in the software development process to automatically answer questions about testing

The questions we want to answer through Test Intelligence concern both our tests and the code changes under test. Therefore, we collect data about both the tests and the changes. Much of this data is already available in the development environment and need only be extracted for our purposes.

To communicate the extracted data to testers, developers, and managers alike, we use *treemaps*. Figure 2a shows such a treemap that represents the code of a UI component of the software system `Teamscale`. Each box on the map represents a single method in the code. The size of the box is proportional to the size of that method in lines of code. We color the boxes to highlight particular properties of the respective code.

2.1 Version-Control Systems

Version-control systems (VCS), such as `git` or `TFS`, are a de facto standard in today's software development. Generally speaking, a VCS keeps a chronological history of code changes committed by developers and helps them coordinate their changes. Within the VCS, changes may be organized in branches, where a *branch* is an isolated line of changes that is based on a particular version of the code and may be merged into a later version of the code. If developers use a dedicated branch for the implementation of a particular feature, we call it a *feature branch*.

From the change history in a version-control system, we may extract the list of changes since any given baseline, be it a particular release, the last test run, the start of a feature branch, or any other point in time. Figure 2b shows code changes on a treemap. Methods that were added by one of these changes are highlighted in

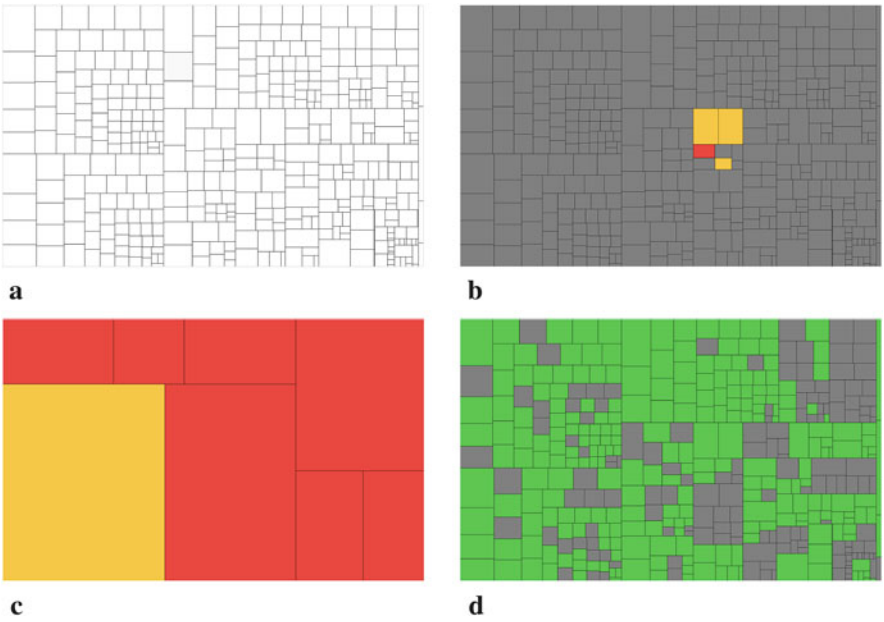


Fig. 2 Treemaps that show data about the software system `Teamscale` from the version-control system, the ticket system, and profiling test execution. (**a**) Source code as a Treemap. Each box represents one method. The size of the box is proportional to the lines of code in that method. (**b**) Code changes since a fixed baseline. New methods are red, changed methods are yellow, and methods that remain unchanged are gray. (**c**) Code changes for ticket `TS-15717`. New methods are red and changed methods are yellow. (**d**) Test coverage on method-level. Methods that have been executed are green, all other methods are gray

red, methods that were changed are highlighted in yellow, and methods that remain unchanged are highlighted in gray.

2.2 Ticket Systems

Ticket systems, such as `Jira` or `GitHub Issues`, are used in most software projects to keep track of change requests, such as bug reports, feature requests, or maintenance tasks. Such systems allow the development team to manage each request as a ticket, which usually has a unique ID, an assignee, and a status, among other metadata.

It is a widely used practice that developers annotate changes in the version-control system with the ID of the ticket(s) that motivated the changes, usually by adding the ID to the commit message that describes the respective change. Using these annotations of the code changes and the metadata from the ticket system, we

can group all changes that belong to the same ticket. This allows us to focus on only the changes motivated by a particular ticket.

Figure 2c shows the changes related to a single ticket on a treemap. Methods that were added by a change are highlighted in red and methods that were changed are highlighted in yellow.

2.3 Profilers

Profilers, such as `Jacoco` or `gcov`, record which parts of the application code are executed, i.e., they record *code coverage*. Depending on the technology in use, profiling approaches range from instrumenting the target code, over attaching a profiler to a virtual machine, to using hardware profilers. Regardless, profiling is always possible.

Different profilers record coverage at different granularity, e.g., they record which methods, statements, or branches get executed. With most profilers, a finer granularity results in a larger performance penalty. Recording coverage on method level is, in our experience, always feasible with an acceptable overhead.

When a profiler records the coverage produced by a test, we speak of *test coverage*. Recording test coverage is a widely used practice for automated tests in CI and uncommon in other types of test environments. Technically, however, we may profile any type of execution, be it through a unit test or an end-to-end test, automated or manual. Thus, using a profiler, we may obtain the coverage of each test in our entire test suite.

Existing profilers typically aggregate the coverage of all tests executed in a single test run, because they work independently of the test controller and simply trace execution from start to end of the run. However, conceptually, we may also record *test-wise coverage*, i.e., separate coverage for each test case. When we record test-wise coverage, we may also trivially record each test's individual runtime.

Figure 2d shows the aggregated coverage of a test suite on a treemap. Methods that were executed are highlighted in green and methods that were not executed are highlighted in gray.

3 Change-Driven Testing

Change-Driven Testing is one particular instance of Test Intelligence that makes testing both more efficient and effective. Figure 3 depicts the idea. The key insight behind Change-Driven Testing is that existing tests will only fail if new mistakes are introduced,[1] and that new mistakes can only be introduced through changes. Consequently, when considering the (possibly buggy) changes between

[1] Leaving aside causes for sporadic test failures, such as flaky tests or interaction with third-party systems, which typically indicate a problem with the test setup rather than with the system under test.

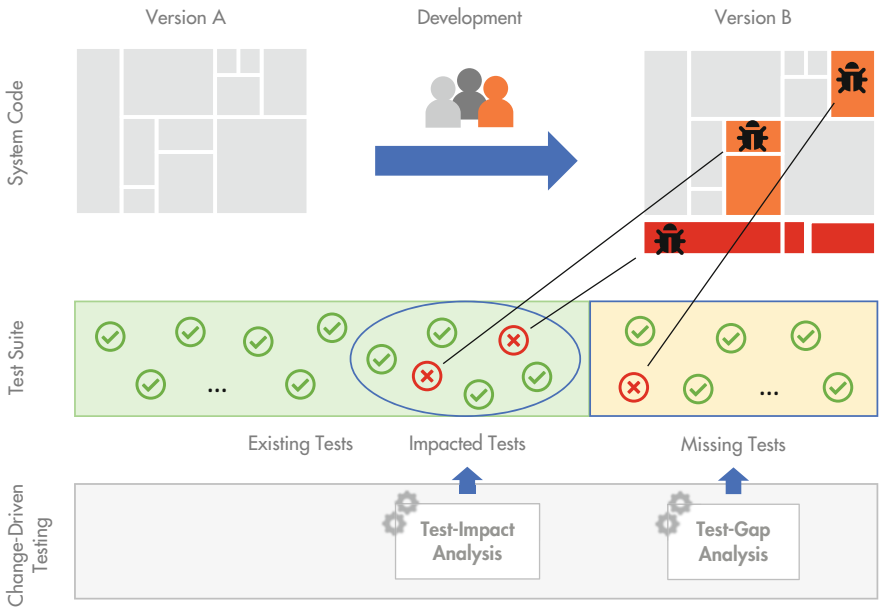


Fig. 3 Change-driven testing: testing efficiently by running only the tests impacted by changes. Testing effectively by testing all changes

two versions of our system, e.g., two consecutive releases of the system before and after implementing a particular feature, we proceed in two phases:

1. To make our testing more *efficient*, we test just the changes, excluding all other parts of the system. We automatically identify the existing tests that are relevant to the changes, i.e., the impacted tests, through a Test-Impact Analysis.
2. To make our testing more *effective*, we ensure that we test all changes. To guide us towards this goal and to verify whether we achieved it, we automatically identify changes that lack tests through a Test-Gap Analysis.

Both analyses are only interested in changes that require testing. Therefore, we use static code analyses to identify *relevant changes*. We consider a change relevant if it modifies the *behavior* of the code and, thus, may contain mistakes that later cause errors. Consequently, we filter out changes that correspond to refactorings, such as changes to documentation and renaming or moving of methods or variables.

3.1 Test-Impact Analysis

Figure 4 depicts the process of the Test-Impact Analysis (TIA). It combines relevant code changes (see Sect. 2.1) with test runtimes and test-wise coverage (see Sect. 2.3)

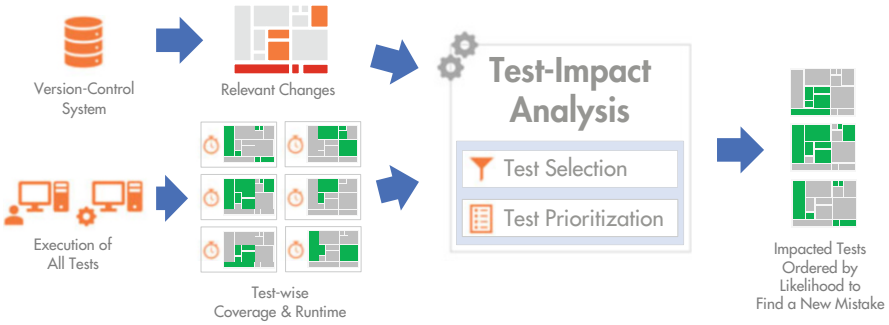


Fig. 4 Process of the Test-Impact Analysis (TIA). Given a set of changes, TIA selects the impacted tests and orders them by their likelihood to find a new mistake in the changes

to compute a subset of the entire test suite that identifies as many mistakes as early as possible. The process consists of two steps:

1. In the *Test Selection* step, TIA selects the *impacted tests*, i.e., all tests that execute any changed method, according to the recorded coverage. It also includes all tests that were added or modified by the change, because it cannot know which parts of the code those tests cover.
2. In the *Test Prioritization* step, TIA orders the impacted tests such that all changes are covered as quickly as possible, to find new mistakes as early as possible. Since computing the optimal ordering of the tests is infeasible, TIA uses a greedy heuristic: It selects that test next, which covers the most additional changed code per execution time.

In a study with twelve software systems [3] we found that the impacted tests selected by TIA find 99.3% of all randomly inserted mistakes that the entire test suite could find. The impacted tests found more than 90% of those mistakes in all study systems and even 100% in seven of them.

In a second study with over 100 different systems [4] we found that TIA identifies on average over 90% of the mistakes that the entire test suite identifies in only 2% of the execution time of the entire suite. Using TIA is especially beneficial for small and local changes, where test selection results in a small set of tests. This perfectly suits our need to quickly test many incoming changes, which are usually small compared to the code of the entire system.

3.2 Test-Gap Analysis

Figure 5 depicts the process of the Test-Gap Analysis (TGA). It matches relevant code changes (see Sect. 2.1) with the aggregated test coverage (see Sect. 2.3) to identify those changes that were not covered by any test. We call these changes *test gaps*.

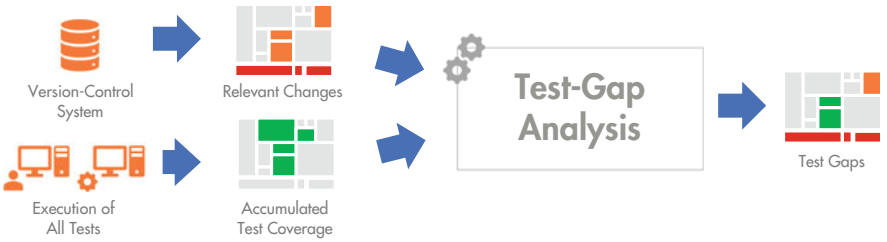


Fig. 5 Process of the Test-Gap Analysis (TGA). From a given set of changes TGA identifies those changes that were not yet tested, i.e., test gaps

To determine test gaps, TGA considers the chronological order of changes and test runs: New changes invalidate any previous test coverage of the changed code and open new test gaps. With subsequent testing, new coverage is recorded, which closes respective test gaps on earlier changes. Consequently, TGA can give us an update on our test gaps after every test run and every code change. And since the analysis works incrementally, it computes the update in a matter of seconds, even for very large systems.

In a case study on a large industrial software system [1] TGA revealed that more than 55% of the code changes in two consecutive releases remained untested. In retrospect, we traced over 70% of the reported field bugs back to untested code.

In a second case study on another industrial software system [5] TGA found 110 test gaps in the changes from 54 tickets, which corresponds to 21% of the 511 change methods. Provided with the data, developers found 35% of these gaps worth testing. Interestingly, they also found that 49% of the gaps resulted from cleanup work that was not part of the ticket description and, thus, remained untested even after the change requested by the ticket was thoroughly tested.

3.3 Limitations

To make best use of the analyses, it is important to be aware of their limitations. One limitation of both TIA and TGA is changes on the configuration or data level. Since such changes are not reflected in the code, they remain hidden from the analyses. Consequently, TIA cannot adequately select impacted test and TGA cannot show impacted parts of the code as untested.

A related limitation of TIA comes from the use of indirect calls. For example, if a test executes all classes with a certain annotation, TIA typically does not select this test when the annotation is added to another class, because the test's historical impact did not include that class and the test itself did not change on the code level.

TIA assumes that the coverage of test cases is stable, i.e., that executing the same test twice results in the same coverage. If this is not the case, e.g., because manual test steps are performed differently with every execution, TIA cannot properly

capture the impact of the test and, therefore, cannot adequately select impacted tests. In practice, even the coverage of automated tests may vary between runs, e.g., due to garbage collection. Though the effects are typically small, this means that TIA cannot guarantee that the impacted tests always find all the mistakes that the entire test suite may find. To the best of our knowledge, there is no test-selection strategy that gives such a guarantee under these conditions. Therefore, we recommend to regularly execute the entire test suite to ensure that nothing slipped through and to keep the data about test runtimes and coverage up to date. This is the same as using a traditional regular (e.g., nightly) testing strategy, except that we now have the additional fast feedback from the immediate testing using TIA, which already captures the majority of mistakes.

Another limitation of TGA is that it does not consider how thoroughly a change was tested, but only whether the methods containing the change were at all executed in a test. Like any other testing approach, TGA cannot prove the absence of mistakes in the code. However, by revealing changes that have not been tested at all, it identifies changes for which we are certain that no mistakes can have been found. Such untested changes are five times more likely to contain mistakes [2]. In our experience, TGA usually reveals substantial test gaps and, thereby, enables significant improvements of the testing process.

4 Using Change-Driven Testing

To illustrate the use of Change-Driven Testing, we follow the development of a feature of our own software product `Teamscale`. `Teamscale` is a platform that assists software development teams in the analysis, monitoring, and optimization of code and test quality. For this purpose, it integrates with various other development tools, such as version-control systems, build servers, and ticket systems. Its web frontend provides an administration perspective that allows to manage credentials for such external systems. Figure 6 shows feature request `TS-15717` that asked to add the possibility to delete such external credentials.

Before starting to work on `TS-15717`, our developer creates a feature branch to work on. Then she implements the relatively small feature, writes a new test, and commits the changes as

> `TS-15717`: Support deletion of external credentials.

The treemap in Fig. 2c shows all of these code changes.

4.1 Testing with TIA

Triggered by the commit, our CI environment starts building `Teamscale` and testing the change. In the testing stage, the CI queries TIA to learn which tests to execute. To compute its response, TIA uses the test-wise runtime and coverage of

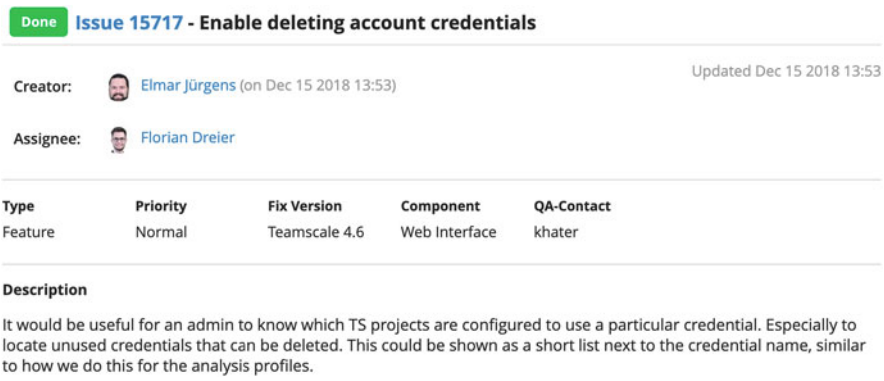


Fig. 6 Feature request `TS-15717`: enable deleting account credentials

each of the roughly 6.5k tests (including unit tests, integration tests, system tests, and UI tests) in our test suite. To obtain this data, we augmented the CI environment with profilers that record test-wise coverage in both the JavaScript code of the frontend and the Java code of the backend and ran the entire test suite once, which took about 45 min. Based on this data, TIA now determines a list of six tests impacted by the changes: Five regression tests covering parts of the changed code and the new test that came with the changes. The entire CI run with these impacted tests takes about 1.5 min, saving us over 96% runtime.

Thanks to TIA, only 1.5 min after committing her changes, our developer learns that the new test (ranked second by TIA) fails. With her changes still fresh in her mind, she investigates the problem and fixes it in about 10 min, committing the new changes as

> `TS-15717`: Fix deletion of external credentials.

Since the fix is very local, TIA selects only a single impacted test, namely, the test that previously failed. Therefore, the second CI run takes only about 45 s in total. This time all tests pass.

Overall, two consecutive CI runs using TIA plus correcting the mistake in between took less time than one execution of our full test suite. Figure 7 illustrates this improvement.

4.2 Testing with TGA

To ensure that all her changes are properly tested, our developer next looks at the test-gap treemap for her changes in the context of `TS-15717`. Figure 8 shows how TGA displays the test coverage recorded in the two previous CI runs. The treemap shows what we call the *ticket coverage* of `TS-15717`: most of the code changes

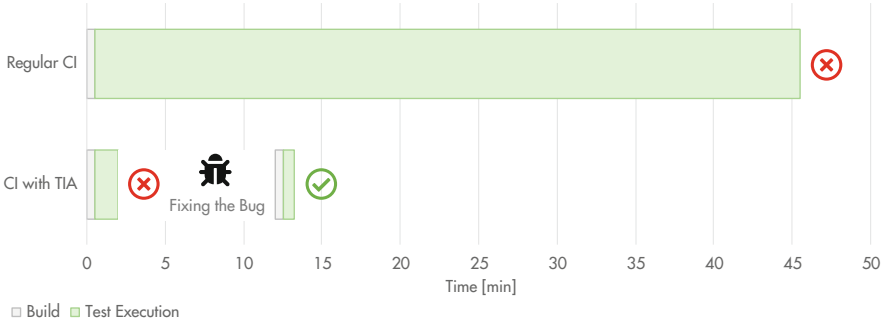


Fig. 7 The gain from using test-impact analysis. Two consecutive CI runs using TIA plus correcting a mistake take less time than one CI run with `Teamscale`'s full test suite

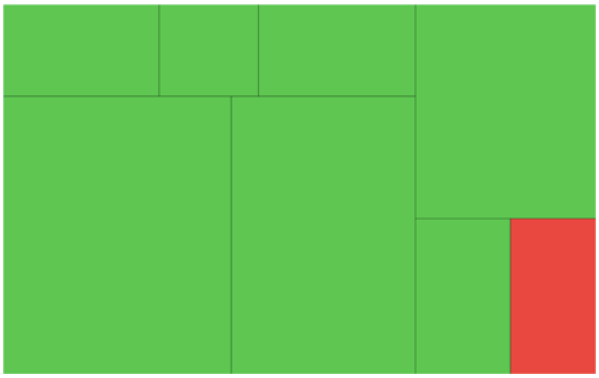


Fig. 8 Ticket coverage for `TS-15717` after executing the impacted tests identified by TIA. TGA reveals one remaining test gap

were tested (the green rectangles), but one test gap remains (the red rectangle in the lower right).

To decide whether the remaining test gap is worth closing, our developer drills down from the treemap into the code. She discovers that the untested code is responsible for requesting an additional confirmation, when a user attempts to delete credentials that are still used by `Teamscale`. Since this code is relatively simple and unlikely to ever change, she decides to test it once manually.

Our developer starts the development version of `Teamscale` from her local machine, using a manual-test startup script that we maintain with our code. She opens the system under test in a browser, navigates to the administrative perspective, and checks whether the additional confirmation is indeed requested. It is, and so she shuts down the system under test, which causes the startup script to automatically provide the recorded coverage for TGA. As a result, all changes for `TS-15717` were now tested, verifiably documented by an all-green test-gap treemap.

Since our developer opted for a manual exploratory test, there is no regression test for this particular functionality. However, since TGA is aware of the chronological order of changes and test coverage, it will again report a test gap should the functionality ever change in the future. Thanks to this safety net, it is reasonable to opt for a quick manual check instead of writing an automated UI test or a manual test for code that is unlikely to ever change again.

4.3 Closing the Loop

At this point, our developer is satisfied with her changes and sends them to one of her peers for code review. Once she and the reviewer agree that the changes are fine, he merges the feature branch. In response, our CI environment runs our entire test suite. This ensures that the main product line is error free, even if TIA should have mistakenly excluded a relevant test, and also records coverage and test execution times to keep our data up to date. Note that the vast majority of CI runs still benefits from TIA, since merging feature branches happens much less frequently than committing changes to feature branches.

Before each `Teamscale` release (as of this writing, every 6 weeks) a test architect inspects all remaining test gaps on changes since the last release across the entire system. This provides us with a second quality gate, to ensure that no critical functionality accidentally slipped through testing. In this process, the architect uses the same data that was used in the development process of the features, but on a treemap that represents the entire code instead of only the code changes for an individual feature. Figure 9 shows a section of this global test-gap treemap, representing one of `Teamscale`'s UI components.

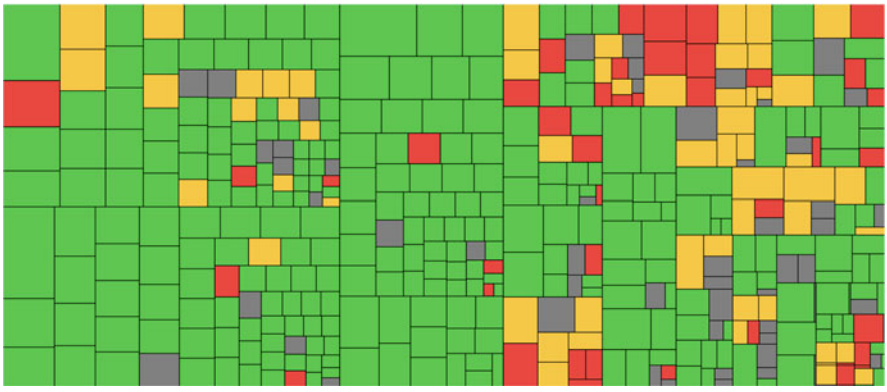


Fig. 9 Test-gap Treemap for a UI component of `Teamscale`. A global analysis of remaining test gaps serves as an additional quality gate before a release

5 Adapting Change-Driven Testing

In practice, we encounter very different testing setups and strategies, depending on the concrete requirements and the history of the respective projects. Consequently, the implementation of Change-Driven Testing should be adjusted to deliver the most value given the project's parameters. We subsequently discuss some aspects that we encounter repeatedly, without aiming for an exhaustive list.

- Both Test-Impact Analysis and Test-Gap Analysis may consider test coverage from all testing stages as well as from a combination of both automated and manual tests. The type of test literally makes no difference for the benefits of Test-Gap Analysis. Test-Impact Analysis, on the other hand, is especially beneficial if tests are time consuming, i.e., if excluding tests saves significant time, and if testing time is limited, because test prioritization makes it likely that we catch mistakes early, even if we cannot afford to run all impacted tests.
- If development mostly happens on feature branches, a sensible strategy is to use Test-Impact Analysis for testing changes and ticket coverage to avoid test gaps on these branches. In addition, to ensure no mistakes slip through testing, the full test suite should be executed upon merge of a feature branch. If, on the other hand, development happens on a main branch, we may use Test-Impact Analysis to test individual changes and run the full test suite periodically or before a release.
- It is always possible to combine Test-Impact Analysis with other test-selection strategies, e.g., if some tests for highly critical functionality should always run. We then simply run the union of the impacted tests selected by TIA and the tests identified by any complementary strategy.
- We found that it is most efficient to investigate test gaps using ticket coverage, because the ticket provides us with additional context when analyzing the gaps. However, even if TGA cannot map code changes to tickets, it can reveal test gaps for the system as a whole. Such a system-wide TGA is valuable on its own as much as in addition to ticket coverage, as a second-level quality gate.
- In many projects, the people analyzing test gaps are not necessarily the developers who wrote the code changes, but testers or architects. Such a separation of work sometimes makes the interpretation of test gaps more difficult, because the analysts may be unaware of possible reasons for a particular change or test gap. In such cases, the data from the version-control system again proves helpful, because it names the developer responsible for any change, telling the analysts who to talk to.
- While it is the theoretical ideal, it is never a goal in itself to reach an all-green treemap, i.e., 100% test coverage. In many situations, leaving test gaps is quite reasonable, e.g., if the gap is on code that prepares future functionality, but that it not yet live or if it is on code that is only rarely used internally, such that testing resources are better invested elsewhere. In the end, the testing process is always subject to tradeoffs and prioritization. TGA enables us to make conscious decisions about where to direct our limited resources.

6 Conclusion

Today, testers have to test ever larger amounts of software in ever smaller periods of time. This makes it infeasible to simply execute even fully automated test suites in their entirety for every change. Also it has become impractical—if it ever was—to manually ensure that the tests cover all changes. Therefore, we need to rethink our testing strategies to become both more efficient and effective.

In this chapter, we introduced Change-Driven Testing. In Change-Driven Testing, we analyze existing data from the software development process to automatically answer questions that drive our testing. We use Test-Impact Analysis to automatically find the impacted tests for any given code change and sort them in a way that increases the chance of catching mistakes early on. This makes testing more efficient, catching over 90% of mistakes in only 2% testing time. We use Test-Gap Analysis to automatically identify test gaps, i.e., code changes that lack testing. This enables us to make conscious decisions about where to direct our limited testing resource to improve our testing effectiveness.

References

1. Eder, S., Hauptmann, B., Junker, M., Juergens, E., Vaas, R., Prommer, K.H.: Did we test our changes? Assessing alignment between tests and development in practice. In: Proceedings of the Eighth International Workshop on Automation of Software Test (AST'13) (2013)
2. Juergens, E., Pagano, D.: Did We Test the Right Thing? Experiences with Test Gap Analysis in Practice. Whitepaper, CQSE GmbH (2016)
3. Juergens, E., Pagano, D., Goeb, A.: Test Impact Analysis: Detecting Errors Early Despite Large, Long-Running Test Suites. Whitepaper, CQSE GmbH (2018)
4. Rott, J.: Empirische Untersuchung der Effektivität von Testpriorisierungsverfahren in der Praxis. Master's thesis, Technische Universität München (2019)
5. Rott, J., Niedermayr, R., Juergens, E., Pagano, D.: Ticket coverage: putting test coverage into context. In: Proceedings of the 8th Workshop on Emerging Trends in Software Metrics (WETSoM'17) (2017)

The Next Generation Tester: Meeting the Challenges of a Changing IT World

Graham Bath

Abstract If we stop and look back at events that are shaping IT, the pace of change is amazing. Things are so different now, even if we just consider the last 5 years. Projects using, for example, artificial intelligence or Internet of Things are no longer "over the horizon"; they are reality.

Testing needs to keep moving with the times even more than in the past. It needs to face the challenges posed by a changing IT landscape and continue to add value. To do that testers must make sure they are providing the right competencies to projects, regardless of the software development lifecycle used or the particular type of project: It's time to shape what the "Next Generation (NG)" Tester should be.

Keywords Software testing · Software quality · Software tester · Software testing skills · Software testing compentencies · Artificial intelligence

1 Introduction

If we stop and look back at events that are shaping IT, the pace of change is amazing. Things are so different now, even if we just consider the last 5 years. Projects using, for example, artificial intelligence or Internet of Things are no longer "over the horizon"; they are reality.

Testing needs to keep moving with the times even more than in the past. It needs to face the challenges posed by a changing IT landscape and continue to add value. To do that testers must make sure they are providing the right competencies to projects, regardless of the software development lifecycle used or the particular type of project: It's time to shape what the "Next Generation (NG)" Tester should be.

G. Bath
T-System, Frankfurt, Germany

ISTQB, Edinburg, UK

S. Goericke (ed.), *The Future of Software Quality Assurance*,
https://doi.org/10.1007/978-3-030-29509-7_2

For two decades the software testing industry has established the tester as a role. Organizations such as the International Software Testing Qualifications Board (ISTQB) and the International Software Quality Institute (iSQI) have done much to establish this role and have also helped to define more specific roles such as Test Manager and Test Analyst. Those roles are now well established, which is a credit to those organizations, and a great improvement on the "dark" days of the past where being a tester was not always considered to be a particularly attractive and challenging career to pursue. But does this role-based approach still serve the IT industry well?

Looking at the title of this article, you might be forgiven for thinking that the Next Generation (NG) Tester is just another attempt to define a new role in testing. In fact, the NG Tester is more a conceptual idea than a new role. The NG Tester may be considered as someone with a particular set of testing competencies that suits an organization or project. The word "competency" is of particular relevance here. A competency:

- Defines something that a tester can do (e.g. design effective test cases for AI applications)
- Is something that a tester can deliver in real life projects
- Can be evaluated within the context of a skills framework

This chapter first considers some of the key factors that influence businesses today and how the IT industry has responded. Based on those factors, the chapter considers some of the principal testing competencies required of an NG Tester. These are described within the conceptual model of a professional development skills framework which provides both a structure for organizing testing competencies and an approach to defining and evaluating them. Using the skills framework, organizations and projects will be able to flexibly define what competencies "their NG Testers" need.

2 Forces That Drive Businesses

Fundamentally, the following three forces influence business today:

1. A continuous drive for cost-effectiveness
2. The need to deliver new products and services quickly, flexibly, and frequently
3. The ability to adapt (i.e. agility)

Quality? Yes, but achieving quality *must* be in balance with these key forces.

3 Trends in IT

3.1 Solutions in IT

In response to the forces facing business, several new solutions in IT have emerged, all of which impact the competencies required for testing. These include:

- Highly connected applications
  - Applications using the concept of the Internet of Things (IoT)
  - Mobile applications
- Use of Artificial Intelligence (AI)
- Use of Big Data (BD)

The list is, of course, not exhaustive. There are many other IT trends that could also be listed, but those mentioned are considered to have the most impact on software testing in general.

3.2 Practices in IT

The way in which software projects are conducted continues to be influenced by the following trends:

- A more collaborative way of working between project stakeholders
  - Agile
  - DevOps
- Less focus on specific testing roles
- Increased levels of automation
  - Continuous integration (CI)
  - Integrated Test Automation Frameworks (TAF)
  - Modelling for test automation (MBT)
- More integration into projects of particular engineering disciplines
  - Design engineering for IoT
  - Usability engineering

4 What Competencies Will Be Needed in Testing?

Considering the trends in IT mentioned above, testing professionals will clearly need a wide range of testing competencies if they are to be considered valuable members of project teams. Providing an answer to the high-level question of “what

competencies" is therefore best addressed by using a framework model which enables the different competencies to be organized. Initially, the well-known T-Model will be used, but this will later be extend to a more comprehensive and usable form.

The T-Model, as the name suggests, organizes competencies into two categories:

1. Broad-based, general competencies which may be applied to a wide range of projects. These competencies, such as being able to communicate effectively to a wide range of stakeholders, are represented by the horizontal part of the T. Testers should try to achieve all of these competencies.
2. Specialized competencies, such as being able to set up a test environment for mobile application testing, are represented by the vertical part of the T. Testers may acquire any number of these competencies as required.

A selection of typical competencies from each of these categories is given below.

5 General Competencies

Some of the more important general competencies are suggested below. It would be expected that different people might want to put other competencies on this list. Defining the NG Tester using a skills framework allows for this flexibility. Having said that, it would be unusual to not find the following competencies within a particular NG Tester definition.

5.1 Testing Essentials

The fundamental skills of a tester will always be required, whatever the project or context. These are the skills which enable testers to find defects more effectively and efficiently than those without those skills.

Competencies required of the NG Tester:

- Design tests using the "traditional" black-box techniques. This includes the ability to design tests using equivalence partitioning, boundary value analysis, decision tables, state transition diagrams and combinatorial techniques.
- Perform effective exploratory testing.
- Take on various tasks in the testing process (e.g. analysis of user stories, design and execution of appropriate tests).

5.2 Risk-Based Testing

As an established and well-understood approach, applying risk-based testing is an essential general competency.

Competencies required of the NG Tester.

- Identify and categorize different types of risks.
- Prioritize testing according to risk levels.

5.3 System Architectures

The architecture of a system has an impact on how it is tested.

Competencies required of the NG Tester:

- Understand and analyse a specific type of system architecture (e.g. web-based, micro-services, Model-View-Controller, client-server).
- Discuss architectures with system designers to identify risk areas and target testing.

Remember, the competencies here may be focused on one or more particular types of architecture which a given project applies. It is uncommon to require competencies in all possible architectures.

5.4 Communication

Effective communication within projects is essential to ensure correct understanding and give stakeholders the required information.

Competencies required of the NG Tester:

- Communicate effectively in teams.
- Talk to stakeholders at their level.
- Judge how much communication is enough.

5.5 Automation

Competencies in automating tests are no longer considered a speciality; they are an essential skill, especially where agile and DevOps approaches are used in a project.

Competencies required of the NG Tester:

- Achieve efficiencies by using test automation tools.
- Program test automation scripts which are maintainable and efficient.

5.6 Adapting to Different Lifecycle Approaches

A tester must have competencies in working with different lifecycle models, including sequential (e.g. V-Model) and iterative (e.g. agile).

Competencies required of the NG Tester:

- Place testing activities within the particular lifecycle model.
- Apply communication and automation competencies as required by the lifecycle model.

Summary of the General Competences
The NG Tester can:

- Practice the essentials of testing in order to detect defects early
- Apply risk-based testing to prioritize testing
- Understand different system architectures in order to establish testing needs
- Communicate effectively in teams
- Communicate with stakeholders at their level
- Judge how much communication is enough
- Achieve efficiencies by using test automation tools
- Program test automation scripts which are maintainable and efficient
- Place testing activities within the particular lifecycle model
- Apply communication and automation competencies as required by the lifecycle model

6 Specialist Competencies

A wide spectrum of different specialist competencies may be acquired by the professional tester. In this section, we consider the competencies needed for testing applications which use Big Data, connected devices and artificial intelligence. These three subjects illustrate the typical challenges facing the tester and the competencies they require to deal with them (further specialist subjects are covered, for example, by the ISTQB in their Specialist stream of certification syllabi www.istqb.org).

6.1 Testing Big Data (BD) Applications

Big Data applications enable organizations to extract value from the vast amount of data provided by sensors, monitors and various other sources of business information.

Challenges for testing

- A wide range of functional and non-functional quality attributes must be considered, such as:
  - Performance of data analysis to provide business with "instant" results
  - Scalability of applications to enable potentially huge volumes of data to be acquired and integrated into the existing system data
  - Functional correctness of analysis methods and algorithms applied to the data
- Data currency, backup and recovery capabilities must be tested to ensure the high quality of data. In many cases this data may represent a "single point of truth" for the entire business.

Competencies required of the NG Tester:

- Provide stakeholders with an assessment of product risks which consider the key quality attributes of BD applications.
- Agree testing goals with a wide variety of business stakeholders.
- Conduct exploratory testing sessions with business stakeholders to evaluate the value and accuracy of results provided by the Big Data application.

6.2 Testing Applications Using Connected Devices

Applications which use connected devices fall into two broad categories:

- Apps which run on hand-held mobile devices such as a smart phone
- Applications built on the concept of the "Internet of Things". In this case, the connected device might be a car, a refrigerator or any other form of device which can be wirelessly connected via the internet

Challenges for Testing

The wide range of quality characteristics to be considered presents the main challenge for testing. In addition to functionality, these include the following non-functional characteristics and associated answers to be provided by testing:

- Usability:
  - Can the user effectively and efficiently use the application?
  - Is there a positive experience for the user?
  - Can users with disabilities also access the applications?
- Performance
  - Does the system respond to user requests in a timely manner?
  - Can the system deal with the loads imposed by many users?

- Security
  - Are communications protected from security threats (e.g. man in the middle attacks)?
  - Can a mobile device be misused by unauthorized users (e.g. distributed denial of service attacks)?
- Interoperability
  - Can we be sure our application runs on the devices and versions intended?
  - Can the required different operating systems, browsers and platforms be supported?
  - Can we easily upgrade or swap different hardware and software components?
- Reliability
  - Can the system handle a wide range of different exception conditions, such as loss of connection, intermittent signals and hardware or software failures in particular system components?

Just consider the testing effort that would be required to test all of these individual characteristics and provide answers to all those questions. Although it might be desirable to cover everything, this is rarely a practical proposition in terms of available budget and timescales. The challenge for the tester therefore comes in helping to decide how to prioritize testing.

Competencies required of the NG Tester:

- Identify the principal risks associated with applications and systems using connected devices.
- Determine a test strategy to mitigate a wide range of technical risks.
- Design and perform tests which focus on the particular non-functional quality characteristics of applications and systems which use connected devices
- Apply appropriate tools (e.g. simulators and emulators) to testing applications which use connected devices.

6.3 Testing Artificial Intelligence (AI) Applications

Testing AI applications focusses principally on the ability of the application's algorithms to match ("fit") requests for information with the application's available knowledge base. Testing focusses on two main aspects:

- Detection of situations where the application cannot match information requests at all and therefore provides no response (underfitting). This is an indication that the matching algorithm may be too precise.

- Detection of situations where too many matches are found and potentially incorrect responses are provided to requests (overfitting). This is an indication that matching algorithms are too general and provide unwanted responses.

Challenges for testing

- Any test needs expected results to enable detection of defects. Because AI applications are continuously "learning" and updating their knowledge bases, the definition of expected results is difficult.
- Testers must involve business users in the evaluation of results and in any decision to raise a defect report.

Competencies required of the NG Tester:

- Make stakeholders aware of the key risks of AI applications.
- Coordinate and test specific aspects of AI applications in cooperation with business stakeholders.
- Understand AI application architectures and the specific purpose of the application.
- Set up different sets of data in order to identify issues with underfitting and overfitting.

7 The Skills Framework

So far we have considered the general competencies to be provided by the "next generation" tester and a selection of the specialist competencies. Whereas ideally all of the general competencies should be available, testers, employers and projects need to decide on which specialist competencies are required and where skills development should take place. Given the wide range of possible testing competencies, these decisions are best supported by a skills framework which goes well beyond the simple T-Model. A skills framework captures the following elements:

- Competency descriptions
- Competency levels
- A method for assessing competencies
- Profiles

7.1 Competency Descriptions

Competency descriptions are provided of general and specialist competencies. At high level, these would be similar to those outlined above, but may be further broken down as required.

7.2 Competency Levels

Each competency in the skills framework is assigned one of the following suggested levels:

- Initial level: At this level a person can demonstrate a particular competency in a project if supported by others with higher competency levels.
- Full level: At this level a person can fully demonstrate the competency in a project without support
- Consulting level: At this level, a person is sufficiently competent to provide consultation services to people, projects and organizations on the particular subject. The person can take the initiative in identifying ways of improving the competencies of others.

7.3 Assessing Competencies

A skills framework must describe the conditions for achieving individual competencies at different levels. Given the three levels described above, the following conditions may be defined for their assessment.

Initial level

- Proof of understanding is required: Evidence for this may be provided by a certification provided by internationally recognized authorities such as iSQI and ISTQB.

Full level

- Proof of practical experience over a minimum period (e.g. 18 months) is required. A project portfolio is submitted and a reasonably complex predetermined practical task completed as evidence of achieving the competency. An independent evaluation of the portfolio and the practical task is performed.

Consulting level

- The criteria to be achieved are similar to those described for the full level, except that the length of the experience is longer, the portfolio is more extensive and the predetermined practical task more complex.

Note that the individual levels mentioned above should not be formally bound to each other by pre-conditions. For example, if a person can show evidence of achieving competencies at consulting level, they should not first need to be assessed at initial and full levels.

7.4 Profiles

A key element of a skills framework is its ability to be easily and flexibly applied to the practical needs of people, projects and organizations. Profiles provide a way to define collections of specific competencies at particular levels which match particular demands. For example:

- An organization which creates mobile applications which make use of AI may define a profile which includes all general testing competencies to initial level plus specialist competencies at full level in testing applications using connected mobile devices and AI.
- An individual may construct a profile which enables them to define a goal for their personal development. They may, for example, set the goal of being competent at full level in certain general competencies and competent to initial level in a particular speciality.

Note that profiles are similar to but are not the same as roles. Profiles are more flexible in their definition and relate more directly to a specific need. They are constructed from the competencies provided in the skills framework and do not suffer from the inflexible "one size fits all" problem which is inherent in traditional role definitions.

8 Conclusion

Role-based approaches in testing have shown their value and are still in use today. However, projects (in particular those using agile approaches) are increasingly moving away from fixed roles and towards a more flexible competency-based approach. Adopting this type of approach enables projects to define their precise competency needs and to align better with the needs of business.

A skills framework supports the overall concept of a competency-based approach. They provide the skills definitions that enable people, projects and organizations to set up their own flexible competency profiles. By including levels of competencies and a scheme for assessing them, the skills framework becomes a flexible tool with which to measure skill levels and promote their development.

Testing in DevOps

Frank Faber

Abstract DevOps can be explained by a set of principles and these principles indicate that Testing in DevOps should be looked at differently. The customer must be at the center of testing and more specialized tests must also be used. To ensure that there is continuous feedback, automation should be used as much as possible. All these tests are the responsibility of one team. To achieve this, a test engineer in DevOps must become a T-shaped engineer.

Keywords Software testing · Software quality · Test automation · Test engineer · DevOps · Agile testing

1 Introduction

In many IT organizations there is a "Wall of Confusion" [1] between development and operations. On the one hand, development wants change, but on the other, operations wants stability. Between them is a wall with a lot of confusion.

There has been a shift to break down this "Wall of Confusion." More teams are starting to work with a DevOps mindset. Engineers from different organizational silos are working together to create value for the customer. The DevOps mindset places more emphasis on delivering quality in a service, with end-to-end responsibility.

To assure the quality of the service, testing was and will be an essential part of the process. With DevOps, changes are necessary to make sure testing really focuses on the quality in the delivered service.

F. Faber
ALTEN Nederland, Eindhoven, The Netherlands

S. Goericke (ed.), *The Future of Software Quality Assurance*,
https://doi.org/10.1007/978-3-030-29509-7_3

2 What Is DevOps?

The term DevOps consist of nothing more than the contraction of the words: *Development* and *Operations.* DevOps started in 2009 with the DevOps Days, a conference organized by Patrick Dubois [2]. Although many have tried to come up with a definition for DevOps, there is not one universally accepted definition. The DevOps Agile Skills Association (DASA) has formulated six principles which cover most definitions of DevOps.

1. Customer-Centric Action
2. Create with the End in Mind
3. End-to-End Responsibility
4. Cross-Functional Autonomous Teams
5. Continuous Improvement
6. Automate Everything You Can [3]

The six principles help understand what DevOps really is about.

2.1 Customer-Centric Action

DevOps is usually mentioned together with Agile because they share some core ideas. The Agile Manifesto [4] shows a way of working in which a small incremental piece of software adds value for the customer. DevOps builds on the same principles by putting the customer in the center of the action. DevOps teams work directly for the customer and should base all their activities on the mindset that they should add value for the customer.

To be able to accomplish this, it is inevitable to have feedback from the customer. There should be a short feedback loop where the customer can influence the actions of the team. Teams can experiment with new features for services. With a properly implemented feedback loop, customers can give quick feedback on the usage of these features.

2.2 Create with the End in Mind

The scope of work within DevOps teams has changed. Before DevOps, the wall of confusion between Development and Operations prevented engineers to work with the scope of the entire Software Development Life Cycle (SDLC). Development created software and ceased after tests were successfully executed. Operations picked the software up for deployment on production with some knowledge on what happened before they got their hands on the software. With DevOps, Development and Operations expertise should be present in a team. A DevOps team should invest in a process where they own the entire SDLC of a functional service.

2.3 End-to-End Responsibility

Responsibility can give teams positive satisfaction in the work they do. It is one of the motivators for work as described by Frederick Herzberg [5]. As part of the two-factor theory, Herzberg distinguishes motivators and hygiene factors for job satisfaction.

Within DevOps, there is an End-to-End responsibility for the service(s) a team delivers. That implies that teams are accountable from "concept to grave" [3]. Teams are responsible for the entire SDLC and they should be aware and act on all changes the service undergoes. Security and performance, for example, could become a higher priority for the team, because they are responsible for maintaining the service. Security, performance, and all other quality attributes are not the responsibility for specialists anymore but should be built in the SDLC process from the start by the entire DevOps teams.

This responsibility can motivate teams and lead to positive satisfaction.

2.4 Cross-Functional Autonomous Teams

In DevOps, teams should be cross-functional and autonomous. All the skills and knowledge to deliver and maintain a service with End-to-End responsibility should be available within the team. This doesn't mean that engineers should be superhumans and should be able to perform every task available. The skills and knowledge of an engineer can be linked to the T-shaped model [6]. The T-shaped model consists of two bars where the vertical bar stands for the depth of knowledge and skills in a specific area. The horizontal bar shows the knowledge and skills in many areas where the engineer is no expert. This should make it possible for an engineer to pick up simple tasks in their team from outside their expertise. Sharing within the team is important because it will allow engineers to grow in the horizontal bar of the T-shaped model due to the knowledge and skills of their team members. This however is only possible with the vertical bar of the T-shape from other team members. There should be expertise available in depth to be able to share this with other team members.

2.5 Continuous Improvement

New customer demand and changes in the environment are reflected in the work a team needs to do. Teams need to be continuously focused on improvements in their service. This is implemented by the "The Second Way" from The Phoenix Project [7]. "The Second Way" is about implementing a constant flow of feedback. As a team you want to have as much feedback as possible, as soon as possible. Customer collaboration helps with the feedback of the customer. Testing and monitoring is

another way of giving teams feedback on the way their services operate. Teams should be able to react quickly on failed tests or changes in performance as shown in monitoring.

2.6 Automate Everything You Can

Many teams, including DevOps teams experience that they have to perform the same task multiple times in succession. To enable flow in the SDLC, many teams turn to automation. Automation allows them to remove error prone manual steps. When well implemented, it offers a faster process and can free up time for the team to focus on delivering quality in their service.

In the current market there are many companies that use the term DevOps to promote their tools for automation. Although tools can help a team transition to DevOps, DevOps won't start with a tool but with a change in the process.

3 Testing in DevOps

"The process consisting of all lifecycle activities, both static and dynamic, concerned with planning, preparation and evaluation of software products and related work products to determine that they satisfy specified requirements, to demonstrate that they are fit for purpose and to detect defects." [8]

This definition of software testing is known throughout the world. In DevOps these activities still need to be undertaken to verify quality of a product. Before DevOps, testing tended to be a phase in the SDLC which ended before the product was delivered to Operations. Testing, however, should not be a phase. It should be present from the start and should be a responsibility shared by the entire (delivery) team [9]. Done right, testing helps to "Build quality in" as how W. Edwards Deming described this [10]. The shared responsibility together with embedding it in the entire process should lead to a good quality product.

3.1 Scope

The scope change that comes with DevOps means that teams have an end-to-end responsibility over the quality of their service. Activities that assure the quality of their service are part of this end-to-end responsibility. You could easily argue that all test activities should be done in a DevOps team. The tests a team performs should be in theory end-to-end testcases, because it should cover the entire scope of the teams' service. This can be difficult when a team is responsible for a service that connects with many other services. The definition of end-to-end for a team can differ from the

definition a customer has. Business processes can cover multiple services, and end-to-end testing will test the services of multiple teams. The responsibility of these end-to-end tests can cause a problem because they can be owned by multiple teams.

One way to deal with this problem is to use contract-based testing [11]. DevOps lets you look at software as a service and it is common to create a contract for a service. Contract-based testing is verifying if the service upholds that contract. The tests should imitate a customer (or another service) and make sure there is no contract breach. If a company consists of multiple services and all teams have successfully implemented contract-based testing, you could even argue that there is no need for end-to-end testing anymore. The functionality should be covered in all contract and therefore in the tests. This, however, is only possible when the contract matches with the customer's needs and no mistakes are made in the chain of contracts. There should be something to cover any discrepancies between contracts. End-to-end testing could be used to mitigate the risk in these discrepancies. In a microservices architecture [12] contract-based testing is a more applicable test strategy, because the architecture consists of multiple services. In a monolithic architecture it can be more difficult to apply contract-base testing, because it is more difficult to identify services that can be individually tested.

End-to-end testing that cover multiple teams could be made from small tests connected to each other in a framework until they cover the entire business process. Teams will be responsible for a small part of the tests and should offer help to other teams to make sure the test chain will work. With this approach it is still possible to perform end-to-end testing with DevOps teams. It requires a lot of coordination and a testing framework that supports the input of multiple teams.

In certain situations, it can and will be more fruitful to make one team responsible for end-to-end testing when it covers multiple services. This team can deliver an end-to-end testing service for multiple teams. DevOps teams should provide criteria for the tests for the parts that cover their service and should act on negative results from the tests. The end-to-end testing team should be responsible for making sure the tests will be executed, but the DevOps teams are still responsible for their own service in these tests. Whatever you choose, it is best to ensure that a team is created with the principle of customer-centric action first.

3.2 Customer-Centric Testing

The Agile Manifesto was the start of changing software development. It focusses highly on the process. In his book *Specification by Example*, Gojko Adzic writes: "building the product right and building the right product are two different things" [13]. He makes the distinction between the process and the outcome of the process. While your process can be as you need it to be, there is no guarantee that the outcome of your process – your product/service – has the right quality. To achieve quality, a direct focus on the customer is needed, because they determine what the requirements of your service should be. In this context it makes sense that the first

DASA DevOps principle is about Customer-Centric Action [3]. Testing could help focus on the customer when you apply customer-centric testing.

The first step in customer-centric testing is collecting the requirements, because they will lead to test cases. *Specification by example* [13] is a term which describes a way of dealing with requirements. Another term which shares the same set of principle and ideas is Behavior-Driven Development [13, p. xiii]. Adzic writes that specifying should be done collaboratively to be most effective. In Agile, the term "The Three Amigos" started, where a business analyst, a developer, and a tester worked together on the requirements of a new feature [14]. This process reduced complexity and tried to get the broadest view on requirements with a small group. In DevOps a team draws up requirements in a collaborative manner, whereby all the team's expertise is considered. It takes it a step further from The Three Amigos when you focus on the expertise the different engineers bring in a team. These kinds of expertise can shape the quality of your service.

Specification by example starts with the customer perspective when writing requirements. One format for doing that is the Given-When-Then format, also known as Gherkin language [15, 16]. This way of writing down requirements in terms of behavior is a way of putting the customer in the center. It's also a format which is much easier to understand by a customer, either internal or external. Using multiple of these statements will give a set of examples which identifies the functionality a customer wants. It enables discussion between the customer and the DevOps team on a level both can relate to. Examples are a way to illustrate the behavior of a customer and these examples take requirements a step toward a more customer-centric approach.

From the Given-When-Then format to the actual tests is a small step to take with multiple test tools. Tools like Cucumber [17] make it possible for engineers to make test cases with this format. In these tools, Given, When, and Then act as keywords and can be linked to test code which is used to run the tests. Cucumber and other tools keep the requirements visible, so they can be part of ongoing discussions on the requirements and test cases. In test execution reporting, the same format can be used to share the results of tests to stakeholders. Teams must consider the management of the tests and glue code in Cucumber, because it may result in a lot of extra work if it is not manageable.

DevOps does not demand teams to implement *Specification by Example* or use Cucumber as their primary test tooling. Both the method and tool are merely examples of another way of testing where the approach is to look directly at the customer when creating requirements and test cases. It can help create a mindset for Customer-Centric Testing.

3.3 Specialized Testing

More specialized tests like security and performance testing should also be done within a team. This can potentially pose a problem, because what do you do when

the knowledge of this kind of testing is not present in the team? There are multiple approaches to face this problem depending on the situation in the team.

As mentioned previously, DevOps doesn't mean engineers should be superhumans. Engineers will probably lack knowledge and skills in certain areas. In general engineers with a testing expertise will know a little of security and performance testing, because they are considered to be specialized fields. Being a T-shaped engineer in DevOps, however, is also about challenging an engineer to expand his knowledge and skills. It can be possible for an engineer to become potent in both security and performance testing. If you also consider that all the knowledge of the service the team provides is within the team, team members should be able to assist the engineers in this. Operations expertise could help understand the performance on server level. Development expertise could help understand framework and methods of security in code. Engineers should not necessarily become experts in security and performance testing, but they should be able to create tests which properly test the requirements set for their service.

Another approach can be to use a testing service for performance and security. End-to-end responsibility does not mean teams have to do everything themselves, but they should keep control. In bigger enterprise organizations it is more common that teams deliver a security or performance testing service. What should be considered though is that working with these testing services can prove to be difficult to fit in the way of working of the teams. There can be different priorities between the teams, which could result in not getting the testing service at the moment you need to. It should also be clear what the expectations are on both sides of the testing service. DevOps teams should deliver a clear request on what they want to have tested and should provide the necessary knowledge. On the other side it should be clear for DevOps teams what is expected from the tests.

Between these two approaches some hybrid approaches are possible. Having an engineer with security/performance testing knowledge in the team for a short period of time or having security/performance testing teams educate the DevOps teams to the required knowledge level are just two examples.

With each approach it is very important for DevOps teams to understand that the scope of their testing has changed. They are responsible and should keep control of all kinds of testing their service needs.

4 Automation

DevOps will not work without automation. Error-prone manual tasks can and should be replaced by automation. DevOps teams require fast feedback and automation is the way to get this to the team. It can speed up the existing processes and make sure the team receives feedback about the process as soon as possible. When automation is working, team members can focus on tasks which do require human intervention.

Automation can play a role in the breakdown of the "Wall of Confusion." It could be possible that Development and Operations used their own set of tools for

deploying and other processes. Within DevOps it is best if teams start using the same tools for the entire SDLC. This can be a way of bringing team members together and make the SDLC clear and coherent. Different skills present in the team can shape the automation to where it fits the needs of the entire team.

4.1 Test Automation

In testing, more and more tests are being automated. Test engineers work more with testing tools and automation supporting their tests. It creates fast feedback loops which drives development design and release [9]. In DevOps you want to automate as much as possible. The testing quadrant diagram, as created by Brian Marick, was adopted by Lisa Crispin and Janet Gregory to create the Agile Testing Quadrants [18]. The quadrants show different kinds of tests where automation can play a role. Technology facing tests supporting the team, like unit tests, are automated. Business facing tests supporting the team, like functional tests, can be automated or done manual. These functional tests in DevOps should be automated, based on the DevOps principle "Automate everything you can" [3]. These functional tests are the tests that should be part of customer-centric testing as mentioned before. Technology facing tests critique to the project, however, require mostly tools and are therefore already automated.

The automation of the first three Agile Testing Quadrants should leave time and space for the last quadrant with business facing tests that critique to the product. These tests should be done manual and cannot be automated. With multiple skillsets in a DevOps team, it would benefit the team to perform these tests with the team during the time they saved with implemented automation.

The test pyramid can help the implementation of test automation. It makes a distinction between tests that can be executed fast on low levels (unit tests) and tests that are slower to execute (UI tests) [19]. The lower-level tests are most suitable for automation, which is why they are usually fast to execute. The test pyramid combines tests originally performed by developers (unit tests) and those performed by test engineers (service, UI tests). This is a testing strategy that will work with DevOps because it is a cross-functional strategy. Engineers in a DevOps team should share their knowledge and expertise to fully implement this test strategy. This strategy also helps teams making testing a shared responsibility within the team.

4.2 Continuous Testing

Automated testing can be part of a deployment pipeline and can be part of Continuous Integration, Delivery, or even Deployment. Deployment pipelines are the "automated implementation of your application's build, deploy, test and release process" [9]. The deployment pipelines are a way to empower teams to take control

over their deliverables. A pipeline can help a team deploy their service and verify the quality in an automated way. In DevOps teams, it should empower all team members to deploy any version on any environment with the right controls in place. The pipeline can limit the complexity of deploying and testing a service. The knowledge difference between team members can be smaller when every team member could deploy and test a service with a push of a button.

Continuous testing can act as continuous process where every time the pipeline is started tests are being executed. Tests can act as go or no-go points in the pipeline to go to the next step in the process. It also gives the team up-to-date feedback on the quality of their service in different stages of development. Testing can also be used to test the automation. It can help understand if the automation executes the correct steps in a correct way. This includes the automation used for deploying services on different environments. With deployment testing, a team can take control of the infrastructure and check whether it is in the state where it should be. Testing will give teams control on their automation when they are relying much more on it.

4.3 Monitoring

It is increasingly common to arrange monitoring from the start of software development. With an end-to-end responsibility in DevOps teams, monitoring is a way to get feedback from the service on production environments to the teams. Monitoring can vary from technical monitoring on server level: measuring CPU usage, memory, etc., to more functional monitoring: how many users are logged in etc. Functional monitoring gives to a certain degree insight into customer perception of the service. It can allow teams to track customers through their usage of the service.

It could be argued that monitoring can replace parts of testing. If a DevOps team has reached "the third way" [7], they gather feedback continuously and experiment with their service. Monitoring can help the team in the gathering feedback. The third way is when teams are mature enough to deliver new features fast and experiment to see what fits the needs of the customer.

Monitoring should be a more reactive way of getting feedback where testing is a more proactive approach. That is mainly because monitoring is focused on production environments and therefore a later step in the process. Testing can start at an early point in the process and gives teams more options to adapt to the outcome of testing. Monitoring can act as a way of testing when teams are able to adapt quick to the outcome of monitoring. If teams can create new features and deploy them fast, using, for instance, Continuous Deployment, teams can react fast. Monitoring can act as business facing tests critique of the product and would fit in the Agile testing quadrant [18].

5 The Role of a Test Engineer

A test engineer was and is someone who can act as intermediate between business and IT. A test engineer can connect the requirements from the business side to the technical implementation of IT side. This can be an enabler for an ongoing conversation between Business and IT. With DevOps, this conversation must keep going to make sure the customer is in the center of the action. In DevOps it is more likely to have engineers with different skillsets present in a team. The function of a conversation enabler is required. A test engineer can bridge the gap between Dev and Ops, because quality is what connects these two. Test engineers can play a vital role in this conversation. They can facilitate that quality is a shared responsibility in the team. Quality can be the most important topic that brings teammates closer together. Engineers can give their view on quality based on their own expertise. A test engineer can make sure that all these different views are combined in a single test strategy.

Test engineers in DevOps are not the only one responsible for preparing and executing test cases. T-shaped engineers with different kinds of expertise should also be able to prepare and execute test cases. Test engineers can act as coaches to their team members and help them understand how to test. The test expertise from a test engineer should be shared within the team.

5.1 T-Shaped, Test Shaped

A T-shaped test engineer should start with making sure his or her own expertise fits in DevOps. The test expertise should contain knowledge and skills to gather the correct requirements for the system under test. From these requirements test cases can be created and executed using test techniques. The test expertise should also contain knowledge on how to choose and use the correct tooling in the process. This expertise differs not from the existing role of test engineer outside DevOps.

With all the automation present in DevOps, a test engineer needs to have technical skills and knowledge to implement testing in Automation. From Test Automation to Continuous Delivery, a test engineer must be able to fit in where the team has implemented automation. This usually means that a test engineer needs to have some basic coding skills and understand the structure of most programming languages. Next to technical skills a test engineer must understand Test Automation and be able to implement it in a fitting test strategy. Following the T-shaped model you could say that programming knowledge is part of the horizontal bar and need not be an in-depth knowledge and skill. Test automation and complementary test techniques should be part of the vertical bar and the Test expertise of an engineer.

Test engineers can act as intermediaries between Business and IT or between different kinds of expertise in a DevOps team. This enables them to gain knowledge from different parties present in the team. They should be able to gain knowledge, and this can help them expand the horizontal bar of their T-shape.

5.2 Soft Skills

Although the focus now seems to be on the technical skills that a test engineer needs, it is also important that a test engineer has good soft skills. Soft skills are needed to get the conversation between different parties going and to keep it going. If a test engineer is going to be an intermediary, then people skills must be in order. These skills make it possible to take a leading role.

6 Conclusions

The practice of testing in DevOps starts with the same foundations as testing in a non-DevOps environment would have. The end-to-end responsibility that a team has for a service means that quality requirements for both development and operations must be considered. Quality in the Operations section of the SDLC was usually not included but is now part of the scope of testing. The scope for testing in DevOps is on the entire functional service a team delivers. End-to-end testing therefore can take a different form. More specialized tests, like performance and security tests, will part of the new scope, although on a smaller scale. Due to the specialized nature of these tests, however, it is possible that these tests will be executed as part of a testing service outside the team. DevOps teams should take ownership of these tests and should make sure they are executed. Functional tests in DevOps should focus more on the customer as part of customer-centric testing. This makes sure the quality a customer wants is put central in the work a team performs.

In DevOps you want to automate everything you can. Automation in testing must be used where it can be used. With automation teams can get continuous feedback on their service and the steps they take to add new features to their service. Monitoring can be a valuable addition to testing and help to get quick feedback on the functional and technical levels on the service a team delivers. This feedback can be used to shape the service in a way it fits the customer.

The role of a test engineer changes to a DevOps Engineer with a test expertise. The test expertise as part of the T-shaped model consists of knowledge and skills to implement and teach test strategies in a team. The test expertise should be able to connect Business and IT and different kinds of expertise in a team to get all the requirements for quality in a service. Responsibility for testing must be shared with the entire team. The engineer with test expertise can take a leading role in this and act as a coach.

References

1. Shafer, A.: Agile infrastructure. Speech presented at Velocity Conference (2009)
2. Mezak, S.: The origins of DevOps: What's in a name? https://devops.com/the-origins-of-devops-whats-in-a-name/ (2018). Accessed 24 January 2018
3. 6 Principles of DevOps. https://www.devopsagileskills.org/dasa-devops-principles/ (2017)
4. Manifesto for Agile Software Development. https://agilemanifesto.org/ (2001)
5. Herzberg, F., Mausner, B., Synderman, B.B.: The Motivation to Work. Wiley, New York (1959)
6. Guest, D.: The hunt is on for the renaissance man of computing. The Independent (1991, September 17)
7. Kim, G., Behr, K., Spafford, G.: The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win. IT Revolution Press, Portland, OR (2013)
8. International Software Testing Qualifications Board: Certified Tester Foundation Level Syllabus Version 2018 Version. https://www.istqb.org/downloads/send/51-ctfl2018/208-ctfl-2018-syllabus.html (2018)
9. Humble, J., Farley, D.: Continuous Delivery. Addison-Wesley, Upper Saddle River, NJ (2011)
10. Deming, W.E.: Out of the Crisis. MIT Press, Cambridge (2000)
11. Aichernig, B.K.: Contract-based testing. In: Formal Methods at the Crossroads. From Panacea to Foundational Support, pp. 34–48. Springer, Berlin, Heidelberg (2003)
12. Richardson, C.: What are microservices? https://microservices.io/ (2018)
13. Adzic, G.: Specification by Example: How Successful Teams Deliver the Right Software, p. 3. Shelter Island, NY, Manning (2012)
14. Dinwiddie, G.: The Three Amigos: All for One and One for All. Better Software. https://www.stickyminds.com/sites/default/files/magazine/file/2013/3971888.pdf (2011). Accessed November/December 2011
15. North, D.: Behavior Modification. Better Software (2006, June 5).
16. Gherkin Reference. https://cucumber.io/docs/gherkin/reference/ (n.d.)
17. Cucumber. https://cucumber.io/ (n.d.)
18. Crispin, L., Gregory, J.: Agile Testing. Addison-Wesley, Boston, MA (2008)
19. Fowler, M.: Bliki: TestPyramid. https://martinfowler.com/bliki/TestPyramid.html (n.d.)

The Tester Skills Program

Teaching Testers to Think for Themselves

Paul Gerrard

Abstract In 2018, Gerrard Consulting was approached by the IT@Cork Skillnet (https://www.skillnetireland.ie; https://www.itcork.ie) and Softest (https://softtest.ie) organizations and asked to support an initiative that aimed to improve the skills of testers in the south of Ireland software community.

Around 20 testing and QA managers had decided to look at their challenges and plan a way forward to develop and improve the testing-related training available to local testers. The first meeting took place on 29 November 2018. The presentation that introduced the initiative was titled, 'Developing Testing Skills to Address Product Quality Challenges'.

What started as an attempt to create a 3-day class for beginner testers became a much more substantial learning and development (L&D) program. This chapter describes the reasons why the program is necessary, the current status and how it is likely to evolve in the future.

Keywords Software testing · Software quality · Software testing skills · ISTQB · Software tester

1 Introduction

1.1 Background

In 2018, Gerrard Consulting was approached by the IT@Cork Skillnet [1, 2] and Softest [3] organizations and asked to support an initiative that aimed to improve the skills of testers in the south of Ireland software community.

Around 20 testing and QA managers had decided to look at their challenges and plan a way forward to develop and improve the testing-related training available to local testers. The first meeting took place on 29 November 2018. The presentation

P. Gerrard
Gerrard Consulting, Macclesfiled, UK

S. Goericke (ed.), *The Future of Software Quality Assurance*,
https://doi.org/10.1007/978-3-030-29509-7_4

that introduced the initiative was titled, 'Developing Testing Skills to Address Product Quality Challenges'.

What started as an attempt to create a 3-day class for beginner testers became a much more substantial learning and development (L&D) program. This chapter describes the reasons why the program is necessary, the current status and how it is likely to evolve in the future.

1.2 Stakeholders

The stakeholders in this program are:

it@cork Skillnet offers a broad range of training courses to address the varied training needs of our member companies, who operate at all levels across the IT sector and beyond.[1]

SoftTest is Ireland's Independent Software Testing Special Interest Group. Its goal is to facilitate knowledge sharing within the Irish software testing community.

Program Members are the group of software testing professionals representing software companies based in the south of Ireland. At the time of writing, there are 23 members from industries as diverse as software services, healthcare, FinTech, security, gaming, computer hardware, insurance, biotech and HR systems.

1.3 Initiation

The first meeting in November had two goals:

1. To introduce the participants and outline the difficulties faced by the Program Members
2. To identify the skills and capabilities required to achieve a professional and productive level of competence

James Trindle of McAfee introduced the session with a brief presentation summarizing the problems currently faced by software teams in acquiring and retaining talented testers. His talk offered a stark prospect, and in fact he called it an existential crisis.

Paul Gerrard facilitated the discussion of these problems to define the scope of the challenges faced. The meeting then split into smaller groups to brainstorm the skills requirements for a professional tester.

[1] it@cork is a leading not for profit independent organization representing the Tech Sector in the South of Ireland. it@cork manages it@cork Skillnet, which is funded by Skillnet Ireland and member company contributions. Skillnet Ireland is a national agency dedicated to the promotion and facilitation of workforce learning in Ireland. It is funded from the National Training Fund through the Department of Education and Skills.

2 Why a New Tester Skills Program?

This chapter starts with the existential crisis that companies face when hiring and retaining testers. Later sections provide a wider industry view and a proposed new skills set.

2.1 Existential Crisis for Testers

Testing Is Obsolete The general feeling was that the approaches offered by training providers, books and the certification scheme(s) are no longer fit for purpose. They are outdated and have not kept pace with the industry changes experienced by all members.

Replaced by Automation A common perception is that testers and testing in general can be replaced by automated approaches. Managers are swayed by the promise of Continuous Delivery (CD), pervasive automation and the emergence of machine learning and artificial intelligence. Testers have not found a way to articulate the reasons why testing is much harder to automate and eliminate than people believe.

How Do You Add Value to the Team? If you ask a tester what value they bring to their teams, they find it extremely difficult to make a strong case. The argument that testing is important is won already. But how do testers explain their value? My experience is that almost all testers don't know who their customers (stakeholders) are; they don't know what stakeholders want or how the information testers provide is used to make decisions still. As a profession (if testing actually is a profession) we have failed to make the case.

Titles Changing—Evolution of SDET Role Companies are implementing various ways of redistributing testing in their teams. The Shift-Left idea has caught on, with developers taking more responsibility, testers acting as coach/mentors to devs and other team members and being more closely involved in requirements. These are all good things. More popular in the US than Europe, the SDET (Software Development Engineer in Test) role is a hard one to fill. What is clear is that testing is in a state of flux. Testers are finding it hard to assimilate the change and to contribute towards it.

We're All Engineers; Everyone Must Write Code Related to the SDET approach, testers who never wrote code (and might not ever want to) are being encouraged to learn a programming or scripting language and automated test execution tools. The pressure to program and use tools is significant. This is partly because of the relentless marketing of the vendors. But it is also fuelled by a lack of understanding of what tools can and should do and what they cannot and should

not do. Testers (whether they use tools or not) are not well briefed in the case for automation or the strategies for successful tool implementation.

Once Highly Respected Skillset/Mindset no Longer Valued The Year 2000 threat caused many companies to take testing seriously and there was a brief period when testers were more highly respected. But when Agile appeared on the scene and was widely adopted, the role of testers was badly or not defined. They were expected to just 'get on with it'. Agile, at the start at least, was mostly driven as a developer initiative with little thought for how requirements and testing were done. After 15 years, testers have a much better idea of their role in Agile. Unfortunately, the next big thing is Continuous Delivery. The mantra is that testing, of whatever type, should be automated. Once again, testers are under pressure to re-define their role and/or get out of the project.

Technology Changing at Unprecedented Rate There's little doubt that test approaches have not kept pace with the changing technology. Although test execution tools appear within a year or two of new user interface technologies, the new risks, modelling approach and test management methods emerge very slowly. Tester skills seem to be tied to technologies. Skills should be independent of technology, enabling testers to test *anything*.

2.2 The Drive to Digital

Across the business world, there is a revolution in the way that IT is being specified, developed, implemented and used. There is lots of hype around the whole 'Digital Transformation' phenomena. Digital Transformation programs are affecting business across all industry and government sectors. There is no doubt that it also affects people in their daily lives.

Digital includes traditional IT but includes:

- Mobile anything
- The Internet of Things
- Autonomous vehicles
- Our home, workplace, public and private spaces
- Robots (physical)
- Bots (software)
- Artificial Intelligence, Machine Learning, Deep Learning
- And so on . . .

Digital visionaries promise a lot:

- No human intervention in your systems (Autonomous Business Models)
- Marketing messages created, sent, followed up and changed almost instantly
- Full range of data from the smallest locale to global in all media formats at your disposal

- Autonomous drones, trucks and cars can transport products, materials and people
- Physical products needn't be ordered, held in stock and delivered at all—3D printing removes constraints.

Mankind has built some pretty impressive systems (of systems). The most complex systems ever used to be the Space Shuttle with 2.5 m parts, but this was superseded by the Nimitz class supercarrier which has one billion parts. In fact, the carrier comprises thousands of interconnected systems and with a crew of 5000–6000, it could be compared to an average town—afloat.

Compare this with a 'Smart City'. A Smart City is

> an urban development vision to integrate multiple information and communication technology (ICT) and IoT solutions in a secure fashion to manage a city's assets—the city's assets include, but are not limited to, local departments' information systems, schools, libraries, transportation systems, hospitals, power plants, water supply networks, waste management, law enforcement, and other community services. (Wikipedia)

With a smart city, the number of connected nodes and endpoints could range from a million to billions. The smart city will be bigger, more complex than anything before. Connected citizens and many of the systems:

- Move in the realm of the city and beyond
- Interact in unpredictable ways
- Are places where citizens are not hand-picked like the military; crooks, spies and terrorists can usually come and go as they please

Unlike ships, smart cities are highly vulnerable to attack.

Digital systems will have a social impact on all citizens who encounter them. There are huge consequences as systems become more integrated with the fabric of society. Systems already monitor our every move, our buying, browsing and social activities. Bots push suggestions of what to buy, where to shop, who to meet, when to pay bills to us minute by minute.

Law enforcement is affected too. CCTV monitors traffic, people and asset movement and our behaviours. The goal might be to prevent crime by identifying suspicious behaviour and controlling the movement of law enforcement agents to places of high risk. But these systems have the potential to infringe our civil liberties and the legal frameworks are behind the technology.

Digital affects all industries.

Unlike Agile, which is an ongoing IT initiative, Digital is driven by Business. Agile has taken more than 15 years to get halfway implemented in the software

industry. Digital has taken no time at all—perhaps 2–3 years and it is all-pervasive in the West. Digital is the buzz-phrase of the moment.

> The speed of delivery is partly about pro-action, but it is also about survival.

Often, Chief Digital Officers are Marketers. Marketers move at the pace of marketing and they want change at the same pace. To the marketer, frequent software delivery is critical. Mobile users *expect* apps to change almost daily with new features, offers, opportunities appearing all the time. Users often don't care which supplier they use, as long as their apps work reliably, so businesses are in an APPS RACE.

> S/W development at the pace of marketing.

So, automation (and not just test automation) is critical. What business needs is IT responsiveness—what you might call *true agility.* This doesn't necessarily mean hundreds of releases every day; but it does mean business want rapid, regular turnaround from ideas to software delivery.

With continuous integration/deployment, DevOps, developers can now *promise Continuous Delivery*

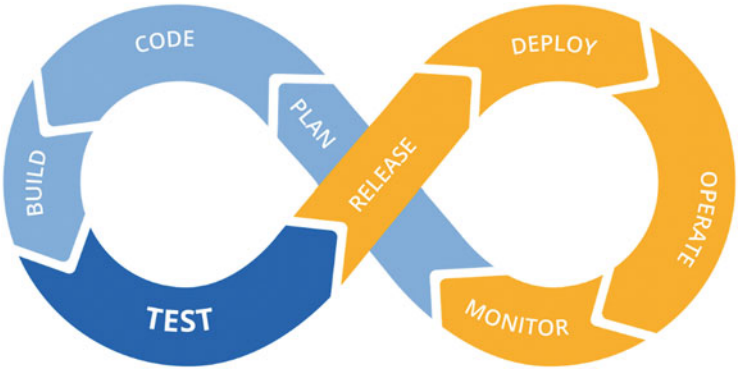
> Testers need to provide *Continuous Assurance*

This means automation through the (shortened) life cycle. What exactly is possible and impossible with automation, right here, right now? Are Continuous Delivery and DevOps the route to success? Could testing be the bottleneck that prevents success? How do testers operate in dynamic, high-paced, automation-dominated environments?

2.3 Waterfall Thinking Won't Work with Continuous Methods

Continuous Delivery or an adapted version of it is becoming increasingly popular in Digital projects and if Digital is the 'future' for the majority of organizations, then we had better prepare for it. Testers need to adapt to fit into their continuous delivery regimes so let's look at how continuous approaches are normally described.

The most common diagram one sees is the figure eight or infinite loop below. The principle is that the plan, code, build, test through release, deploy, operate and monitor phases are sequential but are repeated for every release.



But there's a problem here. If you unwrap the infinite loop, you can see that the phases are very much like the stages of a Waterfall development. There are no feedback loops, you have to assume one phase completes before another starts.

So, it appears that Continuous Delivery is just Waterfall in the small. What do we *know* about waterfall-style developments?

- *It's sequential*—one stage follows another—no variation
- *Dependencies rule*—you can't start one stage before the previous stage is done
- *It's not re-entrant*—no flexibility to react to external events
- *Testing has stages itself*—we *know* that testing has itself stages of thinking and activities spread through the process
- *Only one phase of testing*—but there are developer and tester manuals and automated test execution activities
- *Testing is squeezed*—timeboxed activities—the thinking, preparation and execution time is all limited
- *No feedback loop(s)*—we know that testing finds bugs—but the continuous process has no feedback loop.

If Agile has taught us anything, it's that the dependence on staged approaches made Waterfall unsuccessful in more dynamic environments.

> Staged thinking won't work in a continuous process.

We need another way of looking at process to make Continuous Delivery work.

2.4 Separating Thinking from Logistics

There are two problems to solve here:

1. The first is that there is no one true way or best practice approach to implementing, for example, continuous delivery. Everyone does it slightly differently, so any generic training scheme has to offer generic practices.
2. The second is that any credible training scheme must recognize that there are skills that can be taught in the classroom, but the *employer* must take on the role of explaining local practices and embedding skills.

These local practices are what we call *logistics*. Logistics are how a principled approach is applied locally. Locally might mean 'across an entire organization' or it could mean every project you work on works differently. If you work on multiple projects, therefore, you will have to adapt to different practices—even if you are working in the same team.

> Principles and thinking are global; logistics are local.

It's clear that to offer training alone is not enough. There must be a contribution by the local employer to nurture trainees, coach them in local practices and give them work that will embed the skills and local approach.

> To offer training to practitioners, we must separate the principles and thinking processes from the logistics.

How do we do this?

2.5 Logistics: We Don't Care

We need to think clearly and remove logistics from our thinking. The simplest way to do this is to identify the aspects of the local environment and descope them, so to speak. The way Paul usually introduces this concept is to identify the things that we don't care about.

As a practitioner you will care deeply about logistics, but for the purposes of identifying the things that are universally applicable—principles and our thought process—we need to set them aside for the time being. Here are the key logistical aspects that we must remove 'to clear our minds'.

Document or Not? We don't care whether you document your tests or not. Whether and how you record your tests is not germane to the testing thought process.

Automated or Manual? We don't care whether you run tests by hand, so to speak, or use a tool, or use some form of magic. It isn't relevant to the thought process. The mechanism for test execution is a logistical choice.

Agile vs. Waterfall? We don't care whether you are working in an Agile team or in a staged, waterfall project or are part of a team doing continuous delivery. It's not relevant to the testing thought process.

This Business or That Business? We don't care what business you are in whether it is banking or insurance or healthcare or telecoms or retail. It doesn't matter.

This Technology vs. That Technology? We don't care what technology you are working with. It's just not relevant to the thought process.

Programmer or Tester? We don't care who you are—developer, tester, user business analyst—the principles of testing are universal.

Test Manager or No Test Manager? We don't care whether you are working alone or are part of a team, with or without a test manager overseeing the work. This is a logistical choice, not relevant to the testing thought process.

2.6 Without Logistics: The New Model for Testing

If we dismiss all these logistics—what's left? Some people might think we have abandoned everything, but we haven't. If you set aside logistics, what's left is what might be called the universal principles and the thought process. Now, you might think there are no universal principles. But there clearly are—they just aren't muddied by local practices. Paul's book, *The Tester's Pocketbook* [4, 5] identifies 16 so-called *Test Axioms*.

Some Axioms, for example the stakeholder axiom, 'Testing Needs Stakeholders', are so fundamental they really are self-evident. Other axioms such as the Sequencing axiom, 'Run our most valuable tests first—we may not have time to run them later',

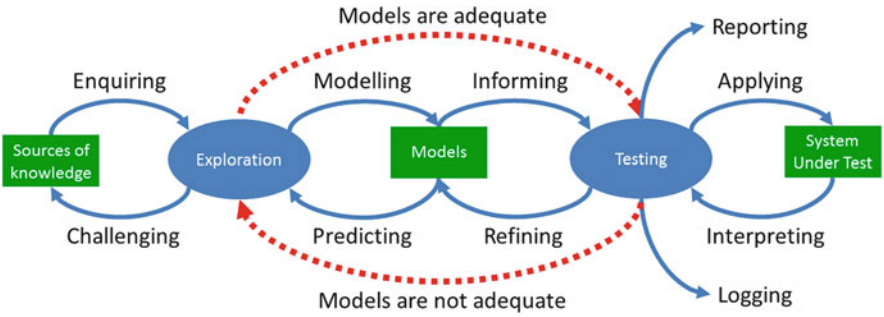


Fig. 1 The new model for testing

are more prosaic—it sounds logistical. But sequencing is a generally good thing to do—HOW you prioritize and sequence is your logistical choice.

The New Model for Testing is an attempt to identify the critical testing thought processes (Fig. 1). A Webinar [5] and white paper [6] give a full explanation of the thinking behind the model, which is reproduced below.

The model doesn't represent a process with activities, inputs, outputs, entry and exit criteria and procedures. Rather it represents the *modes of thinking* which people who test go through to achieve their goals. Our brains are such wonderful modelling engines that we can be thinking in multiple modes at the same time and process thoughts in parallel. It might not be comfortable, but from time to time, we must do this.

The New Model suggests that our thinking is dynamic and event-driven, not staged. It seems like it could be a good model for testing dynamic and event-driven approaches like continuous delivery.

> Using the New Model as the basis for thinking fits our new world of testing.

The ten thinking activities all have associated real activities (logistics usually) to implement them and if we can improve the way we think about the testing problem, we are better able to make informed choices of how we logistically achieve our goals.

There are several consequences of using the New Model. One aspect is how we think about status. The other is all about skills.

2.7 Rethinking Status

As a collaborative team, all members of the team must have a shared understanding of the status of, for example, features in the pipeline. Now, the feature may be placed

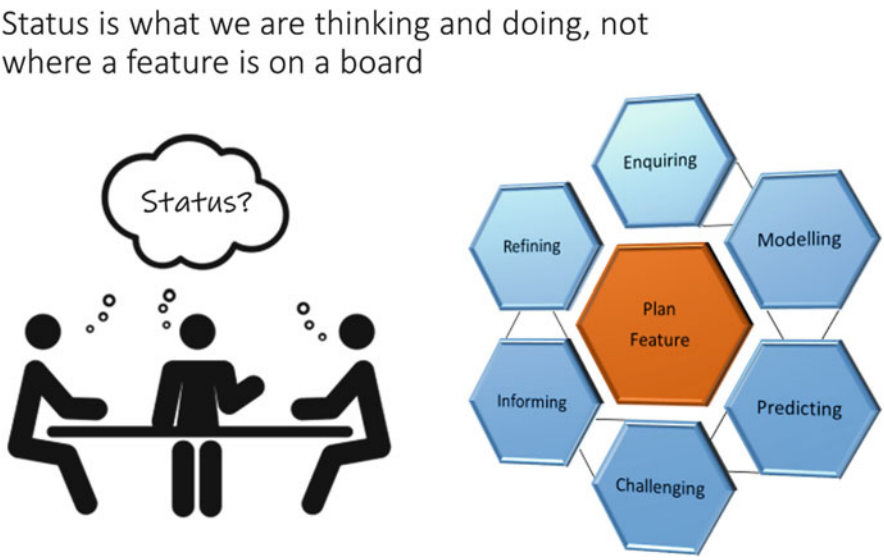


Fig. 2 Status is what we are thinking

somewhere on a Kanban or other board, but does the location on the board truly represent the status of the feature?

Consider the 'three amigos' of user, developer and tester. When it comes to agreeing status, it is possible that the user believes their job is done—the requirement is agreed. The developer might be writing code—it's a work-in-progress. But the tester might say, 'I have some outstanding challenges for the user to consider. There are some gaps in the story and some ambiguous scenarios we need to look at' (Fig. 2).

What is the status of the feature? Done? Work in Progress? Or under suspicion? When all three participants are thinking the same way, then there is a consensus on the status of the feature.

> Continuous Collaboration is an essential part of continuous delivery. The New Model provides a framework for meaningful discussion and consensus.

2.8 The (Current) Problem with Certification

The most prominent tester certification scheme is created and administered by the International Software Testing Qualifications Board—ISTQB [7]. The training classes run and qualifications awarded number 875,000+ and 641,000, respectively.

There are some minor schemes which operate, but ISTQB has become the de facto standard in the industry.

But there are well-known problems with current certification:

- If you look at the certification scheme syllabuses (Foundation and Advances Test Analyst, for example), the table of contents comprises mostly what we have called logistics. *The certification schemes teach many of the things we say we do not care about.*
- The schemes mostly offer one way of implementing testing—they are somewhat aligned with various standards. Incident Management, Reviews, Test Planning, Management and Control are all prescriptive and largely based on the Waterfall approach.
- Much of the syllabus and course content is about remembering definitions.
- The syllabuses infer test design techniques are procedures, where the tester never models, or makes choices. The tester and exam-taker are spoon fed the answers rather than being encouraged to think for themselves. There is no modelling content.
- The syllabuses don't teach *thinking skills.* The word *thinking* appears once in the 142 pages of Foundation and Advanced Test Analyst syllabuses.
- Exams, being multiple choice format, focus on remembering the syllabus content, rather than the competence of the tester.

> Certification does not improve your ability to be independent thinkers, modelers or (pro-)active project participants; the exams do not assess your ability as a tester.

This is a big problem.

3 The Tester Skills Program

After that lengthy justification for a new approach to thinking and inevitably skills acquisition, this chapter focuses on the strategy for the development of a Tester Skills Program (TSP).

Work started on the TSP in late 2018 in line with a provisional plan agreed with it@cork Skillnet. But as time passed it became obvious that the initial goal of creating a 3-day beginner class would not satisfy the requirements of the Program Members. The challenges faced and the range of topics required to fulfil the needs of a professional tester were much more ambitious than could be delivered in just 3 days. With hindsight, this was obvious, but we tried to align with the plan as agreed.

The current strategy emerged over the early months.

3.1 Skills Focus

There were several influences on the depth and range of skills needed and consequently, there are a range of objectives for the program:

- The syllabus would focus on non-logistics, that is the principles, more independent thinking, modelling and people skills. (As a consequence, there is little overlap with ISTQB syllabuses.)
- Practitioners need skills that allow them to work in teams that use pervasive automation for environments, builds, tests.
- Practitioners might be expected to work in teams where continuous, event-driven processes are emerging or being adopted.
- The range of skills implies a broader role of assurance is required which spans requirements through to testing in production; new disciplines of shift-left, continuous delivery, Digital eXperience Optimization (DXO) require consideration.
- Practitioners are assumed to be part of mixed, multidisciplinary teams and must have basic project, collaboration, interpersonal and technical skills.

Skills should align with a better-defined goal of testing: to increase definition and delivery understanding.

3.2 New Tester Behaviours

Program members wanted the scheme to encourage specific behaviours of practitioners:

- To think more analytically (modelling, systems and critical thinking)
- To move from passive to active collaboration; to challenge and refine requirements
- To understand customer or digital experience optimization; to be aware of and exploit other predictive models and align testing to these models
- To act like a pathfinder or navigator (rather than a 'follower')
- To collaborate with confidence and at more senior technical and business/stakeholder levels

The TSP syllabus aims to encourage more outward-facing, collaborative, proactive behaviour.

3.3 TSP Is a Learning and Development Scheme

Clearly, TSP is more than a few days classroom training. The training must be part of a more comprehensive L&D regime. Training can impart new ideas, concepts and skills, but to trigger new behaviours, these skills must be embedded in the practitioners' mind and aligned with local ways of working.

For every hour of training material, there needs to be 1–2 h support, assignments and practical work to achieve the goal of new behaviours. Employers are encouraged to support learners by answering their questions and providing local logistics knowledge to complete the learning process.

> When learners do receive line manager support, 94% go on to apply what they learned. There's a positive correlation between the transfer of learning to the workplace, line manager support and performance improvement.
>
> (Kevin Lovell, Learning Strategy Director at KnowledgePool)

3.4 The Skills Inventory

The initial work of the Program Members was to define the range of skills required for a testing practitioner. It was understood at the outset that the range of skills meant that there had to be a graduated set of L&D schemes. The Skills Inventory would be a shopping list of topics that could be part of Foundation, Advanced or Mastery level schemes.

The summary Topic Areas in the inventory appear below:

- Adapting Testing
- Advanced Testing
- Agile Testing Approaches
- Assertiveness
- Certification
- Challenging Requirements
- Collaboration
- Communication
- Critical Thinking
- Developer Testing
- Exploratory Testing
- Exploring Sources of Knowledge
- Facilitation
- Hiring Testers
- Instrumentation
- Modelling
- Monitoring
- Non-Functional Testing
- Process Improvement

- Planning
- Reconciliation
- Regression Testing
- Requirements Test Analysis
- Risk Management
- SDET Role
- Technical Testing
- Technology Skills
- Systems Thinking
- Test Assurance
- Test Automation Frameworks
- Test Automation
- Coaching
- Test Design—Model-Based
- Test Design—Domain
- Test Design—State-Based
- Test Design—Logic
- Test Design—Purposeful Activity
- Test Motivation
- Test Strategy
- Testability
- Testing and Stakeholders
- Testing Fundamentals
- Testing in Teams
- Working Remotely

As you can see, there are quite a few topics that you won't find on common test training courses. Personal and professional development topics include Critical Thinking, Assertiveness, Collaboration, Communication, Facilitation, Hiring, Process Improvement, Systems Thinking, Coaching, Testing and Stakeholders, Testing in Teams and Working Remotely.

3.5 Program Member Challenges

As part of the discussion of the Existential Crisis, the Program Members identified a range of challenges that face them. Not everyone has the same challenges, but the list below gives an indication of the kind of problems being faced in tester recruitment, education and retention. For each challenge, the relevant skills topic area(s) have been assigned. Understanding these challenges is helping a lot to define the syllabus topics. In this way, the syllabus focuses on the right problems.

Challenges	Skills areas
Tester candidates: on paper look great, but they don't seem to be able to analyse a requirement and be a tester	Requirements Test Analysis, Testing Fundamentals
Instilling sense that you need to test your work, college kids want to write code, but not test their own work	Test Motivation, Testing Fundamentals
Agile teams, dev and test teams work as 'agile' but not together	Testing in Teams
Cafeteria agile, teams choose to do what teams like to do	Adapting Testing to Change, Agile Testing
Lots of hands on deck to get things delivered, but who leads team? Who leads on testing?	Test Leadership, Testing in Teams
Is the tester the lead on quality? If not, who is?	Testing in Teams
Everyone does their own thing, but who sets the strategy? Gaps and overlaps?	Testing in Teams, Test Strategy
Brief sprint—devs want to hand off asap, but testers are left behind, left with questions	Testing in Teams
Changed focus from testing to quality engineering—but what is quality engineering?	Adapting Testing
From tester in team responsible for automation and acceptance, should they move towards being a test coach, testmaster?	Coaching
Risk analysis, exploration	Risk Management
Skills needed: coaching, exploration and risk management	Coaching
Architecting for testability	Testability
TDD and role in test strategy	Developer Testing
Good design for test—what is it? How to recognize and encourage developers	Developer Testing, Testability
Exploratory testing, critical mindset, seeing the difference in confirmation and challenging the product	Exploratory Testing, Critical Thinking, Test Motivation
Coaching, within the team, devs and users, but also across teams, encouraging observability, logging, instrumentation	Coaching, Monitoring, Instrumentation
Macro-level consistency checks, instrumentation and logging	Monitoring, Instrumentation, Reconciliation
Influencing teams—helping them to spot flaws, educating, leading teams, facilitation	Coaching. Facilitation
Role for policing?	Test Assurance
Leading retrospectives, continuous improvement, leaving room for innovation and improvization	Coaching, Process Improvement
Balance between control and innovation	Process Improvement
Does coaching operate only on a local level?	Coaching
Is there a difference between coaching testers and developers? At an individual vs. team level?	Coaching
Challenging requirements, user stories, etc.	Challenging Requirements
Waterfall test has no voice. In agile we have a voice. To be pro-active is a matter of critical thinking but also guts	Testing in Teams, Assertiveness, Critical Thinking

(continued)

Coaching up as well as down (managers)	Coaching
Communication skills, how to articulate questions and information	Communication
Critical thinking, and influencing	Critical Thinking, Communication
Critical testing skills	Testing Fundamentals
Collaboration skills, testers create tests and automation in isolation. Testers bring in pairing, risk discussions, bug bashes	Collaboration, Communication
Testing as an activity not a role, but organizations exist to achieve outcomes	Testing and Stakeholders
'There are people who lose their ability to think', don't ask questions, don't challenge	Test Motivation
Critical thinking	Critical Thinking
Understanding what devs do well and testers/SDETSs do well (and not so well)	Developer Testing, SDET Role
High-performing teams have devs writing automation	Test Automation
What do stakeholders need from testers?	Testing and Stakeholders
Can you teach exploratory testing? Is it a mindset? Or an aptitude?	Exploring Sources of Knowledge, Exploratory Testing
BBST is a much more in-depth regime. Big investment of time.	Advanced Testing
Coach in all directions	Coaching
Devs need better test skills too!	Developer Testing
Mastery vs. capability?	Advanced Testing
How long to achieve (assessment of?) capability?	Certification, Advanced Testing
How long to achieve (assessment of?) mastery?	Certification, Advanced Testing
Qualification, certification?	Certification, Hiring Testers
A lot of testers lack the confidence/aptitude to explore and to question software	Test Motivation, Testing Fundamentals, Exploratory Testing
Most people are not critical thinkers	Test Motivation, Critical Thinking
Certification used as a tick-box to impress employers, not to improve skills	Certification
Testers unwilling to learn, improve	Test Motivation, Adapting Testing
Reluctance to test below the UI	Technical Testing, Testability
Unwillingness to test 'outside the box'	Test Motivation, Modelling
Inability to demonstrate the value of testing	Testing and Stakeholders, Test Motivation

3.6 Structure of the Foundation Scheme

At the time of writing, only the Foundation Level scheme is defined, although the syllabus and course content are still work-in-progress. The high-level syllabus appears over the page.

The teaching content comprises approximately 40 h of material; this would be supported by 40–80 h of assignments and offline discussion with peers and managers, reading and further research. Much of the early training, on test fundamentals gives learners a set of questions to ask and discuss with peers, managers and stakeholders.

Beyond that, the assignments tend to be one of the following:

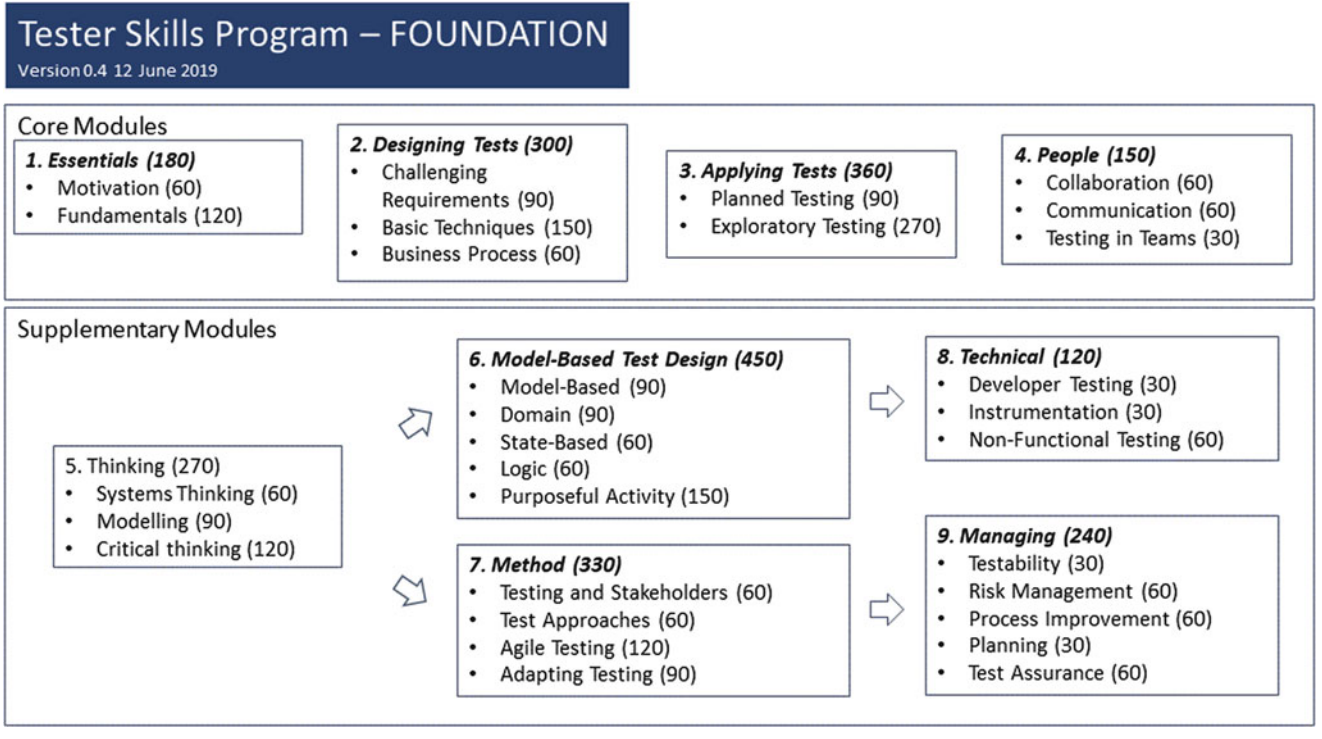
- Research, reading and study on testing-related issues
- Topics such as test design have specific assignments, such as requirements to analyse and online applications to explore and test.
- Modelling (focusing on requirements, stories, software, usage, tests)
- Practical test assignments (exploration, test design and bug-finding)

The scale of teaching is such that our expectation is that most companies would opt for a mix of classroom instructor-led, online instructor-led and purely online teaching. ALL training material will be presentable in all three formats. For the initial pilot classes, the 'core' modules would be presented in a classroom and feedback obtained. It is anticipated that the non-core modules would be usually accessed online.

Since a focus of the scheme is to help people to adapt to dynamic project environments, the thrust of the training is to help testers to think for themselves. A core component of this is the systems, critical and testing thinking modules. Not every learner will be comfortable with all of this material, and systems thinking in particular focuses on broader problem-solving than just testing. But exposure to systems thinking is still deemed to be of positive benefit.

Finally, there are some 'people skills' modules. These are intended to provide insights into the challenges of working in teams, collaboration and basic communication skills. At the Foundation level these are purely introductory and the intention is to get learners to at least pay attention to people issues. The Advanced scheme is likely to go into much more depth and offer specific personal skills modules.

Overall, the goal of the Foundation level is to bring new hires with little or no testing experience to a productive level. We have not used Bloom's taxonomy to assign learning objectives, but the broad goal is for learners to achieve K4 level 'Analysis' capability in the analysis and criticism of requirements and the selection of models and modelling and testing of systems based on requirements or exploratory surveys. *Compare this with the ISTQB Foundation, where the learning objectives are split K1: 27%. K2: 59%, K3: 14%—a far less ambitious goal* [8].



4 The Future

4.1 Engaging Competent Training Providers to Deliver TSP

One of the deliverables of the scheme is a template 'Invitation to Tender' for the Foundation level training. it@cork Skillnet require a means of inviting training providers to build and deliver classes against the TSP Syllabuses. In this way, companies can compete to deliver training, and this should ensure prices are competitive.

One of the shortcomings of the certification schemes is the sheer size of the syllabuses. The ISTQB Foundation and Advanced Test Analyst documents add up to 142 pages. The TSP Foundation syllabus—roughly equivalent in duration, if not content, will be less than 10% of that length.

One of the reasons the ISTQB syllabuses are so over-specified is that it allows non-testing professionals to deliver certified training. The thrust of the TSP is that the materials are bound to be delivered by experienced testing professionals (who may have left their testing career behind but are nevertheless qualified to deliver good training). The TSP Foundation allows trainers to teach what they know, rather than 'what is written' and means trainers are in a good position to answer the tricky questions that are built into many of the assignments.

4.2 Could the TSP Be the Basis of Certification?

This currently isn't on the agenda in the Program Membership. However, the ISTQB scheme which relies entirely on multiple choice exams and requires no employer or peer review or practical experience could be improved upon.

TSP at least attempts to engage peers and managers in the support of learners. Certainly, the assignments could be compiled into a 'workbook' which could contain a summary of the assignments, verified by a peer, manager or mentor. These could demonstrate that learners have done their homework at least.

In common with professional engineering bodies, a formal certification scheme would require some proof of relevant work experience and perhaps some original work relating to the experience or further study in a subject related to the TSP topic areas. The mechanical, civil or electrical engineering professional bodies in the UK (or elsewhere) could be examined perhaps to derive a scheme that would go some way towards making testing a real profession.

4.3 Classroom, Online Instructor-Led or Online Teaching?

The goal to date is to offer all course materials in a format that could be used in a classroom or online delivery. Although most practitioners would prefer classroom courses, the flexibility and lower cost of online training is attractive. The choice is likely to be driven by whether the student is, for example, self-employed or employed by a larger organization preferring self-study or instructor-led teaching.

The market will determine what formats are most appropriate and commercially successful.

4.4 Proliferating the TSP

Could the TSP be adopted as worldwide standard? Time will tell. But there are opportunities to proliferate the TSP scheme [9].

Irish Commercial Adoption The Program Members intention is to adopt the Foundation scheme for their graduate intake. it@cork Skillnet will partially subsidize the training and encourage the scheme to be adopted across Ireland. It is to be hoped that local training providers create (or cross-license) training material and 'train their trainers' to deliver classes.

University/College Adoption It would be extremely helpful to have universities and colleges teach the TSP program as part of their curriculums. This would allow the academic institutions to make their graduates more marketable and remove some less attractive content. Although there are early discussions in Ireland, the general view is the academics won't move until the TSP scheme is a proven success. Time will tell.

Overseas Commercial Adoption Gerrard consulting will promote the scheme in both Ireland and the UK in the belief that the scheme offers a significant uplift in the quality and value of testing and assurance related education. Being the facilitator and first provider of training gives a commercial advantage, but to scale up the scheme, other providers must come into the market.

Creating One's Own TSP It goes without saying, that the process of development of a local TSP could be repeated in other regions. There is no reason, in principle, why other regions can't identify their own local skills needs and build their own program. The 'not invented here' syndrome may also have an effect and bigger training providers might want exclusive rights to their own scheme in their locality.

It would be nice to think that a global TSP standard could be the ultimate outcome of this work, but commercial pressures and human nature suggest different schemes might spring up if the Irish TSP is deemed to be a success.

References

1. Skillnet Ireland. https://www.skillnetireland.ie. Accessed 19 July 2019
2. it@cork. https://www.itcork.ie. Accessed 19 July 2019
3. https://softtest.ie. Accessed 19 July 2019
4. https://testaxioms.com. Accessed 19 July 2019
5. Agile Testing Days Webinar. The new model for testing. https://youtu.be/1Ra1192OpqY. Accessed 19 July 2019
6. A New Model for Testing – white paper. https://gerrardconsulting.com/mainsite/wp-content/uploads/2019/06/NewModelTestingIntro.pdf. Accessed 19 July 2019
7. Gerrard, P.: The Tester's Pocketbook. Testers' Press, Maidenhead, UK (2011)
8. https://istqb.org. Accessed 19 July 2019
9. https://testerskills.com. Accessed 19 July 2019

Testing Autonomous Systems

Tilo Linz

Abstract The development of autonomous vehicles is currently being promoted massively, not least in the German automotive industry, under very high investments. The railway industry, shipbuilding, aircraft industry, and robot construction are also working on further developing their products (trains, ships, drones, robots, etc.) into self-driving or autonomous systems.

This chapter therefore discusses the question in which aspects the testing of future autonomous systems will differ from the testing of software-based systems of today's character and gives some suggestions for the corresponding further development of the test procedure.

Keywords Software testing · Software quality · Autonomous vehicles · Autonomous systems

1 Motivation

The development of autonomous vehicles is currently being promoted massively, not least in the German automotive industry, under very high investments. The railway industry, shipbuilding, aircraft industry, and robot construction are also working on further developing their products (trains, ships, drones, robots, etc.) into self-driving or autonomous systems.

The world's leading research and advisory company Gartner provides the following assessment in its report *Top 10 Strategic Technology Trends for 2019: Autonomous Things* [1]:

- By 2023, over 30% of operational warehouse workers will be supplemented by collaborative robots.

T. Linz
imbus AG, Möhrendorf, Germany

S. Goericke (ed.), *The Future of Software Quality Assurance*,
https://doi.org/10.1007/978-3-030-29509-7_5

- By 2025, more than 12% of newly produced vehicles will have autonomous driving hardware capability of Level 3 or higher of the SAE International Standard J3016.[1]
- By 2022, 40 of the world's 50 largest economies will permit routinely operated autonomous drone flights, up from none in 2018.

It can be assumed that within the next 10 years mobile systems will conquer the public space and be autonomously (or at least partially autonomously) "on the way" there.

The degree of autonomy of these systems depends on whether and how quickly manufacturers succeed in equipping their respective products with the sensors and artificial intelligence required for autonomous behavior.

The major challenge here is to ensure that these systems are sufficiently safe and that they are designed in such a way that they can be approved for use in public spaces (road traffic, airspace, waterways). The admissibility of the emerging systems and their fundamental social acceptance depend on whether the potential hazards to humans, animals, and property posed by such systems can be minimized and limited to an acceptable level.

Consensus must be reached on suitable approval criteria and existing approval procedures must be supplemented or new ones developed and adopted. Regardless of what the approval procedures will look like in detail, manufacturers will have to prove that their own products meet the approval criteria.

The systematic and risk-adequate testing of such products will play an important role in this context. Both the Expert Group on Artificial Intelligence of the European Commission and the Ethics Commission "Automated and Networked Driving" set up by the German Federal Minister of Transport and Digital Infrastructure explicitly formulate corresponding requirements for testing in their guidelines [3, 4].

This chapter therefore discusses the question in which aspects the testing of future autonomous systems will differ from the testing of software-based systems of today's character and gives some suggestions for the corresponding further development of the test procedure.

2 Autonomous Systems

We understand the term "Autonomous System" in this chapter as a generic term for the most diverse forms of vehicles, means of transport, robots, or devices that are capable of moving in space in a self-controlling manner – without direct human intervention.

An older term for such systems is "Unmanned System (UMS)" [5]. The term emphasizes the contrast with conventional systems that require a driver or pilot on board and also includes nonautonomous, remote-controlled systems.

The modern term is "Autonomous Things (AuT)" [6]. This term is based on the term "Internet of Things (IoT)" and thus conveys the aspects that Autonomous

[1] See [2].

Systems can be networked with each other and with IT systems on the internet, but also the development towards (physically) ever smaller autonomous things.

Examples of[2] such systems are:

- Motor vehicles (cars, lorries) which partially or (in the future) completely take over the function of the driver[3]
- Driverless transport vehicles that are used, for example, for logistics tasks and/or in production facilities[4]
- Ocean-going vessels, boats, inland waterway vessels, and other watercrafts[5] which are used, for example, for the transport of goods
- Driverless underwater vehicles or underwater robots which, for example, carry out[6] inspection or repair tasks under water independently
- Driverless trains, suburban trains, underground trains, or train systems for passenger or freight transport[7]
- Unmanned or pilotless aircrafts, helicopters, or drones[8]
- Mobile robots, walking robots, humanoid robots that are used for assembly, transport, rescue, or assistance tasks[9]
- Mobile service or household robots, for example, automatic lawn mowers or vacuum cleaners, which carry out service work in the household[10] and communicate with the "Smart Home" if necessary

Although all these systems are very different, they share some common characteristics:

- These are cyber-physical systems, that is, they consist of a combination of "informatic, software-technical components with mechanical and electronic parts."[11]
- They are mobile within their operational environment, that is, they can control their movements themselves and navigate independently (target-oriented or task-oriented).

[2]The listed examples name civil areas of application. However, the development of autonomous systems and corresponding technologies has been and continues to be strongly motivated and financed also because of their potential applications in the military sector.

[3] https://en.wikipedia.org/wiki/Autonomous_car, https://de.wikipedia.org/wiki/Autonomes_Fahren

[4] https://en.wikipedia.org/wiki/Automated_guided_vehicle, https://de.wikipedia.org/wiki/Fahrerloses_Transportfahrzeug

[5] https://en.wikipedia.org/wiki/Autonomous_cargo_ship, https://en.wikipedia.org/wiki/Unmanned_surface_vehicle

[6]https://en.wikipedia.org/wiki/Autonomous_underwater_vehicle

[7]https://en.wikipedia.org/wiki/Automatic_train_operation, https://en.wikipedia.org/wiki/List_of_automated_train_systems

[8]https://en.wikipedia.org/wiki/Unmanned_aerial_vehicle

[9]https://en.wikipedia.org/wiki/Robot#General-purpose_autonomous_robots, https://en.wikipedia.org/wiki/Autonomous_robot, https://en.wikipedia.org/wiki/Legged_robot, https://en.wikipedia.org/wiki/Humanoid_robot

[10]https://en.wikipedia.org/wiki/Service_robot

[11]https://en.wikipedia.org/wiki/Cyber-physical_system

- They can perform a specific task (e.g., mowing the lawn) or head for a specific destination (e.g., "drive to Hamburg") without having to specify the details of the task or the exact route in advance.

2.1 Autonomy and Autonomy Levels

"Autonomy" (of an UMS) is defined in [5] as: "A UMS's own ability of integrated sensing, perceiving, analyzing, communicating, planning, decision-making, and acting/executing, to achieve its goals as assigned by its human operator(s) through designed Human-Robot Interface (HRI) or by another system that the UMS communicates with."

The degree to which an autonomous system fulfils these properties (*sensing, perceiving, analyzing*, etc.) can be very different. In order to be able to classify systems according to their degree of autonomy, various classification systems were defined.

A well-known scale of this kind is the classification of autonomy levels for autonomous driving according to SAE Standard J3016 (see [2]). The following table is a simplified representation of these levels based on [7]:

SAE level	Name	Description	Control	Environment observation	Fallback
0	No automation	The driver drives independently, even if supporting systems are available.	Driver	Driver	–
1	Driver assistance	Driver assistance systems assist in vehicle operation during longitudinal or lateral steering.	Driver and system	Driver	Driver
2	Partial automation	One or more driver assistance systems assist in vehicle operation during longitudinal and simultaneous lateral control	System	Driver	Driver
3	Conditional automation	Autonomous driving with the expectation that the driver must react to a request for intervention.	System	System	Driver
4	High automation	Automated guidance of the vehicle without the expectation that the driver will react to a request for intervention. Without any human reaction, the vehicle continues to steer autonomously.	System	System	System
5	Full automation	Completely autonomous driving, in which the dynamic driving task is performed under any road surface and environmental condition, which is also controlled by a human driver.	System	System	System

The SAE levels are structured according to the division of tasks between driver and vehicle.[12] For robots and other basically driverless, autonomous systems, a more general definition is needed. [5] defines a generic framework for "Autonomy Levels for Unmanned Systems (ALFUS)" that is applicable to all types of UMS or autonomous systems with three assessment dimensions:

1. Mission Complexity (MC)
2. Environmental Complexity (EC)
3. Human Independence (HI)

The framework describes how a metric-based classification can be performed within each of these dimensions and how an overall system rating ("Contextual Autonomous Capability") can be determined from this.

2.2 Capabilities of Fully Autonomous Systems

A fully autonomous system should be able to accomplish a predetermined mission goal without human intervention. For a service robot, one such goal could be "get me a bottle of water from the kitchen." A fully autonomous car should be able to drive its passengers "to Hamburg."

The system must be able to navigate autonomously in its respective environment. And it must be able to detect previously unknown or ad hoc obstacles and then avoid them (e.g., by an autonomous vehicle recognizing a blocked road and then bypassing it), or remove them (e.g., by a service robot opening the closed door that blocks the way to the kitchen).

In more general terms, this means that a fully autonomous system must be able to recognize and interpret situations or events within a certain spatial and temporal radius. In the context of the identified situation, it must be able to evaluate possible options for action and select the appropriate or best option with regard to the mission objective and then implement it as measures.

3 Safety of Autonomous Systems

It is obvious that a self-driving car or autonomous robot poses a danger to people, animals, objects, and infrastructure in its vicinity. Depending on the mass and movement speed of the system (or of system parts, e.g., a robotic gripping arm), the danger can be considerable or fatal. Possible hazard categories are:

[12] [2] itself avoids the term "autonomous" because "... in jurisprudence, autonomy refers to the capacity for self-governance. In this sense, also, 'autonomous' is a misnomer as applied to automated driving technology, because even the most advanced ADSs are not 'self-governing' .... For these reasons, this document does not use the popular term 'autonomous' to describe driving automation."

- Infringement of uninvolved third parties by the autonomously moving system
- The violation of direct users, operators, or passengers of the autonomous system
- Injury to animals or damage to objects or infrastructure in the track or operating radius of the system by the system
- Damage to other objects caused by objects that the system handles or has handled
- Damage to the system itself, for example, due to a maneuvering error

Since human intervention may take place too late in a dangerous situation or (for systems with a high autonomy level) is not planned at all, the autonomous system itself must be sufficiently safe. In the overall life cycle of an autonomous system (from development to deployment to decommissioning), the topic of "safety" therefore has an extraordinarily high priority.

The associated safety levels (SIL levels) are defined in the series of standards [8]. The term "safety" is defined there as:

- Freedom from unacceptable risk of physical injury or of damage to the health of people, either directly, or indirectly as a result of damage to property or to the environment. [9].

To ensure sufficient safety, a system must have "functional safety":

- Functional safety is the part of the overall safety that depends on a system or equipment operating correctly in response to its inputs. Functional safety is the detection of a potentially dangerous condition resulting in the activation of a protective or corrective device or mechanism to prevent hazardous events arising or providing mitigation to reduce the consequence of the hazardous event ...
- ... The aim of Functional safety is to bring risk down to a tolerable level and to reduce its negative impact. [9].

3.1 Safety in Normal Operation

The dangers described above primarily result from the movement of the system or system components (e.g., a gripping arm). The level of danger or the associated risk of damage depends on the speed and mass of the system and the complexity and variability of its environment (Environmental Complexity). The following examples illustrate this:

- With a semi-autonomous, automatic lawn mower, the area to be mown is bordered, for example, by a signal wire. The movement space garden is a controlled environment. The robot's movement speed and movement energy are low. Contact-based collision detection is sufficient for obstacle detection. The risk posed by the rotating cutting knife is protected to an acceptable level (for operation within the controlled environment) by the housing and by sensors which detect lifting of the robot or blocking of the knife.
- For a fully autonomous car, the range of motion is open. Motion speed and kinetic energy can be very high. The car moves simultaneously to many other road users in a confined space. Obstacles of any kind can "appear" in the route at any time. Evasion is a necessary part of "normal operation." For safe driving in compliance

with traffic regulations, extremely reliable, fast, predictive obstacle detection is required.

When a robot interacts with objects, damage can also be caused indirectly (in addition to the danger of damaging the object or robot). The following examples from [10, p.77] illustrate this:

- A service robot is instructed to bring the dishes to the kitchen sink. In order to deposit the dishes near to the sink, it recognizes the modern ceramic stove top as preferable surface and deposits the dishes there ... If now a cooking plate is still hot, and there is, for instance, a plastic salad bowl, or a cutting board amongst the dishes, obviously, some risks arise. The situation in which a plastic or wooden object is located very close or on top of the cooking plate can be considered as not safe anymore, since the risk of toxic vapor or fire by inflamed plastic or wood is potentially present.

  The worst case accident can be a residential fire causing human injury or death. The risk is not present in a situation in which these objects are located apart the cooking plate (with a certain safety margin), independent from the state of the cooking plate.
- A service robot is instructed to "watering the plants." In this connection, it is assumed that a power plug fell into a plant pot ... If the robot is watering the plant, the risk of electrical shock arises, both, for human and robot. The risk factors can be considered to be the following: The object recognition again recognizes the power plug while having the watering can grasped (or any plant watering device) and additionally, it can be detected that there is water in the watering can (or similar device). In consequence, a rule should be integrated that instructs the robot not to approaching too close with the watering can to a power plug, or the like, in order to avoid that it is struck by a water jet.

In order to be functionally safe, a highly or fully autonomous system must therefore have appropriate capabilities and strategies to identify situations as potentially dangerous and then respond appropriately to the situation in order to avoid imminent danger or minimize[13] damage as far as possible. The examples *cooking plate* and *watering the plants* make it clear that pure obstacle detection alone is not always sufficient. In complex operational environments with complex possible missions of the autonomous system, some dangers can only be recognized if a certain "understanding" of cause-effect relationships is given.

Such capabilities and strategies must be part of the "intelligence" of highly autonomous systems. The intended system functionality and the necessary safety functions cannot be implemented separately, but are two sides of the same coin.

3.2 Safety in Failure Mode

If parts of the autonomous system fail, become damaged, or do not function as intended (because of hardware faults, such as contamination or defect of a sensor),

[13]The media in this context mainly discuss variants of the so-called "trolley problem", that is, the question of whether and how an intelligent vehicle should weigh the injury or death of one person or group of persons at the expense of another person or group of persons in order to minimize the consequences of an unavoidable accident (see [11]).

the danger that the system causes damage is naturally even greater than in normal operation.

If a (rare) environmental situation occurs that is "not intended" by the software or that causes a software defect that has hitherto remained undetected in the system to take effect, this can transform an inherently harmless situation into a dangerous one and/or render existing safety functions ineffective.

With conventional, nonautonomous safety-critical systems, sufficiently safe behavior can usually be achieved by a "fail-safe" strategy. This means that the system is designed in such a way that in the event of a technical fault, the system is switched off or its operation is stopped, thereby greatly reducing or eliminating immediate danger (to the user or the environment).

This approach is not sufficient for autonomous systems! If a self-driving car would stop "in the middle of the road" in the event of a failure of an important sensor, the car would increase the danger it poses instead of reducing it. Autonomous systems should therefore have appropriate "fail-operational" capabilities (see [12]). A self-driving car should act as a human driver would: pilot to the side of the road, park there, and notify the breakdown service.

4 Testing Autonomous Systems

In which points does the testing of autonomous systems differ from the testing of software-based systems of today's character? To answer this, we consider the following subquestions:

- Which test topics need to be covered?
- What new testing methods are needed?
- Which requirements for the test process become more stringent?

4.1 Quality Characteristics and Test Topics

The objective of testing is to create confidence that a product meets the requirements of its stakeholders (customers, manufacturers, legislator, etc.). "Those stakeholders' needs (functionality, performance, security, maintainability, etc.) are precisely what is represented in the quality model, which categorizes the product quality into characteristics and sub-characteristics." [13]. This ISO 25010 [13] product quality model distinguishes between the following eight quality characteristics: Functional Suitability, Performance Efficiency, Compatibility, Usability, Reliability, Security, Maintainability, and Portability.

These quality characteristics can be used as a starting point when creating a test plan or test case catalog for testing an autonomous system. Within each of these quality characteristics, of course, it must be analyzed individually which specific

requirements the system to be tested should meet and what should therefore be checked in detail by test cases.

Utilizing this approach a test plan for the mobile robot "Mobipick" [14] of the German Research Center for Artificial Intelligence (DFKI) was created in 2018 as part of a cooperation project between imbus AG and DFKI. The test contents were recorded in the cloud-based test management system [15] and made available to the DFKI scientists and the project team. The following list references this case study to illustrate which topics and questions need to be considered when testing an autonomous system:

- *Functional Suitability*: It must be checked if the functional properties of the system are implemented "complete," "correct," and "appropriate." The functions of each individual component of the system are affected (at lower levels). At the highest level, the ability of the overall system to complete its mission shall be tested. The "Mobipick" test cases for example focuses on the functions "Navigation" and "Grabbing" and the resulting mission pattern: approach an object at a destination, grab it, pick it up and put it down at another location. Testing the functionality also must include testing the system's load limits! A restriction for gripping could be, for example, that the robot tips over in the case of heavy objects or is deflected from its direction of travel. Such boundary cases and the system behavior in such boundary cases must also be considered and examined.
- *Performance Efficiency*: The time behavior of the system and its components and the consumption of resources must be checked.
  - Possible questions regarding time behavior are: is the exercise of a function (e.g., obstacle detection) or mission (object approach, grab, and pick up) expected in a certain time period or with a certain (min/max) speed?
  - Possible tests regarding resource consumption (e.g., battery power) are: run longest application scenario on full battery to check range of battery; start mission on low battery to check out of energy behavior; start mission on low battery at different distances to charging station to check station location and estimate power consumption to station.
- *Compatibility*: This concerns the interoperability between components of the system itself (sensors, controls, actuators) as well as compatibility with external systems. Possible questions are: Can the control software, which was brought to the robot initially or after an update, take over sensor data, process it and control actuators correctly? Are the protocols for communication compatible between robot components or with external systems?
- *Usability*: What possibilities does the user have to operate the robot or to communicate with it? How are orders given to the robot? Are there any feedback messages in the event of operating errors and failure to understand the command? How does the robot communicate its status? Which channels and media are used for transmission: via touch panel on the robot, via app over WLAN, or via voice control? This also includes the handling of objects: can the robot hand over a gripped object to its user precisely enough?

- *Reliability*: Reliability is the ability of the system to maintain its once achieved quality level under certain conditions over a fixed period of time. Test topics can be: Can the robot repeat a behavior several times in a row without errors, or do joints misalign in continuous operation? Can the robot tolerate/compensate (hardware) errors to a certain degree?
- *Security*: To check how resistant the system is against unwanted access or criminal attack on data of the system or its users or on the entire system itself. Questions can be:
  - Does the operator need a password to switch on? How secure is this? With autonomous robots such as "Mobipick," the highest security risk arises from the control mode. The easier it is to manipulate the commands given to the system, the easier it is to (maliciously) take over or shut down the system. Is the robot operated via WLAN/radio? Is the data exchange with the system and within the system encrypted? Can third parties read along, possibly latch into the data traffic and manipulate or even take over the system? The unauthorized takeover of an autonomous system can have serious consequences, in extreme cases its use as a weapon. Therefore, security features are always safety-relevant features!
  - In order to be able to clarify liability issues in the event of an accident, legislators already require autonomous vehicles to record usage data during operation. In Germany these must be kept available for 6 months (see [16]). Similar requirements are expected for other autonomous systems. The GDPR-compliant data security of the system, but also associated (cloud based) accounting or management systems, is therefore another important issue.
- *Maintainability*: A good maintainability is given if software and hardware are modular and the respective components are reusable and easily changeable. Questions in this context are: how are dependencies between software and hardware managed? Does the software recognize which hardware it needs? How do the update mechanisms work? Is it defined which regression tests are to be performed after changes?
- *Portability*: At first glance, the software of robots can be transferred to other robot types to a very limited extent because it is strongly adapted to the specific conditions of the hardware platform and the respective firmware.
  - Individual software components (e.g., for navigation), on the other hand, are generic or based on libraries. It must be tested whether the libraries used in the concrete robot (e.g., "Mobipick") actually work faultlessly on this specific platform.
  - The autonomous system itself can also be "ported" or modified for use in other (than originally intended) environments. For example, by installing additional sensors and associated evaluation software.

The examples show how complex and time-consuming the testing of an autonomous system can be. An important finding is:

- "Functional safety" is not just a sub-item of "Functional Suitability"! Each of the eight quality characteristics from ISO 25010 [13] contains aspects which (especially if there are weaknesses) influence whether the system can be assessed as "functional safe." This is particularly true for the topic "Security."

4.2 Implications of Learning

The intelligence of highly autonomous systems will largely be based on learning algorithms (machine learning). Learning will not only be limited to the development phase of a system (learning system). From a certain Mission Complexity and Environmental Complexity on, it will be necessary for autonomous systems to learn from data they collect during normal operation (self-learning system) and thus continuously improve their behavior or adapt it for rare situations. This poses completely new questions to the development, testing, and approval of such systems:

> If robots are required to be able to learn, this reveals additional questions with regard to the problem to ensure safe robot behavior. Learning capabilities implicate that the learning system is changed by the learning process. Hence, the system behavior is not anymore determined by its initial (designed) structure, and not only structure deviations due to occurring faults are of interest anymore. Learning changes the systems structure; thus, its behavior can as well be determined by the newly learned aspects. The residual incompleteness of the safety-related knowledge consequence is that the system differs from its initially designed version. [10, p.131]

> The testing branch is facing new questions: how to test that a system is learning the right thing? How do test cases, which check that certain facts have been learned correctly, look like? How to test that a system correctly processes the learned knowledge by forgetting for example wrong or obsolete information or abstracting other information? How to test that (for example with robot cars) self-learning software follows specific ethic rules? How to formulate test strategies and test cases in such a way that they can handle the "fuzziness" of the behavior of AI systems? [17]

> With regard to the introduction of self-learning systems, the protection of users' physical integrity must be a top priority … As long as there is no sufficient certainty that self-learning systems can correctly assess these situations or comply with safety requirements, decoupling of self-learning systems from safety-critical functions should be prescribed. The use of self-learning systems is therefore conceivable with the current state of the art only for functions that are not directly relevant to safety. [4]

4.3 New Test Method: Scenario-Based Testing

An autonomous system is characterized by the fact that it is capable of independently heading for and achieving a given mission goal. The subtasks that the system must solve for this can be formulated as test tasks and look as follows:

- Sensing: Can the system capture the signals and data relevant to its mission and occurring in its environment?

- Perceiving: Can it recognize patterns or situations based on signals and data?
- Analyzing: Can it identify options for action appropriate to the respective situation?
- Planning: Can it select the appropriate or best options for action?
- Acting: Can it implement the chosen action correctly and on time?

The systematic testing of this chain of tasks requires a catalogue of relevant situations that is as comprehensive as possible. These situations must be able to be varied in many parameters (analogous to different equivalence classes when testing classic IT systems): For example, the "Mobipick" service robot should be able to detect a closed door as an obstacle under different lighting conditions (daylight, bright sunlight, at night) and with different door materials (wooden door, glass door, metal door).

It must be possible to link the situations into scenarios (successive situations) in order to bring about specific situations in a targeted manner, in order to be able to examine alternative paths of action, but also in order to be able to examine the development over time for a specific situation and the timely, forward-looking action of the autonomous system.

Such testing of the behavior of a system in a sequence of situations is referred to as "Scenario-based Testing." [4] proposes "... to transfer relevant scenarios to a central scenario catalogue of a neutral authority in order to create corresponding generally valid specifications, including any acceptance tests.". The standardization of formats for the exchange of such scenarios is being worked on. ASAM OpenSCENARIO "... defines a file format for the description of the dynamic content of driving and traffic simulators .... The standard describes vehicle maneuvers in a storyboard, which is subdivided in stories, acts and sequences." [18].

Scenario-based testing requires that the same test procedure is repeated in a large number of variations of the test environment. When testing classic software or IT systems, however, the test environment is constant or limited to a few predefined variants. If the IT system successfully passes its tests in these environments, it can be considered suitable for use with low or acceptable risk.

If a robot or a self-driving car passes its tests in only one or a few test environments, the system may still be totally unsuitable for real operation, or even pose an extreme safety risk. When testing autonomous systems, the systematic variation of the test environment is therefore an essential and decisive part of the test strategy.

4.4 Requirements for the Test Process

The combination of "complex cyber-physical system" with "Mission Complexity" and "Environmental Complexity" leads to an astronomical number of potentially testable scenarios. Each of these scenarios, in turn, consists of situation sequences, with the possibility of variation in the respective system status, the environmental situation and the potential options for action of the system. Since safety requirements are not an isolated "subchapter of the test plan," but are present throughout all

scenarios, it is difficult and risky to reduce testing effort by prioritizing and omitting scenarios.

Testing only one such scenario in reality can require enormous effort (a secure test site is required, and changing the test setup and the subsequent repeated test drives in that site requires a lot of effort and time). A very large proportion of the necessary tests must and will therefore be carried out in the form of simulations.

Nevertheless, some of the scenarios will always have to take place additionally in reality. Because simulations can be error-prone and they usually will not be physically complete.

An important measure to gain time and safety is a consistent shift-left of tests to the lowest possible test levels and continuous testing during development at all test levels in parallel: at the level of each individual component, for each subsystem, and at the system level. Test-driven development and the formal verification of safety-critical components will play an increasingly important role. Continuous monitoring of the systems in operation ("shift-right") and, if necessary, quick reaction to problems in the field, will also be indispensable. In the *Ethics Guidelines for Trustworthy AI* of the European Commission corresponding demands are clearly formulated: "Testing and validation of the system should occur as early as possible, ensuring that the system behaves as intended throughout its entire life cycle and especially after deployment. It should include all components of an AI system, including data, pre-trained models, environments and the behaviour of the system as a whole." [3].

The test contents and test results of all test levels and the data from fleet operation must be continuously monitored, evaluated, and checked by test management in order to be able to identify gaps in the test coverage but also to reduce redundancies.

Significantly increased importance will be attached to testing by independent third parties. Here, too, [3] formulates proposals: "The testing processes should be designed and performed by an as diverse group of people as possible. Multiple metrics should be developed to cover the categories that are being tested for different perspectives. Adversarial testing by trusted and diverse 'red teams' deliberately attempting to 'break' the system to find vulnerabilities, and 'bug bounties' that incentivise outsiders to detect and responsibly report system errors and weaknesses, can be considered.".

5 Conclusion and Outlook

Procedures and best practices from the testing of classical software and IT systems, as well as from the field of conventional, safety-critical systems or vehicle components,[14] are also still valid for the testing of autonomous systems.

[14]ISO 26262:2018, "Road vehicles - Functional safety," is the ISO series of standards for safety-related electrical/electronic systems in motor vehicles.

A central question is how functional safety of autonomous systems can be guaranteed and tested. The intended system functionality and the necessary safety functions cannot be implemented separately, but are two sides of the same coin. Accordingly, it is not possible to separate the aspects of functionality and safety during testing.

Manufacturers of autonomous systems need procedures and tools by means of which they can test the functionality and safety of such products seamlessly, but nevertheless with economically justifiable effort, and prove them to the approval authorities.

One approach is Scenario-Based Testing. Scenarios can be used to model and describe usage situations and mission processes of an autonomous system. These scenarios can then be used as test instructions for testing in simulations or in reality.

In addition to the standardization of scenario formats or scenario languages, tools are needed to capture and manage scenarios. Integrations between such scenario editors, simulation tools, test benches, and test management tools need to be developed. Such tools or tool chains should also help to create scenario variants systematically and to evaluate scenarios and tests automatically, for example, with regard to safety relevance and achieved test coverage.

References[15]

1. Gartner “Top 10 Strategic Technology Trends for 2019: Autonomous Things”, Brian Burke, David Cearley, 13 March 2019
2. Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles, SAE - On-Road Automated Driving (ORAD) committee, https://saemobilus.sae.org/content/J3016_201806/
3. Independent High-Level Expert Group on Artificial Intelligence – set up by the European Commision, Ethics Guidelines for Trustworthy AI, 04/2019. https://ec.europa.eu/futurium/en/ai-alliance-consultation/guidelines
4. Automatisiertes und Vernetztes Fahren - Bericht-der-Ethik-Kommission, Bundesministerium für Verkehr und digitale Infrastruktur. https://www.bmvi.de/SharedDocs/DE/Publikationen/DG/bericht-der-ethik-kommission.html (2017)
5. Autonomy Levels for Unmanned Systems (ALFUS) Framework Volume I: Terminology, Version 2.0, Autonomy Levels for Unmanned Systems (ALFUS) Framework Volume II: Framework Models Version 1.0, https://www.nist.gov/el/intelligent-systems-division-73500/cognition-and-collaboration-systems/autonomy-levels-unmanned
6. https://en.wikipedia.org/wiki/Autonomous_things
7. https://de.wikipedia.org/wiki/SAE_J3016
8. IEC 61508:2010, Functional safety of electrical/electronic/programmable electronic safety-related systems - Parts 1 to 7, https://www.iec.ch/functionalsafety/
9. IEC 61508 Explained, https://www.iec.ch/functionalsafety/explained/
10. Safety of Autonomous Cognitive-oriented Robots, Philipp Ertle, Dissertation, Fakultät für Ingenieurwissenschaften, Abteilung Maschinenbau der Universität Duisburg-Essen (2013)

[15]The validity of the given URLs refers to July 2019.

11. Eimler, S., Geisler, S., Mischewski, P., Ethik im autonomen Fahrzeug: Zum menschlichen Verhalten in drohenden Unfallsituationen, Hochschule Ruhr West, veröffentlicht durch die Gesellschaft für Informatik e. V. 2018 in R. Dachselt, G. Weber (Hrsg.): Mensch und Computer 2018 – Workshopband, 02.–05. September 2018, Dresden
12. Temple, C., Vilela, A.: Fehlertolerante Systeme im Fahrzeug – von "fail-safe" zu "fail-operational". https://www.elektroniknet.de/elektronik-automotive/assistenzsysteme/fehlertolerante-systeme-im-fahrzeug-von-fail-safe-zu-fail-operational-110612.html (2014)
13. https://iso25000.com/index.php/en/iso-25000-standards/iso-25010
14. Deutsches Forschungszentrum für Künstliche Intelligenz GmbH (DFKI), Robotics Innovation Center, Robotersystem Mobipick. https://robotik.dfki-bremen.de/de/forschung/robotersysteme/mobipick.html
15. Cloudbasierten Testmanagementsystem der imbus AG. https://www.testbench.com
16. Deutscher Bundestag, Straßenverkehrsgesetz für automatisiertes Fahren, Drucksache 18/11776 vom 29.03.2017. https://www.bundestag.de/dokumente/textarchiv/2017/kw13-de-automatisiertes-fahren-499928, http://dip21.bundestag.de/dip21/btd/18/113/1811300.pdf
17. Flessner, B.: The Future of Testing, imbus Trend Study, 3rd edition. https://www.imbus.de/downloads/ (2017)
18. ASAM OpenSCENARIO. https://www.asam.net/standards/detail/opensce nario/

Further Reading

[ISO 10218:2011-07] Robots and robotic devices - Safety requirements for industrial robots, Part 1: Robots, Part 2: Robot systems and integration
[ISO 12100:2010] Safety of machinery - General principles for design - Risk assessment and risk reduction
[ISO 13482:2014] Robots and robotic devices - Safety requirements for personal care robots
[ISO 8373:2012-03] Robots and robotic devices – Vocabulary

Testing in the Digital Age

Rik Marselis

Abstract How do you test a robot? Is testing an intelligent machine different from testing traditional IT systems? Can we use a robot to make testing easier? Will artificial intelligence take over all testing activities?

These and many more questions may come to mind in the current digital age. When reading about the tremendously fast-moving developments that are currently happening in information technology (e.g., computers) and operational technology (e.g., factories), with artificial intelligence, machine learning, robots, chatbots, self-driving cars, autonomous vacuum cleaners, and many other intelligent machines.

Keywords Software testing · Software quality · Artificial intelligence · Intelligent machines · Test automation

1 Testing *Of* and *With* Intelligent Machines

How do you test a robot? Is testing an intelligent machine different from testing traditional IT systems? Can we use a robot to make testing easier? Will artificial intelligence take over all testing activities?

These and many more questions may come to mind in the current digital age. When reading about the tremendously fast-moving developments that are currently happening in information technology (e.g., computers) and operational technology (e.g., factories), with artificial intelligence, machine learning, robots, chatbots, self-driving cars, autonomous vacuum cleaners, and many other intelligent machines.

In the digital age we prefer not to talk about the function of test engineer but about test engineering as a set of skills and activities that can be performed by people with many different roles. The skills needed to set up digital test engineering are numerous. It is not feasible that all skills are gathered in one person. Teams of digital test experts work together to set up, for example, a test automation system

R. Marselis
Sogeti, Amsterdam, Netherlands

S. Goericke (ed.), *The Future of Software Quality Assurance*,
https://doi.org/10.1007/978-3-030-29509-7_6

that can do continuous delivery of AI systems. The classical test engineer has to evolve and incorporate new skills like data analysis, AI algorithms, or (as we will see at the end of this chapter) weather forecasting.

In this chapter, we will elaborate first on the testing *of* intelligent machines. After that we will focus on testing *with* intelligent machines, which means the use of intelligent machines to support testing.

2 Testing *Of* Intelligent Machines

Artificial intelligence can (and should) be tested. In this chapter, we talk about the testing *of* AI.

Since AI solutions are quite new, experience in this field is scarce. Testing of AI has to formulate and evaluate complete and strong acceptance criteria that verify the outcome. The outcome is determined by the input data and a trained model. Testing those is the core activity. The quality of cognitive IT systems that use artificial intelligence needs to be assessed. The challenge in this case is in the fact that a learning system will change its behavior over time. Predicting the outcome isn't easy because what's correct today may be different from the outcome of tomorrow that is also correct. Skills that a tester will need for this situation are related to interpreting a system's boundaries or tolerances. There are always certain boundaries within which the output must fall. To make sure the system stays within these boundaries, the testers not only look at output but also at the system's input. Because by limiting the input we can influence the output.

2.1 Six Angles of Quality for Machine Intelligence

People have been assessing the quality of things for centuries. Since the invention of the steam engine the need for a structured approach to quality assessment rapidly grew. After the creation of the first computers in the 1940s people realized that these "decision-making and data-processing" machines again needed a new angle to quality, and the first approaches to testing were published. Nowadays, we have test methods such as TMap, approaches (e.g., exploratory testing) and techniques (e.g., boundary value analysis) to establish the quality of IT processes and to build confidence that business success will be achieved.

Different angles can be used to assess quality. Some angles are known from a traditional need for quality (think mechanical or electrical). These angles are brought to life again because the digital age brings new technologies (e.g., 3D printing in the mechanical world).

The quality approach for a machine intelligence solution must address the following six angles of quality shown in Fig. 1.

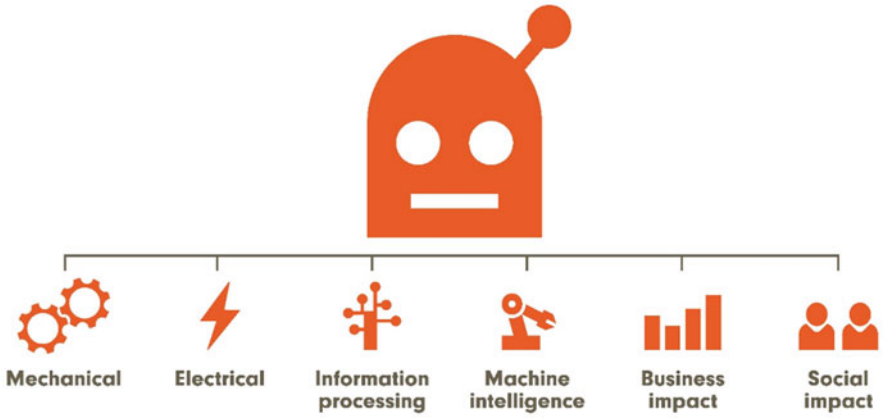


Fig. 1 The six angles of quality in the digital age

Mechanical
In the era of steam engines, for example, the "smoke test" was invented to check, by slowly increasing the pressure in the steam boiler, whether smoke was leaking somewhere from the machine.

With 3D printing, the mechanical angle gets new attention. When replacement parts are printed 3D, they may have different material characteristics. Strength and temperature distribution are quite different when doing 3D printing (also called additive manufacturing) opposed to known methods such as casting with plastics.

Electrical
With the invention of electricity, a new angle to testing was added because not just mechanical things needed to be tested but also the less tangible new phenomenon of electricity. In the digital age we see new electrical phenomena such as wireless transfer of energy to charge batteries and wireless communication using a wide variety of protocols and techniques.

Information Processing
Since the early computers of the 1950s and 1960s, people have built techniques and methods for testing software. The Year 2000 problem boosted the creation and use of testing methods and frameworks. TMap dates back to 1995 and gives a clear approach to testing complex IT systems.

With the rise of machine intelligence and robotics, testing methods are again used to test basic information processing components of intelligent machines.

Machine Intelligence
And now we see the rapid rise of machine intelligence which brings new challenges to test engineering professionals. Both the intelligence part (especially the learning) and the machine part (robots) require this additional angle to quality. These intelligent machines bring a new challenge in the world around us: the impact of

machine intelligence is way beyond the impact that previous technologies had on our businesses and our society. Testing of machine intelligence is different from traditional testing of information processing. With traditional IT, the tester could always predict the outcome of a test since the rules were defined upfront. In complex systems, it may be a tough task but fundamentally it's always possible to define the expected outcome.

But now we have learning machines. Based on the input they gather using all sorts of sensors, they pick up information from their environment and based on that determine the best possible result at that given point in time. By definition, the result at another point in time will be different. That's a challenge during test design and test execution. For example, this requires that testers work with tolerances and upper and lower boundaries for acceptable results.

Business Impact
New technologies have always had impact on businesses. But the very fast evolution and the impressive possibilities of the implementation of intelligent machines in business processes require special attention for the business impact.

As quality-minded people, we want to make sure that machine intelligence positively contributes to business results. To ensure this, IT teams need to use both well-known and brand-new quality characteristics to evaluate whether the new technology contributes to the business value.

Social Impact
Until now, machines always supported people by extending or replacing muscle power. Now we see machines that replace or extend human brain power. This may have tremendous consequences for the way people interact and for society in a broader sense, which brings us to the last angle of quality: social impact. New technologies have had social impact. So, what's new this time? It's the speed with which this technology enters our lives. And the possibility of machines taking decisions for people, even to the extent where people don't understand the reasoning behind the choice for a specific option.

2.2 Quality Is Defined and Tested Using Quality Characteristics

To get a clear view on the quality level of any system, we need to distinguish some subdivision of quality. Therefore, we use quality characteristics. The long-known standards evolved in an era when IT systems were focused on data processing and where input and output were done by means of files or screen-user-interfaces. We use the ISO25010 standard as a basis for our elaboration. In Figure 2 you see the traditional ISO25010 quality characteristics in gray. In red you see the quality characteristics for intelligent machines that we have added (Fig. 2).

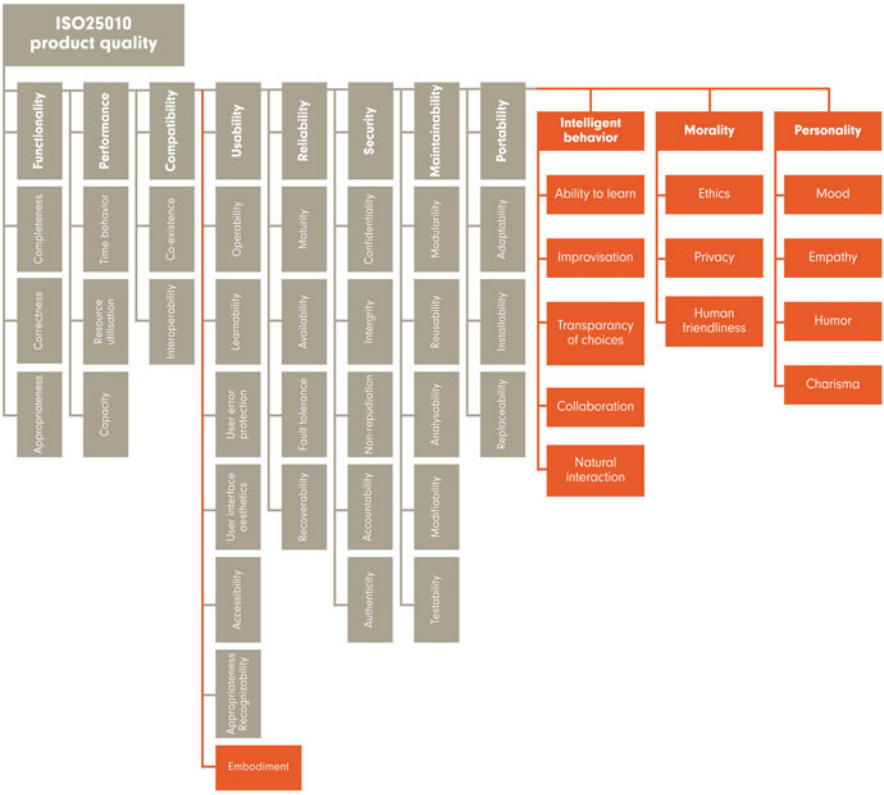


Fig. 2 The ISO25010 standard with quality characteristics in gray and our extension with quality characteristics for intelligent machines in red

Nowadays, we see machine intelligence systems that have many more options. Input is often gathered using sensors (e.g., in IoT devices) and output may be physical (like moving objects in a warehouse). This calls for an extension of the list of quality characteristics. The next sections of this chapter describe these new quality characteristics.

We have added three new groups of quality characteristics, namely, Intelligent Behavior, Personality, and Morality. In their respective sections, we describe these main characteristics and their subcharacteristics. After that we describe the subcharacteristics of embodiment that we added to the existing main characteristic Usability.

2.2.1 Intelligent Behavior

Intelligent behavior is the ability to comprehend or understand. It is basically a combination of reasoning, memory, imagination, and judgment; each of these

faculties relies upon the others. Intelligence is a combination of cognitive skills and knowledge made evident by behaviors that are adaptive.

Ability to Learn
The ability to learn is the ability to comprehend, to understand, and to profit from experience. How does an intelligent machine learn?

We see three levels of learning. The first level is rule-based learning. If a user uses certain options in a menu most frequently, the intelligent machine can order the options such that the most used options are on top. The second level is based on gathering and interpreting data, and based on that learning about an environment. The third level is learning by observing behavior of others and imitating that behavior.

Examples of the levels of learning:

- At the first level of learning, think of a satellite navigation system in a car, if you always turn off the automatic voice right after starting the system, the machine learns to start up without the voice activated.
- At the second level think of a robotic vacuum cleaner, by recording information about the layout it learns about the rooms that it cleans and therefore becomes better at avoiding obstacles and reaching difficult spots.
- At the third level it's about mimicking behavior, for example, a robot watches a YouTube video of baking pancakes and then copies the behavior. After watching several videos, the robot knows all the tricks of the trade.

Of course, the levels of learning can be combined by intelligent machines.

Improvisation
Does it adapt to new situations? Improvisation is the power of the intelligent system to make right decisions in new situations. Such situations it has never experienced before require quick interpretation of new information and adjusting already existing behavior. Social robots must especially be able to adapt their behavior according to the information coming in, since social behavior depends on culture in specific small groups. Applying long-term changes will also be important for a robot to remain interesting or relevant for its environment.

Transparency of Choices
Can a human involved understand how a machine comes to its decisions? An artificial intelligence system works 24/7 and it takes a lot of decisions. Therefore, it has to be transparent how an AI system takes decisions, based on which data inputs. And which data points are relevant and how are they weighted? In several use-cases, the decision-making is crucial. For example, when an Artificial Intelligent system calculates the insurance premium, it is important to investigate how this premium is calculated. Transparency also means predictability. It is important that robots respond as expected by the people who work with the robot. How well can the people involved foresee what (kind of) action the intelligent machine will take in a given situation? This is the basis for proper collaboration (see next paragraph).

To comply with rules for transparency of choices an organization may choose to apply "explainable AI" (also known as XAI).

Collaboration/Working in a Team
How well does the robot work alongside humans? Does it understand expected and unexpected behavior of humans? Robots can work together with people or other robots in a team. How communication works within this team is very important. A robot must be aware of the people around and know when a person wants to interact with the robot. With the help of natural interaction, the robot must make it possible to draw attention to itself. Working in a team is very important in industrial automation where robots and people work alongside each other in a factory, but also in traffic where, for example, a bicyclist should be able to see whether a self-driving car has seen the bicyclist wants to make a turn.

Collaboration between robots only, without humans involved, is very similar to the existing quality characteristic interoperability, but because collaboration can be of great importance in robots and intelligent systems, we would like to mention it separately.

Natural Interaction
Natural interaction is important both in verbal and nonverbal communication. Especially with social robots it is important that the way people interact with a robot is natural, like they interact with people. One of the things that can be watched here is multiple input modalities, so that there is more than one possibility to control the robot (e.g., speech and gestures).

In chatbots it is important that the conversation is natural, but also specific to the purpose of the chatbot. You can imagine that a chatbot making small talk has more room to make mistakes and learn slowly than a chatbot that is supposed to make travel arrangements should clearly understand destination, dates, and other relevant information without erroneous interpretations. Most people that enter "home" as their destination mean their own home and not the nearest nursing home, which a traditional search-engine would assume. In this case, asking clarification is very important for the chatbot.

2.2.2 Morality

Morality is about the principles concerning the distinction between right and wrong or good and bad behavior. A very well-known science fiction author who gave a great deal of thought to the morality of intelligent machines is Isaac Asimov. One of his contributions was drawing up the "laws of robotics" that intelligent machines should adhere to.

The classic Asimov laws of robotics are:

- law 0: A robot may not harm humanity, or, by inaction, allow humanity to come to harm.

- law 1: A robot may not injure a human being or, through inaction, allow a human being to come to harm.
- law 2: A robot must obey the orders given to it by human beings except where such orders would conflict with the First Law.
- law 3: A robot must protect its own existence as long as such protection does not conflict with the First or Second Law.

Other sources added some additional laws:

- law 4: A robot must establish its identity as a robot in all cases.
- law 5: A robot must know it is a robot.
- law 6: A robot must reproduce. As long as such reproduction does not interfere with the First or Second or Third Law.

Unfortunately, we observe that, unlike in Asimov's stories, these robot laws are not built in in most intelligent machines. It's up to the team members with a digital test engineering role to assess to what level the intelligent machine adheres to these laws.

Ethics

Ethics is about acting according to various principles. Important principles are laws, rules, and regulations, but for ethics the unwritten moral values are the most important. Some challenges of machine ethics are much like many other challenges involved in designing machines. Designing a robot arm to avoid crushing stray humans is no more morally fraught than designing a flame-retardant sofa.

With respect to intelligent machines important questions related to ethics are:

- Does it observe common ethical rules?
- Does it cheat?
- Does it distinguish between what is allowed and what is not allowed?

To be ethically responsible, the intelligent machine should inform its users about the data that is in the system and what this data is used for.

Ethics will cause various challenges. For example: it isn't too difficult to have an AI learn (using machine learning) to distinguish people based on facial or other body part characteristics, for example, race and sexual preference. In most countries, this would not be ethical. So testers need to have acceptance criteria for this and do something with it.

Another ethical dilemma is who is responsible when an intelligent machine causes an accident? There is no driver in the car, just passengers. Should the programmer of the intelligent software be responsible? Or the salesman that sold the car? Or the manufacturer? All ethical (and some legal) dilemma's. And who should be protected in case of an autonomous car crash? Some manufacturers of autonomous cars have already announced that their cars will always protect the people inside the car. That may be smart from a business point of view (otherwise no one would buy the car) but from an ethical perspective, is it right to let a few passengers in the car prevail over a large group of pedestrians outside the car?

Finally, an ethical dilemma is about feelings of people towards the intelligent machine. The 2013 Oscar-winning film *Her* shows how a man (actor Joaquin Phoenix) falls in love with his operating system. From an ethical point of view, we may wonder if we should allow a machine to acknowledge and return such feelings.

Privacy

Privacy is the state of being free from unwanted or undue intrusion or disturbance in one's private life or affairs. Does the intelligent machine comply with privacy laws and regulations? The fuel of machine learning algorithms are data. They determine what the solution can and will do in the end. It has to be ensured that the gathered data and the insights gained from that data are aligned with the business goals. There are also legal constraints, which depend on national and international laws, regulations, and the analyzed data. Especially the EU has with the GDPR one of the strictest regulations with the ability for severe financial sanctions.

Data breaches occur. That's a fact. This gives hackers access to sensitive security information, such as that contained in email attachments, which should not be there in the first place. Privacy considerations now have a bigger scale and impact. They should be handled carefully, not only because of the social responsibility, but because legislation, like GDPR, must be complied with. Reference: there is an ISO standard for privacy: ISO 29100.

Human Friendliness

Human friendliness refers to the level to which intelligent machines don't cause harm to humans or humanity. Today, often the term "beneficial AI" is used in discussions about the attitude of Artificial General Intelligence towards human society.

Most of the leading experts and companies in AI see the risk that AI and robotics can be used in warfare. This does not only challenge our current ethical norm, but also our instinct of self-preservation.

The Future of Life Institute is taking a close look at these dangers. They are real and should be considered when developing new solutions.

Safety and security (especially in cobotics) are often confused, but they are not the same. Security is the protection of the application against people (or machines) with bad intention. This is something other than safety that guarantees no harm comes to people. For robots this is very important since a coworker may want to know: how big is the chance that I will get a big iron arm against my head if I try to communicate with this robot?

It is often thought that robots and other intelligent machines will take over all labor from people. The fear that machines will take over all labor from people has been expressed many times in the last couple of centuries. And indeed, some human labor has been automated. But every time new challenging tasks came in return.

This time it won't be different. But a specific phenomenon will be "backshoring." In recent years lots of work has been offshored to countries where the hourly wages are lowest. Nowadays, robots can do work even cheaper, and 24/7. Therefore, transport costs will be the determining factor. And consequently, work is "backshored" to the place where the result of the work is wanted. Which also brings some highly

skilled support work (to organize this work, be creative about new possibilities, etc.). In this sense, AI is human friendly because the work is evenly spread over the globe, based on the location where the results are needed.

2.2.3 Personality

A personality is the combination of characteristics or qualities that form an individual's distinctive character. Let's focus on having robots as a partner or assistant. We want to build robots with a personality that fits the personality of the humans it collaborates with.

Mood
A mood is a temporary state of mind or feeling.

Will an intelligent machine always be in the same mood? We would be inclined to think that a machine by definition doesn't know about moods, it just performs its task in the same way every time again. But with adding intelligence the machine also may change its behavior in different situations or at different times of day. A good use of moods may be in cobotics, where the robot adapts its behavior to the behavior of the people it collaborates with. For example, at night the robot may try to give as few signals as possible because people tend to be more irritable at night, whereas on a warm and shiny summer day the robot also may communicate more outspoken.

Another aspect of mood is using machine intelligence to change the mood of people. Mood altering or so-called AI-controlled brain implants in humans are under test already. Brain implants can be used to deliver stimulation to specific parts of the brain when required. Experts are working on using specialized algorithms to detect patterns linked to mood disorders. These devices are able to deliver electrical pulses that can supposedly shock the brain into a healthier state. There are hopes that the technology could provide a new way to treat mental illnesses that goes beyond the capabilities of currently available therapies.

Empathy
Empathy is the ability to understand and share the feelings of another. Machines cannot feel empathy, but it is important that they simulate empathy. They should be able to recognize human emotions and respond to them. An intelligent machine should understand the feelings of the people it interacts with. This is especially important with robots working in hospitals as so-called companion robots.

Humor
Humor is the quality of being amusing or comic, especially as expressed in literature or speech. Is there a difference between laughter and humor? Yes, there is. Laughter is used as a communication aid; from the gentle chuckle to the full-on belly laugh, it helps us to convey our response to various social situations. Humor could be defined as the art of being funny, or the ability to find something funny. How will robots detect this very human behavior? That is the next step in AI, programming robots

with the ability to get in on the joke, detect puns, and sarcasm and throw a quick quip back! There is a whole branch of science dedicated to research and development in this area. Scientists in this field are known as Computational Humorists. And they have come a long way; these are just some of the algorithms they have created so far. An example of such an algorithm is SASI.

Charisma
Charisma is the compelling attractiveness or charm that can inspire devotion in others. Do people like the intelligent machine? Do people love the intelligent machine? Is it so nice that they never want to put it away anymore? If a product has this "wow factor" then it is much more likely to be a successful product. So, the charisma of a product is important.

Is charisma a sign of intelligence? It is. It is all learned behavior no matter what factors were employed. To be accepted by users, the robot must appeal in some way to the user. That may be by its looks (see embodiment). But more importantly by its functionality and probably also by its flexibility. One way to keep amazing the user is to continuously learn new things and stay ahead of the expectations of the user.

2.2.4 Usability

In the existing group of quality characteristics, we have added only one extra subcharacteristic, that is Embodiment, in the group Usability.

Of course, other existing quality characteristics and subcharacteristics also are of importance but that's just about a different use or application of the existing.

Embodiment
A big buzzword in artificial intelligence research these days is "embodiment," the idea that intelligence requires a body, or in the case of practicality, a robot. Embodiment theory was brought into artificial intelligence most notably by Rodney Brooks in the 1980s. Brooks showed that robots could be more effective if they "thought" (planned or processed) and perceived as little as possible. The robot's intelligence is geared towards only handling the minimal amount of information necessary to make its behavior be appropriate and/or as desired by its creator. Embodiment simply means: "Does it look right?"

With physical robots, as well as with the user interface of chatbots and even smart speakers, it is very important how they look and how they fit in the space in which they have to operate. An important point here is that the appearance of the robot must match the functions that the robot has. When seeing a robot, people create expectations about the functions of this robot, if the appearance is very beautiful, while the robot does little, people can become disappointed. Another relevant aspect of embodiment is the degree to which a robot resembles a human. In general, people like humanoid robots but as soon as they look too real, people start to feel uncanny. In the graph depicting how people like the embodiment, this is known as the uncanny valley. The quality characteristic of embodiment includes not only the physical

embodiment of a robot, but also the placement of a robot in the world in which it is located.

3 Testing With Intelligent Machines

The goal of using machine intelligence in testing is not to take people out of the loop. The *goal* is to make testing easier *and* faster, just like with traditional use of tooling to support any *activity*. Also, some tasks that couldn't be done before are now possible by using intelligent machines. So, it is about enablement, effectivity, and efficiency.

The future sees test engineers as quality forecasters that use smart sets of test cases which evolve to the best fit of addressing problem areas for a smart solution in the right variety of situations. Quality forecasting is aimed at being ahead of the test results, to make sure that quality problems are addressed even before any failure occurs for the user. To get to that situation, many preconditions must be fulfilled. Data about changes and tests thereof must be gathered in a structured way, models are used to describe the system, AI must be used to analyze this data.

Digital test engineering evolves over time. Forecasting technology coming our way helps us to be ready to find the right tools, roles, and skills that keep guarding the quality asked of future products.

In the digital age, new technology is extending human possibilities, new ways of working, new thoughts, and takes on existing products. Where new things are created, things are tried and tested. This also applies to using intelligent machines for testing. The common denominators with all digital terminology are:

- Speed: Extremely fast market response.
- Data: Huge amounts of data are collected.
- Integration: Everyone needs to integrate with everything.

Extremely high speed, huge amounts of data, and an infinite amount of possibilities for integration are the elements we are facing for testing. They extend beyond our human capabilities.

One way to help us out is test automation. Test automation in the context of digital testing, is automating everything possible in order to speed up the complete product development cycle using all means possible; even a physical robot taking over human test activities, such as pushing buttons, is a possibility. Further help can be found in combining it with another new technology not yet mentioned: artificial intelligence (AI). AI works with huge amounts of data, finds smart ways through infinite possibilities, and has the potential to hand us solutions quickly. Let us not forget that AI needs to be tested as well, but AI can make the difference here.

Technologies like artificial intelligence can help us in testing, for example, IoT solutions. Specifically, an evolutionary algorithm can be used to generate test cases. Together with the use of multimodel predictions for test environments, test cases become smart test cases (Fig. 3).

Fig. 3 Testing is moving from a reactive activity (test execution) through quality monitoring for operational systems towards quality forecasting where faults are predicted so that they can be fixed even before anyone notices a failure

The future sees test engineers as quality forecasters that use smart sets of test cases which evolve to the best fit of addressing problem areas for an IT solution in the right variety of situations. Quality forecasting is aimed at being ahead of the test results. Before a situation occurs, digital test engineering already found the underlying defect. However, to get to that situation, a lot of preconditions need to be put into place. Test execution must be organized such that data is gathered in a structured way. Quality monitoring must be organized in a similar way, so that both testing and monitoring data can be used as the basis for quality forecasting.

Digital test engineering evolves over time. Forecasting technology coming our way helps us to be ready to find the right tools, roles, and skills that keep guarding the quality asked of future products.

3.1 Models Help Quality Forecasting

Testing is a reactive activity. Only after you test something, you know whether a product is working correctly or not. There is one constant: you never know the remaining defects still in your product.

Testing must go to the situation where it can predict if and what defects will pop up in the near future. Only then can testing keep ahead of the quality assurance game.

Let us take weather forecast as an example. In order to predict the temperature for the coming 2 weeks, 50 models are calculated. They all give a different solution. From this set of results the most likely path is chosen (by a person!).

This is something that can be done in testing as well. Assume test automation with dynamic models, generating the right test environments and sets of test cases using real-time data from the field, are in place. By manipulating test environments based on real-time data trends, multiple situations can be calculated. This can be done in simulated environments. This makes it quick and eliminates the use of expensive physical test environments that require maintenance as well.

The test engineer now has to go through the set of predicted outcomes and find defects before they occur in the field. In a sense, the test engineer becomes a quality forecaster.

3.2 Robots Testing Physical Machines

Test engineering is often seen as a "software-only" activity. But in the digital age most intelligent systems consist of software and hardware components that together will have to comply with specified quality criteria. The testing of the software part can be supported by AI-powered testing tools. For the hardware part robots can be of great use. There are different reasons to think about robots for testing physical equipment:

Response to Limitation of Software Testing Approach
Automated system tests can be realized by substituting control devices by a software solution. This is possible nowadays, but in a lot of cases it would require working around advanced security and safety mechanisms and therefore preventing companies from testing the product end-to-end in a "customer state." Using robots in the physical environment to work with the physical end product in the end-to-end solution enables testing in an environment close to real life.

Flexibility of the Testing Solution
Customized solutions for each specific control device are used for automated system testing. A more universal approach with lightweight robots and standard components helps to gain speed in terms of including new control devices into the test environment. Just as with GUI test automation, standard building blocks can be used in the physical domain as well. It could even go so far as to recognize physical interfaces (buttons, switches, touch pads, panels, etc.) and have a standard interaction available for the robot to work with.

More Testing and Fewer Costs
With an automated robotic test set in place, it can also be used in production environments. Small lot production can be tested not having to pay for relative long manual test activities executed on each production item. The long-term availability of such test facilities provides long-term product support (even after specific test knowledge has left the company). The test solution can also scale up to large test sets or to large volume test automation activities.

Manage Repeatability of Interaction
In order to compare test results, timing of physical interaction with a system may be mandatory. A human operator is not able to be as precise as is required. Robotics is a good tool to cover this issue.

Reducing Human Risk
Repetitive tasks can be a risk for the health of operators. Dangerous environments are also not a great place to execute tests. Together with robots we can eliminate these elements from test execution, making the test environment for us people a healthier and safer place.

3.3 Test Execution in a Virtual Engineering Environment

A virtual engineering environment provides a user-centered, first-person perspective that enables users to interact with an engineered system naturally. Interaction within the virtual environment should provide an easily understood interface, corresponding with the user's technical background and expertise. In an ideal world, it must enable the user to explore and discover unexpected but critical details about the system's behavior. Exploring and discovering are terms associated with test design techniques like exploratory testing or error guessing. Finding critical details starts with executing a checklist and extends to setting up full test projects in a virtual environment. The checklist to work with for testing in a virtual engineering environment is made up of the elements that make or build up the environment:

- Production and automation engineering
- User-centered product design
- Mechanics of materials
- Physics and mathematics

In the end, testing is about gathering information about achieved benefits and remaining quality risks, information which is used to determine the level of confidence that stakeholders have for using a system. This final judgment about confidence still has to be made by a human, because the accountability for business processes can't be delegated to a machine, no matter how intelligent they may seem to be.

3.4 Beware of Testing AI with AI

A special case is using artificial intelligence to test other artificial intelligence. In some cases, this may be a very appealing possibility. And however valid this option may be, before deciding to do so, the people involved must very carefully weigh the fact that they will be testing a system of which they don't exactly know what it does, by using another system of which they don't exactly know what it does. Because, all in all, this is piling up uncertainties. On the other hand, you may argue that in our modern systems of systems we have long ago become used to trusting systems we don't fully understand.

Measure Twice, Cut Once: Acceptance Testing

Mitko Mitev

Abstract The challenges it sets are numerous and thus, many project managers face the hard decision of how to organize the process that requires not only technical knowledge but also demands to be able to put themselves in the user's shoes. Being aware of the problems that will be met and relying on the testing performed so far, many decide to skip the need for conducting acceptance testing using both external and internal resources. And what happens next—great loss for the company as the end user is dissatisfied with the software and refuses to use it. The company will face not only profit loss, but what is more important in the modern world: loss of reputation and credibility among partners, competitors and above all—"his/her majesty"—the end user, who is now to judge even harshly as ever, because of all the possibilities he/she has and the ease of switching from one software to another.

That is the reason each project manager must be aware of the main challenges of organizing and conducting the acceptance testing. The benefits from it are numerous and will only lead you and your team to a greater success. Of course, it should be performed correctly and planned really well; otherwise you will have to deal with the losses that follow the release of a lousy product in the eyes of the end user.

Keywords Software testing · Software quality · Software tester · Acceptance testing

1 Acceptance Testing, Part 1: Introduction

Often neglected, one of the most important phases in making sure that the software meets the user requirements is acceptance testing.

The challenges it sets are numerous and, thus, many project managers face the hard decision of how to organize the process that requires not only technical knowledge but also demands to be able to put themselves in the user's shoes. Being

M. Mitev
SEETB, Quality House Bulgaria, Sofia, Bulgaria

S. Goericke (ed.), *The Future of Software Quality Assurance*,
https://doi.org/10.1007/978-3-030-29509-7_7

aware of the problems that will be met and relying on the testing performed so far, many decide to skip the need for conducting acceptance testing using both external and internal resources. And what happens next—great loss for the company as the end user is dissatisfied with the software and refuses to use it. The company will face not only profit loss, but what is more important in the modern world: loss of reputation and credibility among partners, competitors and above all—"his/her majesty"—the end user, who is now to judge even harshly as ever, because of all the possibilities he/she has and the ease of switching from one software to another.

That is the reason each project manager must be aware of the main challenges of organizing and conducting acceptance testing. The benefits from it are numerous and will only lead you and your team to a greater success. Of course, it should be performed correctly and planned really well; otherwise you will have to deal with the losses that follow the release of a lousy product in the eyes of the end user.

Imagine, you find a severe bug in the acceptance testing phase—what are you supposed to do?

It is a well-known truth that bugs are easy to be fixed at an early stage—not only the bug does not affect the integrated system, but the time and resources; both human resources and financial resources are a lot less compared to fixing a bug later on in the development of the project. Imagine, you find a severe bug in the acceptance testing phase. What are you supposed to do? Ignore it and continue to delivery or try to fix it, but the effect that it will have on the system may postpone the release of the software for months. And why not skip the acceptance testing phase? Because it is the end user who will judge if your product is good or not and will determine its success.

Table 1 shows the status of the IT projects in the last 5 years—CHAOS Report 2015.

Following the notably famous reports, according to the CHAOS Report in 2015, the main reason for bugs are bad requirements—they are not clearly specified, they are incomplete and they are not taken into account seriously. This affects the development of the software, the software test cases created and in the end the conception of the project. Thus, the phase of accepting testing where both the client and the end users are included is the best way to check if all the expectations are met. It is better to postpone the release of the software than release a lousy product. Doing that, in the acceptance phase, you will be able to collect data both from the client (one more time, actually) and from the end user and make everybody involved in the project satisfied. Everyone knows how the business works—only one bad reference can ruin it all, despite having thousands of successful projects behind you.

Table 1 Status of IT projects according to CHAOS Report 2015

	2011 (%)	2012 (%)	2013 (%)	2014 (%)	2015 (%)
Successful	29	27	31	28	29
Challenged	49	56	50	55	52
Failed	22	17	19	17	19

The competitors will not hesitate to mention the bad experience you have had in that single project and gain points before prospective clients that were once yours. In the modern, competitive IT world, such big mistakes as skipping the acceptance test phase are no longer allowed.

But, to make sure that we all need that specific phase, here are some more charts derived from the CHAOS reports.

In the first one, we can see the most common reasons IT projects fail (Table 2).

Then, the second chart shows the factors that make a project successful (Table 3).

Comparing the two tables we can see a lot of familiarities and we can be certain to say that a very important factor for each single project is eliciting correct and not drastically changeable requirements. As we are all aware, good requirements are the basis of acceptance testing as well.

Table 2 The most common reasons IT projects fail

Project impaired factors	% of response
1. Incomplete requirements	13.1
2. Lack of user involvement	12.4
3. Lack of resources	10.6
4. Unrealistic expectations	9.9
5. Lack of executive support	9.3
6. Changing requirements and specifications	8.7
7. Lack of planning	8.1
8. Didn't need it any longer	7.5
9. Lack of IT management	6.2
10. Technology illiteracy	4.3
11. Other	9.9

Table 3 The factors that make a project successful

Project success factors	% of response
1. User involvement	15.9
2. Executive management support	13.9
3. Clear statement of requirements	13.0
4. Proper planning	9.6
5. Realistic expectations	8.2
6. Smaller project milestones	7.7
7. Competent staff	7.2
8. Ownership	5.3
9. Clear vision and objectives	2.9
10. Hard-working, focused staff	2.4
11. Other	13.9

2 Acceptance Testing, Part 2: The Process

How to organize the process so that you make sure that everything goes smoothly? What are the most important steps to execute the "perfect" acceptance testing?

To answer these questions we must start with the definition of acceptance testing as provided by the ISTQB: "formal testing with respect to user needs, requirements, and business processes conducted to determine whether or not a system satisfies the acceptance criteria and to enable the user, customers or other authorized entity to determine whether or not to accept the system. Acceptance testing is also known as User Acceptance Testing (UAT), end user testing, Operational Acceptance Testing (OAT) or field (acceptance) testing".

2.1 Types of Acceptance Testing

There are different types of acceptance testing that should be performed under different circumstances and regulations. For example, the most well known is the *user acceptance testing* where the criteria for "done" are usually created by business customers and expressed in a business domain language. Very popular is also the *operation acceptance testing* where the criteria are defined in terms of functional and non-functional requirements. Also, *contract and regulation acceptance testing* are executed where the criteria are defined by agreements or state laws and regulations. The most well known are the alpha and beta acceptance testing. What do they consist of? Developers of market software products often want to get feedback from potential or existing customers in their market before the software is put for sale commercially. That is when it comes to alpha and beta testing.

2.2 Alpha Testing

Alpha testing is performed at the premises of the developing company, not by the developers themselves: usually QAs that were not involved in the project, which provides the certainty of an independent and unaffected testing. This activity is quite often outsourced, which even guarantees greater independence of opinion as you will not harm the reputation and the feelings of your colleagues.

2.3 Beta Testing

Beta testing, also known as field testing, is performed by users or potential users at their own location. Many companies, quite often even start-ups consider this

type of testing as a free acceptance testing. You release the software and the customers do the testing job for you—finding and reporting the bugs. This also shows which functionalities are of greatest importance for the end users and where the development efforts should be focused. It sounds great, especially if you lack the budget to perform acceptance testing and you have a tight budget on other types of testing. Yes, the end user may do it for you, but are you sure that every bug they encounter will be reported, are you sure that even the reported bug is clearly described? Think about how much resources you will spend trying to identify a bug poorly described. And, after all, who will use your software if it turns out that your beta release was a really lousy product? Do not ever forget that in the modern world people are communicating very easily and the opinions in the forums for the products really decide what products the mass of the users will download and use. Our advice is to be very careful when making a beta release and making sure you did an alpha testing in advance.

2.4 Main Differences Between Alpha and Beta Testing

To be more precise in distinguishing alpha and beta testing as they are the most common ones, here is a comparative table (Table 4).

3 Acceptance Testing, Part 3: Approaches

While all other types of testing has the initial intent to reveal errors, acceptance testing is actually a crucial step that decides the future of a software, and its outcome provides quality indication for customers to determine whether to accept or reject the software.

Acceptance testing is often considered a validation process, and it does three things:

1. Measures how compliant the system is with business requirements
2. Exposes business logic problems that unit and system testing have missed out since unit and system testing do not focus that much on business logic
3. Provides a means to determine how “done” the system is

The very basic steps to organize an acceptance testing, where managers shall start, are:

1. Define criteria by which the software shall be considered “working”
2. Create a set of acceptance testing test cases
3. Run the tests
4. Record and evaluate

Table 4 Main differences between alpha and beta testing

Alpha test	Beta test
What they do	
Improve the quality of the product and ensure beta readiness	Improve the quality of the product, integrate customer input on the complete product and ensure release
When they happen	
Toward the end of a development process when the product is in a near fully usable state	Just prior to launch, sometimes ending within weeks or even days of final release
How long they last	
Usually very long and see many iterations. It's not uncommon for alpha to last 3–5× the length of beta	Usually only a few weeks (sometimes up to a couple of months) with few major iterations
Who cares about it	
Almost exclusively quality/engineering (bugs, bugs, bugs)	Product marketing, support, docs, quality and engineering—the entire product team
Who participates/tests	
Test Engineers, employees and sometimes "friends and family"; focuses on testing that would emulate 80% of the customers	Tested in the "real world" with "real customers" and the feedback can cover every element of the product
What testers should expect	
Plenty of bugs, crashes, missing docs and features	Some bugs, fewer crashes, most docs, complete features
How they're addressed	
About methodology, efficiency and regiment. A good alpha test sets well-defined benchmarks and measures a product against those benchmarks	About chaos, reality and imagination. Beta tests explore the limits of a product by allowing customers to explore every element of the product in their native environment
When it's over	
You have a decent idea of how a product performs and whether it meets the design criteria and it's "beta ready"	You have a good idea of what your customer thinks about the product and what she/he is likely to experience when they purchase it
What happens next	
Beta test	Release

3.1 Primary Approach to Acceptance Testing

Well, the primary approach to acceptance testing is a big different and that can easily be seen on the pie chart. It is usually done by the QA team or the BA team when it is not skipped or shortened. Quite often, it is not very clear how to set the boundary or the level of the user involvement in it. So, what the pie chart shows us is given in Fig. 1.

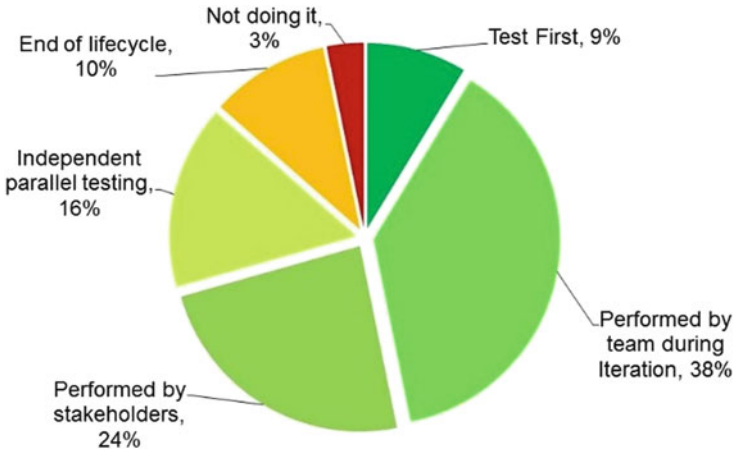


Fig. 1 Primary approach to acceptance testing

3.2 Basic Steps to Organize a Well-Done Acceptance Phase

Having seen what the primary approach and being ignorant of it, let's say, you are about to take the basic steps to organize a well-done acceptance phase. Even before taking them, you must make sure that you have in your pocket all the prerequisites needed to start, such as:

1. Business requirements are available and they are not incomplete or ambiguous.
2. The application code is fully developed.
3. All other types of testing such as unit, integration and system testing are completed.
4. There are no show stoppers or high or medium defects in the system integration testing phase.
5. You must have only the so called "cosmetic" errors before acceptance testing.
6. Regression testing should be completed with no major defects.
7. All reported defects are supposed to be fixed and retested.
8. The acceptance testing environment must be ready.
9. Sign-off mail or communication must be evident.

3.3 Approaches to Create Acceptance Test Cases

After you have made sure that your requirements are present, clear and complete, the next step is to start writing the acceptance test cases. There are several approaches to do so, and what is advisable is a mix of them. The approaches that you should

definitely take into account are the test cases that are requirements based (traditional approach), business process/workflow and data driven approach.

Why we advise you to use more than just one approach? Because if you use only the traditional requirements-based approach, then the test cases you create may also carry over the defects of the requirements. In addition to that when incomplete and incorrect requirements are present, then your test cases become incomplete and incorrect. The downside of the business process/workflow test cases is that data-related testing is out of its scope. And the data-driven ones focus heavily on the data and miss out the process and the business side.

Also, never forget that *writing the acceptance test cases is not the last step of system development.* Writing them starts right after the completion and approval of the requirements, data models and application models, and before the development is completed.

The tips we can give you when you start planning the acceptance testing phase are:

1. Identify and involve the key users—they have a deep understanding of the user requirements.
2. Provide not only demo but also a hands-on training of the system to the users.
3. Have the users write their own test cases as well.
4. Ensure that the users will also execute the tests.

More specifically, it is very important to clearly identify and set SMART boundaries of the roles, the type of testing to be executed—in person or self-paced—and the time frames (as you are aware time is never enough to perform a thorough testing), to set the documentation standards and determine the change control process.

3.4 Roles and Responsibilities

The roles are similar to the one in the Scrum team—you should have an Owner/PM who will manage the process and take the final decisions within the team. The Owner will also update the project sponsor on the status. Then, the project sponsor or Business Owner comes. She/he will take care of the requirements and will assist the Owner/PM when testing them. That person will also be solely responsible for the Change Control process. The team resources will actually do the acceptance testing.

4 Acceptance Testing Part 4: Conclusions

4.1 The Execution: What to Avoid When the Acceptance Testing Is Performed?

The execution is to come—it should not be the last step and should be done frequently and also manually when needed. Do not forget to record and evaluate the data, so that you are sure that nothing is missed out. What to avoid when the acceptance testing is performed—you will face numerous pitfalls, the one you should definitely avoid are:

1. Insufficient planning—we are always ready until we are not.
2. Lack of system availability and stability.
3. Lack of resource availability and stability—imagine a massive turnover rate and everybody leaves when the acceptance starts.
4. Poor communications channels—it is no different than any other aspect of the development and the testing process—if you lack the proper communication and the right communication channels, then you are definitely going to fail as there will always be misunderstanding among the team members.
5. Limited record keeping—keep your eyes on the essentials.

A very important aspect that is a real pain is user involvement—this is a highly recommended factor to make your acceptance successful.

4.2 Involving the End Users

Involving end users in the acceptance testing phase is great, but it also sets a few quite important challenges:

1. Users do the acceptance testing in addition to the their busy schedules and to avoid that you should have them testing not at the last moment, but as early as possible;
2. As it is the last phase, acceptance testing may be turned to be a “formality” and to avoid that, the users should write their own test cases and let the test alone as they have less or no devotion to the team/project/software;
3. Then comes the challenge to motivate the users to do thorough testing even when they have busy schedules. Well, get to know your users and use different motivational techniques.
4. The worst challenge you may face is the lack of understanding of how the system works—here, the frequently mentioned test cases created by users will help you.

And then, having in mind all the troubles that can be met with the involvement of the end user, you decide to follow the scenario where you entirely rely on the QAs in your company who have worked on the project. The fact is that the QAs

that have tested the software influence the acceptance testing phase too much. That is definitely not good if you want to prove that the system does the things it is supposed to do in the way it is supposed to do them.

4.3 Acceptance Testing Performed by the Internal QA Team

The result of doing the acceptance testing with the QA hired in your company and that were doing the system testing are as follow:

- It only proves that the application works as shown by the previous test stages.
- The coverage is quite small and mainly UI—some even argue that this is what most of the users see, the Pareto principle.
- They do what they do day to day as the corner cases are covered by the Functional/System Testing and is proven correct.
- Acceptance testing is not supposed to find any defects—the things that do not work shall come as change requests, i.e. the primary requirement was at fault.
- Testers should not be involved in acceptance testing as it is a milestone for requirements and their correctness.

4.4 What Are the Main Reasons the Acceptance Testing Phase Often Fails?

Thus, we come to the point to find out what are the main reasons the acceptance testing phase often fails. It is because there is no collaboration and no management buy-in. The other reason is the wrong focus—mostly testers focus on *how* and not *what*. Besides that, acceptance testing is often neglected; it is fully performed by different and sometimes not suitable tools. And then, when the objectives of the team are not aligned and the skill set required is underestimated, there is no possibility to have a successful end to the acceptance testing phase.

Therefore, it is quite understandable to see the fear of acceptance testing when it comes to that point in the testing process. It requires a lot of efforts, a lot of good planning and the ability to be able to adjust quite fast to the new realities. It is also not good to underestimate the requirements of the end users involved in that phase—as most of them will not be very good at IT literacy, you should be careful and patient while explaining the system and the steps for writing their test cases. Yes, that is difficult—you use a set of terms with your team and then the end user comes and is not aware of those terms—you will have to change the way you express yourself and find common words for the terms for the ordinary world. But, the result of that involvement should never be neglected—the ideas the end users can provide and the bugs they can find—usually the one you and your team will definitely have missed out.

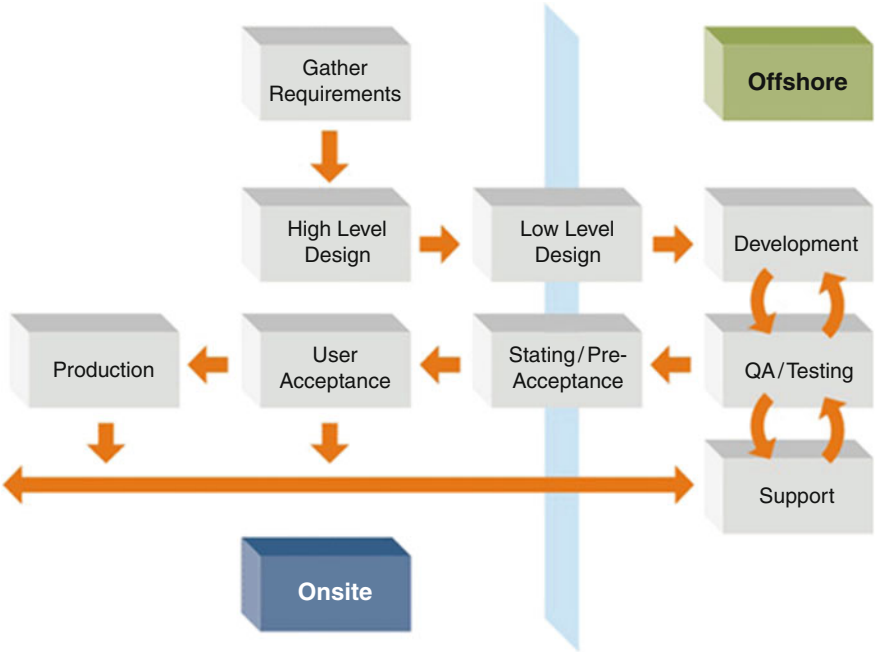


Fig. 2 Outsourcing acceptance testing: onsite and offshore tasks

4.5 Best Option for Successful Acceptance Testing

What is the best option to do the acceptance testing phase successful and, at the same time, avoid all the obstacles set before you?

To outsource the acceptance testing to an independent organization that will organize and carry it out for you both with the QAs it has and with end users that they will find and train (Fig. 2).

Yes, despite the model and the strong belief that outsourcing the acceptance testing phase is not a good decision, many nowadays believe the opposite—outsourcing the acceptance testing is one of the great options as you set the requirements and you can guarantee to your client that the results are really independent and show what the opinion of the others of the product is. The outsourcing model works the proper way when both parties are sharing information in a timely manner and that information is correct and not misleading in any way.

4.6 To Outsource Acceptance Testing or Not?

It will not be misleading to say that the acceptance testing phase is not the preferred one to be outsourced as the managers are quite reluctant to lose control; moreover,

they strongly believe that the internal team has a better understanding both of the system and of the requirements of the end user. As other research show, this is not quite true and may even lead to a failure.

A few reasons in support of outsourcing acceptance testing:

1. An external team definitely adds value in terms of completing the test coverage.
2. They have a more objective view of the business scenarios that may occur in that industry.
3. External consultants can help test the performance of the application during peak periods.

Before outsourcing the acceptance testing phase, there are always things to consider. The six main ones are listed below:

1. Establish goals for engaging with the acceptance testing consultants—the vendor should have expertise in the area and also should be engaged quite early in the process.
2. Innovation and customization are key qualities for the vendors—look for vendors with a creative approach to testing.
3. Analyse trends and metrics
4. Encourage cross-functional coordination and interorganizational communication —proper communication and the cooperation between internal and external teams is key to successful outsourcing.
5. Select the right people for the right job.
6. Develop effective tracking and controlling mechanisms—at least at the beginning. After you have established a good relationship with the vendor, you may skip those well-defined and measurable parameters for monitoring and control.

Thus, after carefully planning everything and considering the option of outsourcing, you will have a really good release at the end of the acceptance testing phase. It is not an easy task, but must not be neglected. Neglecting it will definitely cost more than executing the acceptance testing on your own with all the struggles and a good option is outsourcing.

Distributed Testing Teams

The Challenge of Working with Distributed Testing Teams and Performing a High-Quality Work

Alfonsina Morgavi

Abstract Working with distributed testing teams means a significant change in management, perspective and level of effort.

In this chapter, I'll try to describe some of the experiences I have gone through during the past 3 years, and the strategies we are implementing to succeed.

Keywords Software testing · Software tester · Software quality · Onshore testing · Offshore testing

1 Introduction

Those of us who have been somehow pioneers in testing, know from experience that the specialty has had a difficult childhood. Comparing it with people's life cycle, we might agree that testing has also had a difficult journey through teenage years. We can say that it was not until a few years ago that testing earned its own identity, and the practice started being handled in a more mature way with processes, methodology and a broad spectrum of training possibilities. However, can we say with certainty that we are now going through a stage of maturity?

Constant evolution of technology and new global delivery methodologies have enabled us to work in offsite, nearshore and offshore modes with teams distributed in different regions of the globe, taking advantage of varied expertise in multiple disciplines.

But are we really ready to work with distributed teams and still provide high-quality services? (Fig. 1)

The challenge is how to achieve highest performance when working with distributed teams in different locations, how to train them for a homogeneous work and how to reach the right level of communication so that it doesn't became a barrier in their performance.

A. Morgavi
QActions System SRL, Buenos Aires, Argentina

S. Goericke (ed.), *The Future of Software Quality Assurance*,
https://doi.org/10.1007/978-3-030-29509-7_8

Fig. 1 How is the map for the onshore, nearshore and offshore service?

Management of distributed teams definitely plays a fundamental role.
Let's go over some tips that could help us succeed.

2 Selection of an Offshore Team

Once you decide which tasks will be outsourced to an offshore team you will have to focus on its selection.

The selection of an offshore testing team requires an exhaustive evaluation that goes beyond required technical skills. Considering cultural differences and manners, language, time zones and the way of communicating will become a must.

Getting to know the strengths, weaknesses and technical capabilities of your offshore team will save you a lot of headaches in management.

If disregarded, cultural differences can lead to huge misunderstandings between teams; even when speaking the same language, a single word could have a completely different meaning from one country to another.

3 About Training

Training offshore teams is not as easy as onsite or near-shore trainings. To start with, it generally implies a not so short of a trip.

It is possible that the offshore team has specific skills different from the ones your onsite team has; therefore, it will be necessary to train them on topics such as methodology to be used for the specific project, testing process used by the

contracting company, the way in which the documentation is received, the kind of documentation to be delivered and the business of the final client, among others. Let's take into account that the main purpose of the training is to provide the teams with the general context of each project and align them into a common understanding and way of working.

Do not forget that your onsite team has multiple opportunities to acquire information from other testers, either from their own project team or from other testing projects, and a broad access to informal communication means. This kind of opportunities practically disappear when working with offshore testing teams.

So, it will be necessary to define a training plan that considers the following aspects:

- Identify and list the topics to be included in the training.
- Set dates and times in which sessions will take place.
- Define best training mode: remote, face-to-face, mixed or other.
- Define how sessions will be carried out, who will lead them and where will the training material be made available.
- Decide the training material that needs to be read before the sessions.
- Estimate training duration with as much precision as possible.
- Consider full availability of documentation, videos, reading material and any other means to complete knowledge transfer. Do not forget that teams rotate, and people move forward.

Despite the chosen training mode, at some point nothing replaces "face to face" training efficiency. So, to obtain better results it becomes crucial to consider a "face to face" mode for at least part of the training. It is very difficult to get people to identify with values, vision and mission of a company through a corporate video or a Skype session.

Each team must be clear about what is expected from them, what are the objectives of each project, how management will be carried out, which communication channels will be used, who will inform tasks or results to which person, when and how. All this must be part of the collaboration environment.

4 Define How Communication Will Be

It is necessary to define communication channels, officialize them and formally inform them. It is also very important to proactively verify that all members of the project know these channels and have full access to them.

The healthy practice of agile methodologies that maintain daily huddles is a way of improving connection between work teams. It will be important to consider the geographical location of the teams as, depending on the time zone, the end of the day for some members of the team could be the beginning of the working day for others.

Setting frequent short calls could perfectly avoid long email chains in which it is easy to lose the sequence and generate confusing information without solving the issues. Teams should take part in planning and effort estimation meetings. This will allow to have a realistic feedback on the possibilities and acquire their commitment with set goals.

On the other hand, it is fundamental to define a focal point of the offshore team, that is the contact person with whom you can review the progress of tasks, issues and delays that might appear. Soft Skills, among other competences, is a must for the position.

5 How to Maximize Productivity of the Offshore Team?

5.1 About Requirements

When an offshore team assumes Functional Testing tasks, beyond the initial training that I have already mentioned in the previous sections, it will be necessary for the team to have clear requirements either in the form of use cases, user stories or any other format depending on the project.

An incomplete, unclear or obsolete requirement is a source of problems. It will surely generate delays and rework in the tasks to be carried out later. "Requirements Review" is in my opinion an extremely efficient practice. If the information received is not enough to define some test cases, development will be facing the same problem as the testing team, since for sure something would be missing or wrong and this lack of clarity will also impact the development. The key is to avoid disconnected creativity and work on a common base agreed upon by all parties involved.

When this matter remains unsolved, Development and Testing might have their own different understanding of what is missing. Thus, when later executing the tests, we will find ourselves reporting defects that are not real and missing defects that must be reported.

Regardless of the development methodology used, requirements must be always clear, complete and concise.

5.2 How to Design Test Cases

We know that there isn't a single, unique correct way of designing test cases. This issue should be part of training if necessary. When we have different teams working on one same project, we need to unify criteria for test cases design, defining the level of detail to be used when designing the cases and the information they will contain.

5.3 About Tests Execution

As for test cases, it is important to define how defects will be reported, and what information the defect report will contain. All the information generated must be registered and available for all project stakeholders, either for viewing only or for viewing and updating.

5.4 About Tools

Necessary and appropriate tools for project management should be available for the teams. Make sure to communicate and formalize tools to be used. If necessary, include trainings and periodically check if the tools are being used correctly. Tools will allow to measure at least a large part of the offshore team's work.

6 Retrospective

Post-mortem meetings are common practice of agile teams; however, they might not be familiar in other Development models.

Implement them regardless of the development methodology used! They are an information source that allows you to learn from your mistakes and those of your teams. Don't think of them as "witch hunting"; they are about identifying points of improvement. Get your lessons learned!

Don't think of them as "catharsis", they are about detecting items to be improved and practices to maintain.

Choose a moderator for the sessions, invite to the meeting all technical players of the current project and collect experiences that may be useful for future projects. Identify and record what was done properly during the project and spread it among all teams so that these kinds of practices are maintained, as long as they seem useful for future projects. However, keep in mind that what is valid and useful today for a specific project may not be useful for others.

Do not miss the lesson!

7 Some Conclusions

For these kinds of services, an exhaustive selection of the offshore supplier is highly recommended. The supplier should match your needs and those of the business you are working for.

Do not minimize the importance of teams training; it is essential to reach homogeneity. In addition, full or partial usage of face-to-face training modes gives you the opportunity to meet those people who will provide you with a service.

Take care of communication; confusing or insufficient communication will lead to many problems and delays in projects.

Use the right tools and register, register, register; do not forget to record what has been done!

Last, but not least, do not miss lessons you can obtain from post-mortem meetings. Learn from failures to improve your processes, management and communication.

Further Reading

1. Rothman, J., Kilby, M.: From Chaos to Successful Distributed Agile Teams. Practical Ink, Victoria (2019) ISBN-13: 978-1943487110
2. Derby, E., Larsen, D.: Agile Retrospectives: Making Good Teams Great. Kindle edition by The Pragmatic Bookshelf. (August 2012) ISBN-13: 978-0977616640
3. Venkatesh, U.: Distributed Agile: DH2A – The Proven Agile Software Development Approach and Toolkit for Geographically Dispersed Teams, 1st edn. Technics Publications LLC, Westfield, NJ (2011) ISBN-13: 978-1935504146
4. Ramesh, G.: Managing Global Software Projects: How to Lead Geographically Distributed Teams, Manage Processes and Use Quality Models, 1st edn. Tata McGraw-Hill, New Delhi (2005) ISBN-13: 978-0074638514
5. O'Duinn, J. (ed.): Distributed Teams: The Art and Practice of Working Together While Physically Apart. Release Mechanix, LLC, San Francisco, CA (8 August 2018) ISBN-13: 978-1732254909.
6. Sutherland, L., Janene, K., Appelo, J.: Work Together Anywhere: A Handbook On Working Remotely—Successfully—for Individuals, Teams, and Managers. Collaboration Superpowers, Delft (April 2018).

Testing Strategies in an Agile Context

Zornitsa Nikolova

Abstract Testing in an Agile context is extremely important, not only with its function to ensure quality but also to guide development efforts into the right direction. This is related to a shift in testing paradigms, with quality being viewed as a factor early on in product development, rather than a late-stage reactive activity. It also requires the application of different approaches, such as automation, to enable the flow of potentially shippable product increments. Many teams find themselves stuck into the old ways of testing, especially as they work on legacy systems. However, investment in upskilling quality experts, applying the proper tools, and changing the way testing is done can bring tremendous value and opportunities for innovation. Proper change management needs to be applied to enable teams transition successfully into an Agile mindset and new practices.

Keywords Software testing · Software quality · Agile testing · Test automation

1 Introduction

Irrespective of the methodology that you use for managing your product development, quality assurance is important. And when we say "quality assurance," we typically mean "testing." Agile contexts are not different in this sense. In Agile, we value "working product more than comprehensive documentation,"[1] hence the majority of teams' efforts shall be focused on creating software and making sure it does what it is supposed to do.

Yet, in many cases when we look at processes in various teams, even if they claim they are Agile, we still see quality as just the final step of creating the product. We design, we build, and then we test, usually with a focus on discovering bugs. Even though we talk about team ownership on results, it's still a quality expert mainly

[1] Manifesto for Agile Software Development, www.agilemanifesto.org

Z. Nikolova
Leanify, Sofia, Bulgaria

S. Goericke (ed.), *The Future of Software Quality Assurance*,
https://doi.org/10.1007/978-3-030-29509-7_9

responsible for testing, reporting problems, and making sure Definition of Done is satisfied with respect to quality standards. Imagine a situation: it's the end of the sprint, most stories are finished from a development perspective, and they are in the QA column on the board. Teams' QA experts are working around the clock to make sure they have checked new features thoroughly. Yet, at the Sprint demo they still cannot report full success—developers finished a couple of more stories in the final day of the Sprint, and they couldn't complete testing . . . Sprint has not been entirely successful, and it is a vicious circle. Does it sound familiar?

Unfortunately, I still see the above situation way too often, even though we claim that Agile approaches are by now mainstream. Believe it or not, this way of working stands in the way of true Agile adoption in teams, and requires a certain change of paradigms, so that we can benefit from Agile software development.

2 The Shift in Testing Paradigms

Situations like the one described above happen often when the switch to Agile practices focuses primarily on the question "what methodology shall we apply?" Do we want to do Scrum, Kanban, Scrumban, or something else? While I believe it is an important question, I do not think focusing too much on it really helps us understand and adopt Agile practices. Frameworks and methods, such as Scrum and Kanban, are there to support teams achieve a certain goal. So, defining the goal, the purpose of applying Agile practices, is the first thing to do.

According to the latest State of Agile report from Version One [1], among the key reasons for adopting Agile are the need for faster development as well as enhanced software quality. Yet, in many cases, creeping technical debt and a lot of rework prevail, partially caused by changing requirements, but also, to an extent, by defects found late in development.

Teams that are successful in addressing such challenges apply a different way of thinking about testing, which is illustrated by the concept of Agile testing quadrants [2] (or the Agile testing matrix [3]) (Fig. 1).

The quadrants imply several important aspects of a change in thinking about testing and quality in general. First of all, we shall not think of testing only as a means to discover bugs—this is a very reactive and limiting view on the quality process. A much more empowering view on testing suggests that it has two faces—one focused on product critique (finding functional bugs, unwanted behaviors, performance, security, and other nonfunctional flaws) and another focused on supporting the team to take the right decisions upfront, by doing frequent small tests on unit, component, and feature level (the left side of the matrix). This second aspect of testing is largely underutilized though, especially in teams that transition to Agile from other paradigms. Traditionally, we are used to testing for bugs, and this is the common profile for a quality expert. Eventually, we end up with a lot of back loops from testers to developers for fixing issues that can easily be prevented

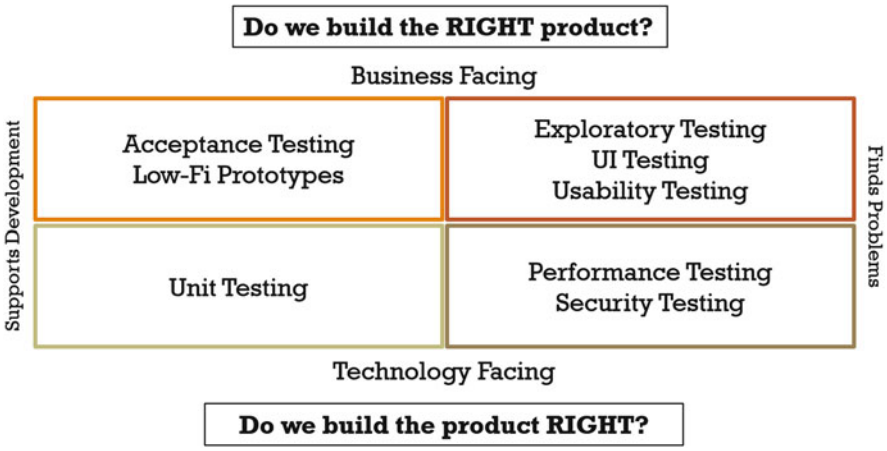


Fig. 1 Agile testing matrix

with upfront investment in tests that check how good requirements are defined, what is the level of understanding, and how we can design our solution in an efficient way.

Secondly, we shall extend our scope of thinking about quality as something related to the technology side of the product. When we create software, we take decisions related to the technical design, architecture, and implementation, but a good portion of decisions are related to the business side of the product as well. How do we check if those decisions are correct? Previously, we would invest a lot of resources to build a complete product and only then launch it for a market test. This used to work for big companies (sometimes) but in the world of startups with limited resources it is not a viable approach. That is why startups naturally adopt Agile practices, including smart ways to test the business aspects of the product as early as possible, through prototypes (often low fidelity) and simulations. Business-oriented testing checks whether we are building the right product from a user and market perspective, while technology-oriented testing checks whether we have built it right.

Finally, an implication from the matrix is that testing is not only a QA expert's job. The various types of testing require various expertise, and contribution by everyone—even customers and users themselves. So, it is in the collective ownership and responsibility of the entire team to embrace the different aspects of quality and engage in activities that contribute to the different types of testing. Quality is no longer a one-man's quest for perfection—it is a team effort to ensure meaningful investment and strong product performance.

3 Investment in Automated Testing

The Agile testing quadrants offer a great thinking model around the aspects of testing that we need to address and master, so that we can have good quality on both business and technology level. However, it is also obvious that doing all those

tests manually is hardly possible, especially if we are aiming for frequent and quick feedback and minimized waste of waiting time. Naturally, the paradigm of Agile testing involves moving from manual to automated testing.

It is easier said than done though, unfortunately. In reality, many teams are faced with a huge legacy of inherited code, written for over 10+ years, and productively used by real customers. Often, to automate testing for such systems requires certain refactoring, which on the other hand is quite risky when unit tests are missing. It is a Catch 22 situation. Moreover, in some cases systems are written in proprietary languages (take SAP's ABAP, for example) that lack proper open-source tools and infrastructure for test automation. Investing a big effort in automation might be a good idea from a purely engineering viewpoint, but it might be hard to justify from a return-on-investment perspective. Doesn't it sound familiar? The constant fight between the team and the Product Owner on how much we shall invest in removing technical debt!

When planning our strategies for automated testing we need to consider a few aspects that might play a role in this decision-making exercise. First of all, it is important to acknowledge where our product is in terms of product lifecycle (Fig. 2).

The graphic represents the standard concept of product lifecycle with respect to its market penetration, applied to software products. In this context, there are slight differences as compared to physical products. First of all, with software products, especially following the ideas of the Lean startup and Agile business-oriented testing, we might see much earlier exposure to the market—already in the Conception and Creation phase. This means that we need to think about quality aspects quite early, as technical debt tends to build up in these early stages of software development and this leads to impediments in the growth stage. At the

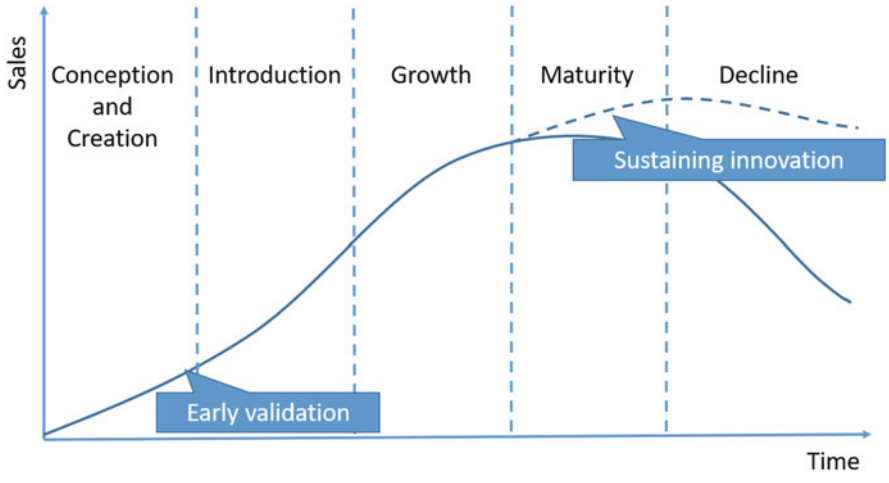


Fig. 2 Software product lifecycle

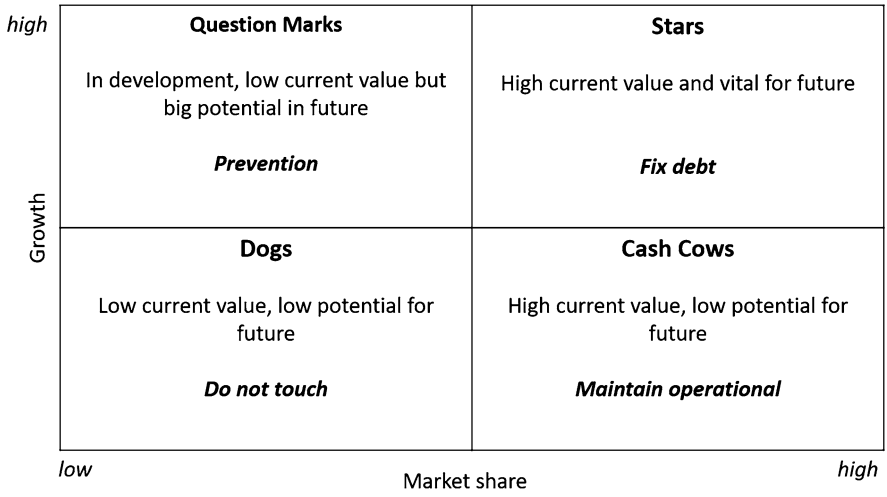


Fig. 3 BCG matrix

same time, a mature product might go back to growth if we decide to invest in sustaining innovation (e.g., to extend its scope and cover a new market segment).

When talking about legacy systems, we shall consider first where they are in terms of lifecycle phase, and to what extent we plan to develop them further (either as part of their natural growth or through sustaining innovation). Any investment in further growth shall be accompanied by an investment that supports development, that is—investment in applying Agile testing and test automation is essential.

Similarly, we can look at our strategy for investment in automation and cleaning up technical debt using the Boston Consulting Group (BCG) matrix (Fig. 3).

Looking at products from the perspective of current and potential value to the business gives an important input when we try to estimate return on investment. Note that in this case we are looking at a return in the midterm rather than short term, as creating the infrastructure and building up automation from scratch is not a short-term task either. So, we can generally follow some of the strategies suggested in the figure. For "cash cows"—the products that are currently in a mature phase, yielding return from previous investments but also not likely to grow significantly in future—undertaking significant investment is not recommended. We might need to optimize to some extent, so that we can improve operational maintenance (e.g., by automating regression testing partially), but shall be conservative when it comes to big effort for automation. On the other hand, for our "stars"—products that are potentially in a growth phase and strategic for business—we might even want to consider a "stop-and-fix" effort. The sooner we invest in building up a solid infrastructure that enables us to continue development with the support of automated testing, the more stable velocity of development we can maintain overtime. Then for "question marks" we are in a position to prevent the buildup of technical debt in general. This means

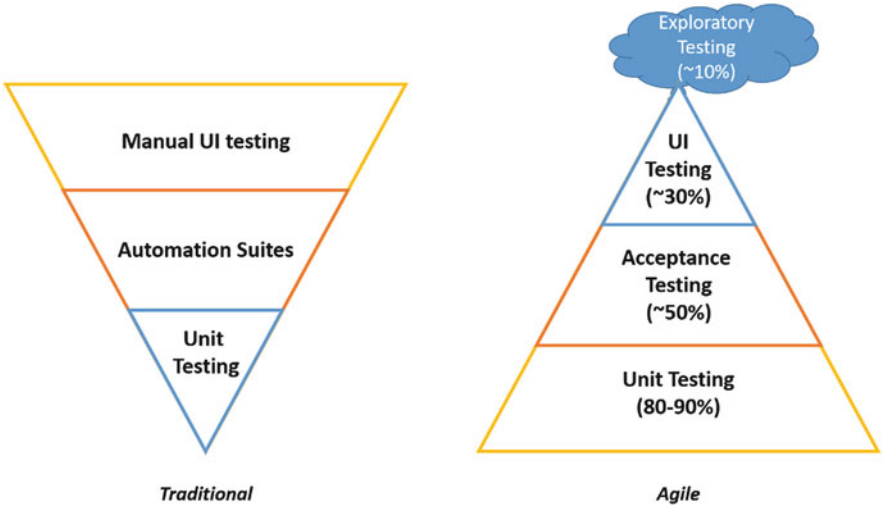


Fig. 4 Agile testing pyramid

making sure that automation is part of the standard Definition of Done and the effort is planned accordingly by the team in each Sprint.

The product lifecycle and BCG matrix offer a more business-oriented view on the question of investment in automation. Now let's look at the technical perspective. Mike Cohn's testing pyramid [4] offers a nice visualization of the system layers where test automation shall be considered, and to what extent (Fig. 4).

In our traditional way of working, most of the testing is done manually, and it typically requires access through a UI layer. This means that testing happens quite late, and it requires significant development effort upfront, hence potentially a lot of defects piling up and being discovered quite late when rework is more costly. As discussed in the previous section, the focus is on finding bugs and product critique, and this is an expensive way to address quality. No wonder that it is often compromised, especially when we are late with deadlines and there is pressure to deliver. Not to mention that manual testing is also much slower, of course, and this makes things even worse.

In the Agile paradigm, we need to reverse the pyramid, as shown on the right side of the picture. The biggest effort for automation is done on the unit test level. This is where ongoing development gets immediate feedback and bugs are quickly removed as part of the regular development process. On the next layer, we can automate acceptance tests based on cross-unit and cross-component functional calls within a certain use case or user story, but not necessarily involving the user interface. This integration testing is a perfect way to ensure working increments during sprints. The highest layer is testing end-to-end scenarios via the UI of the system.

Naturally, the cost of automation raises as we go up the pyramid—automating on the UI layer is typically resource consuming, and automated tests are hard to

maintain and update. Therefore, we shall spend most effort on the lower layers, automating on a unit and service level, while still doing manual testing on the UI layer. Note, however, that we can optimize manual testing strategies as well to get the biggest value of the effort spent there. Coming to the manual UI-based tests, we don't need to do full-blown regression testing, or cover all end-to-end scenarios, as we have already covered them on the integration tests layer. Here, the focus is more on nonfunctional (performance, security, usability, accessibility, etc.) testing, as well as simulation of real-life usage through exploratory and user acceptance testing.

To summarize, major investment in automation makes sense for products that are still in growth and innovation mode, and it is required for a long-term success. We shall also be selective on the amount of investment per system layer to gain maximum returns—investing in automation of unit and integration tests is typically advisable, as it speeds up development. When it comes to UI testing, we might consider automating some of the manually done smoke and regression tests, while taking into account the ongoing test maintenance effort and picking appropriate tools.

4 Transitioning to Agile Testing

Even when the team and the organization is convinced in the benefits of Agile testing, including investment in test automation, getting started on it might be another hard task. There are a lot of challenges to changing the entire process of how you plan and execute tests—from purely infrastructural (tools, test system design, etc.) through skills in the team to create and execute those tests to mindset changes that need to happen, and fears that need to be overcome. Starting from point zero is scary and not easy at all, and many teams might find themselves at a loss as to where they should start.

I am a strong believer in goal-oriented thinking, and systems such as OKR (Objectives and Key Results). Starting with the end goal in mind creates focus, motivation, and resourcefulness in people to overcome challenges as they go. So, defining our short- and midterm objectives is an excellent way to kick off a transformation of quality assurance in the organization. Of course, as in any goal-setting process, being unrealistically ambitious might fire back at some point, creating a sense of disbelief and demotivation in the team. We have to choose targets carefully, ideally in collaboration with the team.

A good practice that I have experienced personally might be to get a team of early adopters on board, train them, and get them support from an Agile coach with knowhow in Agile testing paradigms. This team becomes the catalyst for subsequent activities within the individual Agile teams as well. Note that you can have the Scrum Masters coaching teams in Agile testing practices, but if the Scrum Master is not a technical person, he or she might not be the most appropriate one to assume this role. At this point, the most important thing would be that the person is enthusiastic to learn and work to implement and improve these practices within the team. Once

we have the catalysts, they can initiate discussions in the Agile teams and create a baseline for starting the transformation. They will need to assess what they are already doing and suggest what would be the next important milestone or objective that the team needs to strive for in the midterm (the next 1 year, for example). From there backwards, they can then define reasonable objectives for the short term (let's say next 3 months). Here is an example of how this might look like.

> Objective: Enable automated testing for key integration scenarios in the newly developed modules of product XYZ within Q1'2020
>
> Key results: 80% coverage with unit tests for newly created classes; full coverage of priority 1 integration scenarios as defined by Product Owner

This is a close-to-accurate quotation of an objective that we set in a team I used to work with. They were developing a new add-in product on top of a legacy system that did not offer a very favorable environment even for starting with unit testing. When we started talking about applying Agile testing concepts to new development, the reaction of people was: "This is totally impossible. It would mean that we double or triple the development effort if we also have to write automated unit or integration tests." Still, they were somehow convinced to give it a try (the pain they felt each time they had to touch anything in the legacy was a strong factor in making them more open to experiment). We had to start small and check if we could make it happen at all at a reasonable cost. So, we started with new development only, focusing to cover new classes with unit tests, and key integration scenarios with service-level integration tests. We did not do automated UI testing at this point—manual UI-level tests were run as usual, just to create a baseline for us for checking the effect of automated integration tests as well. It took several sprints before we could really feel the difference. However, in the moment when we started iterating on features developed a couple of sprints earlier based on the user feedback that we got, the benefits of automated integrations tests became very obvious. There was no need to convince anybody anymore—developers happily planned tasks for creating more automated tests in their sprint backlog. A release later we started extending our strategy to cover with automated unit and integration tests also those legacy parts that we had to touch and rework as part of the new development efforts. Essentially, we were doing continuous innovation, and it made sense to start investing also in covering the old pieces with good testing.

Along with setting objectives and measurable results, we also decided to experiment with techniques such as TDD (test-driven development). It was not an easy mindset shift for developers either, but over time they appreciated the added focus on simplicity that TDD drives. We could experience quality improvement in system design, and gradually see a reduction of defects that were discovered at later stages. On the level of business requirements, we introduced BDD (behavior-driven development) or "specification by example." This was a great way to formulate requirements in a way that enabled straightforward creation of integration test cases as well. All of this eventually had a significant impact on both business- and technology-facing aspects of the product and was a good way to minimize

effort spent on documentation (such as requirements and test cases) by creating lean executable specifications.

Regarding UI testing, we intentionally limited the scope of automated testing. We researched a few different tools to see how we can automate part of the tedious regression testing that was done repeatedly before each new release. Our experience showed that tools tend to cover one of two scenarios. In the first scenario, tests are very easy to create by simply recording the manual test case, and then replaying it. However, if any of the screen components change (and this is often the case when using certain development frameworks that rebuild components, changing their IDs at each system build), the effort to adapt them is quite high. In the second scenario, tools allow for better componentization and identification of screen components, which makes initial effort high but leads to less difficult adaptation in case of changes. In our case, we picked a tool that supported the first scenario, and started automating only basic regression tests on critical user journeys to make sure we can quickly cover high-priority testing needs. In addition to that, we engaged in much more Quadrant 2 testing using low- and high-fidelity prototypes to run early tests with real users. This reduced significantly the need to rework UIs later in development, and minimized the effort to update UI tests as well.

The example I shared, in combination with the concepts discussed in the previous sections, might give you a few ideas as to how to start applying some of the Agile testing concepts. When you map your own transformation strategy, however, write it down, start measuring against the KPIs you have defined, inspect, and adapt on a regular basis to make sure that your goals are achievable and you are getting the results that you expect.

5 The People Perspective

Finally, I would like to draw attention to another important aspect as well—creating the appropriate environment for teams and individuals to feel safe in the transition and to effectively achieve an Agile mindset shift. No matter what type of change you are undertaking, having people on board, feeling appreciated and valued, and engaging them in the process is among the key factors to success.

This perspective is a complex one as well. First of all, let's look at teams as a whole. As we discussed in the beginning, in an Agile context quality is a team responsibility. This means that we need to support the building of this collective ownership by coaching teams into self-organization and focus on common results rather than individual achievements. It might require changing the entire performance management approach that we apply in the organization, switching from individual to team goals and product-related metrics—and many Agile organizations do that. We might also need to rethink the KPIs that we use. I was recently discussing the topic of Agile testing with a quality engineer, and he shared that their work is being evaluated by the number of bugs found. What kind of behaviors and thinking does a KPI like that support? What would be the motivation

of people on this team to invest in preventing bugs rather than discovering them late?

In addition to goals, roles and responsibilities in the teams might need to shift, also creating demand to people to extend their expertise into what we call the T-shaped profile (people with deep expertise in a specific topic, such as testing or development, for example, and complementary skills in adjacent topics—development, UX, etc.). This will enable teams to be more flexible in handling the collective responsibility on quality and will strengthen the communication and understanding between team members. It is not a process that goes overnight, however. It requires some space for teams to experiment and for team members to learn (potentially by failing in a controlled way). Managers might need to step back and let teams reshuffle tasks, and individuals cross the borders of their specific function, so that they can learn.

This last point leads us to the individual perspective as well. In most organizations, people are hired to fit a certain job description with clearly defined expectations and responsibilities. With the concept of self-organizing teams and quality as common responsibility, the role of a tester or even quality engineer might alter significantly or even become obsolete. Naturally, this creates fear and uncertainty and leads to additional resistance to change. The role of managers in such an environment is to balance these fears, provide support and guidance, so that team members who need to refocus will be able to do it, see opportunities for personal growth, and continue adding value to the team. Managers need to ensure those individuals have the resources and environment to learn new skills and can see how their role can change to fit the altering needs of the team as well. It is important to note that quality engineers often have a unique view on the product that enables them to play a very important role in activities related to product discovery, user journey mapping, acceptance criteria definition, identifying scope of prototypes and simulations, and test automation, of course.

Looking at the topic from a broader perspective, how we ensure quality in an Agile context is a very significant part of doing Agile and being Agile. It involves the entire team and requires appropriate thinking, skills, and focus. Picking the right strategy would depend on the maturity level of the organization, and the will of people inside to replace traditional approaches with new ones—and it is related to the value that we expect to get from the change on a team and individual level.

6 Conclusion

In this chapter, I have offered a broad view on Agile testing and quality as a process that underlies the success of the product from both business and technology perspectives. I believe the main value of Agile approaches comes from the fact that they do not put a strong demarcation line between technical and nontechnical roles and responsibilities in the team. On the contrary, Agile thinking involves development of customer-centric mindset and understanding of the business domain

in the entire team, while also bringing nontechnical team members on board in tech-related discussions. In well-working Agile teams, this creates an environment of collaboration, common ownership on outcomes, and joint care on all aspects of quality that we have looked at.

Agile practices have been evolving significantly for 25+ years now, and we can leverage what multiple teams and practitioners have achieved through empirical learning. Yet, in most cases, we need to do our own piece of learning as well, mapping our strategies, experimenting and adapting as we go.

As a summary, here are some key takeaway points that we discussed:

- Testing in an Agile context is very important to ensure that we are both building the right product and building it right.
- Agile thinking involves a shift in testing paradigms as well, shifting it left as early in the development process as possible.
- We need to engage different practices and the whole team to be successful.
- For traditional organization, and even some Agile teams, this requires a significant transformation that takes time and needs to be properly managed.

References

1. 13th Annual State of Agile Report. https://www.stateofagile.com/#ufh-c-473508-state-of-agile-report. Accessed 23 June 2019
2. Crispin, L., Gregory, J.: Agile Testing. A Practical Guide for Testers and Agile Teams. Addison-Wesley, Toronto (2009)
3. Brian Marick's blog. http://exampler.com/. Accessed 23 June 2019
4. Cohn, M.: Succeeding with Agile. Addison-Wesley Professional, Upper Saddle River, NJ (2009)

Testing Artificial Intelligence

Gerard Numan

Abstract In AI, the algorithm is not coded but produced by a combination of training data, labelling (concepts) and the neural network. This is the essence of machine learning. The algorithm is not directly insightful and cannot be bug-fixed directly: it is "black box development".

AI systems are used in contexts with diverse data and usage. Choice in training data and labels brings risks in bias and transparency with possible high impact on real people. Testing AI focusses on these risks. An AI tester needs moral, social and worldly intelligence and awareness to bring out the users, their expectations and translate these in test cases that can be run repetitively and automated. AI testing includes setting up metrics that translate test results in a meaningful and quantifiable evaluation of the system in order for developers to optimize the system.

Keywords Software testing · Software quality · Artificial intelligence · Machine learning · Test automation

1 Introduction

The future is AI. It has entered our everyday lives and is being used by major companies all around the world. Adaptability of AI seems endless. And yet many doubts and concerns exist. For example, in the case of self-driving cars: liability in case of accidents, wobbly object recognition and complex interaction with unpredictable human traffic participants are blocking widespread acceptance.

Some possible scary effects of AI have already manifested themselves. AI algorithms can create or enlarge bias. Like in the case of the ethnic cleansing in Myanmar, where tens of thousands of Rohingya were killed and one million fled. Already existing ethnic tension was supported by the Facebook algorithm, which

G. Numan
Polteq Test Services B.V., Amersfoort, The Netherlands

S. Goericke (ed.), *The Future of Software Quality Assurance*,
https://doi.org/10.1007/978-3-030-29509-7_10

strengthened prejudiced opinions because it was optimised to reward click-success. Negative information showed up in search results increasingly.

Every software developer and customer of AI struggles with these doubts and risks. What is a bug in case of AI and how to fix it? How to be certain that the system does the right thing with a great variety of input and users? How to get the right level of confidence? Are the results fair to all concerned? Are current developments, opinions and values reflected in the algorithm?

What are the biggest risks with AI and how to deal with them from a testing point of view?

2 An Introduction to AI for Testers

This chapter is a short introduction to AI and an analysis of aspects relevant to testing.

2.1 AI Is Black Box Development

In AI, the algorithm, the behaviour of the system in terms of criteria, decisions and actions, are *not* explicitly engraved in the code. In non-AI development the code directly expresses the algorithm. In AI the algorithm is the product of training data, parameterisation, labels and choice of the neural network. But the algorithm cannot be found in the code. The code, the neural network, is just a, be it very essential, part of a system which produces the algorithm by training. This is the essence of machine learning.

2.2 Machine Learning and Neural Networks

There is a strong analogy between machine learning and human learning. Take for example a child who learns to use a concept for the first time. The child has been told that the hairy creature it cuddles is a “cat”. Now the child sets its own neural network to work. The concept of the cat is compared to objects which aren’t cats, such as “daddy”. The neural works is finding ways to configure itself in such a way that had it seen the cat, it would classify it as a cat and not as daddy. It does so by finding differences, criteria, such as fur, whiskers, four legs, etc. But we do not know exactly what these criteria are. They might also be “hunting mice”, “purring”, or “being white”. We cannot find the concept of a cat and it’s criteria inside the brain, nor can we correct it directly in the brain.

A neural network consists of many blocks of code (“nodes”) which are arranged in layers. Each layer of nodes is connected to its top and bottom layers. The nodes

are not programmed upfront to perform specific tasks, they are empty. The nodes are merely small calculators, processing parts they have been presented by top layers, returning a calculated result. When the neural network is presented with an example in training it will systematically configure itself so the different layers and nodes will process parts and aspects of the input so the end result of all nodes will give the result that is given to the network (the label). Given two pictures, of a cat and of daddy, it will try different configuration in order to find the configuration that would determine one example to be a cat and the other as daddy. It would seek out the differences so it's configuration would come up with the right classification next time.

In this way the neural network creates models of the labels: these reflect differences between a cat and daddy the neural network has identified based on the training data.

2.3 *Algorithm = Data + Code + Labels*

So what the system produces is an algorithm that consists of models derived from examples so it can classify and recognise input and assign these to labels. The algorithm is the product of the neural network but based strongly upon the training data (the examples) and the goals (the labels). So the algorithm is NOT the code, but the code + training data + labels. Because the algorithm cannot be identified it can also not be fixed directly. Brain surgery will not fix the child's flaws in recognising a cat.

2.4 *Fuzzy Logics and Mathematics*

Although all the system does is calculating, produce numbers, these numbers will not produce a Boolean result: for example: "this is daddy" or: "this is a cat". The result will be the summation of all calculated numbers from the nodes and layers, each giving a number which expresses the extent to which criteria have been met as per each given label. This will hardly ever be 1 on a scale of 0–1. Next to that: it will also produce the extent to which the example scores on the other labels. So a new picture presented to the system could score "cat-ness" as 0.87 and "daddy-ness" as 0.13. The conclusion would be that the example is a cat, but it's not 100% a cat, nor is it 0% daddy.

So the end product of AI is a calculation, a probability and never a 100% certainty.

2.5 Development and Correction

Development of a neural network consists of developing a neural network itself, but most developers take a neural network off the shelf. Next they need to configure the neural network so it can receive the input at hand and configure labels, so examples are linked to these.

Finally the layers of the neural network can be parameterised: the calculated results can be weighted so certain results will have more impact on the end result than others. These are the main tweaking instruments developers have. If the system is not performing satisfactorily the parameters can be tweaked. This is not a focussed bug fix, correcting one case of faulty decision.

Parametrisation will influence the outcome, but each tweak will have impact on the overall performance. In AI there is massive "regression": unwanted and unexpected impact on parts of the system that are not intended to be changed.

Training data and labels are also likely candidates for influencing the system. In certain issues with AI, such as underfitting, expanding the training data will very likely improve the system. Underfitting means the algorithm has a too simplistic view of reality, for example when a cat is only classified as a furry creature. Adding more examples of a cat to the training data, showing more variety of species, races and behaviour, could help the system distinguish a cat from other creatures better.

2.6 Overall Version Evaluation and Metrics

When bug fixes cannot be focussed and each tweak has massive regression, massive regression testing is necessary. The question "did we fix this bug?" becomes a minor issue. We want to know the overall behaviour each time we change something. We want to know what the overall performance of the system is compared to other versions. In that overall evaluation we need to take into account the output of AI: calculated results which are not either true or false. Each result is a grade on a scale. So the end results should be thoroughly compared, weighed and amalgamated so we can decide if a version as a whole is better than another and we should use it or not. The result will be metrics calculating the value of output based on expectations and their relative importance.

3 Risks in AI

We'll discuss the most important risks here. These risks are typical of AI and could have serious impact on the quality of AI, it's customers, users, people and even the world. These risks should be considered before starting testing, giving clues to where to put emphasis as a tester. When analysing test results the risks should be

considered as a cause-effect analysis of unwanted outcome. This could give clues for optimising the system. For example: under-fitted systems most likely need more diverse training data, over-fitted systems streamlining of labels.

3.1 Bias

The main risks with AI are types of "bias". In human intelligence we would call this prejudice, reductionism or indecisiveness. Because of limits in training data and concepts, we see things too simple (reductionism) or only from one point of view (prejudice). A high granularity in concepts could mean that the system can't generalise enough, making the outcome is useless (indecisiveness).

Significant types of possible bias in AI are discussed next.

3.1.1 Selection Bias

If the training data selection misses important elements from the real world, this could lead to selection bias. Compared to the real results, the polls for the last European elections predicted much higher wins for the Eurosceptic parties in the Netherlands than they did in the real election. The polls did not filter on whether people were really going to vote. Eurosceptics proved more likely not to vote than other voters.

3.1.2 Confirmation Bias

Eagerness to verify an hypothesis heavily believed or invested in can lead to selecting or over-weighing data confirming the thesis over possible falsifications. Scientists, politicians and product developers could be susceptible to this kind of bias, even with the best of intentions. A medical aid organisation exaggerated a possible food crises by showing rising death numbers but not numbers of death unrelated to famine and the overall population number in order to raise more funds.

3.1.3 Under-fitting

Training data lacking diversity causes under-fitting. The learning process will not be capable to determine critical discriminating criteria. Software that was trained to recognise wolves from dogs, identified a husky as a wolf because it had not learned that dogs can also be seen in snow. What would happen if we only get drugs-related news messages in the Netherlands?

3.1.4 Over-fitting

Over-fitting occurs when the labelling is too diverse and too manifold for the purpose of the AI system. If we want to see patterns and groupings, a high granularity of labels compromises the outcome, making it unusable because of its indecisiveness.

3.1.5 Outliers

Outliers are extreme examples that have too much influence on the algorithm. If the first cat your 1-year-old child sees is a Sphynx, a naked race, this will have a major impact on his concept of a cat and will take multiple examples of normal cats to correct.

3.1.6 Confounding Variables

Pattern recognition and analysis often requires combining data, especially when causal relations are looked out for. Confounding variables occur when different data patterns are associated for data analysis purposes that have no real causal relation. It has often been believed that drinking red wine could evoke a migraine attack, because drinking red wine and migraines reportedly occur sequentially. New research has shown that a migraine attack is preluded by changes in appetite, such as a craving for red wine. Drinking red wine is a side effect and not a cause of migraine!

3.2 Over-confidence in AI

AI can perform some types of mental activities on a scale and with a velocity and precision that is unachievable by humans. The algorithm of AI is not directly accessible or adjustable. From this the intuition can be easily obtained that AI cannot be judged by human standards and is superior. Intellectual laziness and comfort can be an important motivation too for uncritically trusting AI. Who questions the results of Google search?

A possible consequence of over-confidence is the transfer of autonomy to an instance outside of our individual or collective consciousness. AI does not need to achieve self-consciousness to be able to do this, as sci-fi teaches us. It takes over-confidence or laziness.

3.3 Under-confidence in AI

The other side of this is under-confidence. A rational debate on whether to use AI can be blurred by uncertainty, irrational fear or bias in the media (or sci-fi movies). Accidents with self-driving cars get more headlines than ordinary accidents. People are afraid to become obsolete or that a malicious ghost in the machine might arise.

3.4 Traceability

With non-AI-systems the algorithm is the code. This is not the case with AI-systems so we don't know the exact criteria by which the AI-system takes decisions. Next to that it's hard to oversee the total population of training data and therefore get a good understanding of how the AI system will behave. So when the outcome is evidently incorrect, it is hard to pinpoint the cause and correct it. Is it the training data, the parameters, the neural network or the labelling? Lack in traceability fuels over-confidence and under-confidence (as was shown above) and causes uncertainty in liability (was it the software, the data, the labelling or the context that did it?) and lack of maintainability (what to correct?).

4 Testing AI

The key to mitigation of the AI risks is transparency. In bias we need insight into the representativeness of training data and labelling, but most of all we need insight into how important expectations and consequences for all parties involved are reflected in the results.

Building the right amount of confidence and traceability needs transparency too. Transparency will not be achieved by illuminating the code. Even if this were possible, by showing a heat-map of the code indicating which part of the neural network is active when a particular part of an object is analysed or a calculation in a layer is produced, means close to nothing. Looking inside a brain will never show a thought or decision. It could show which part is activated but all mental processes always involve multiple brain parts to be involved and most of all experience from the past.

AI systems are black boxes, so we should test them like we do in black box testing: from the outside, developing test cases that are modelled on real-life input. From there expectations on the output are determined. Sounds traditional and well known, doesn't it?

The basic logic of testing AI might be familiar, the specific tasks and elements are very different.

Traditionally requirements and specifications are determined upfront and testers receive them ready to be used at the start. In AI, requirements and specifications are too diverse and dynamic to expect them to be determined at the start completely and once and for all. Product owners and business consultants should deliver requirements, but testers need to take initiative to get the requirements in the form, granularity and actuality that they need.

The challenges with testing AI and their accessory measures from start to finish are discussed next.

4.1 Review of the Neural Network, Training Data and Labelling

Static testing can detect flaws or risky areas early.

The choice for the neural network or its setup can be assessed: is it fit for purpose? What are the alternatives? For this review a broad knowledge is required of all possible neural networks and their specific qualities and shortcomings.

The training data and labels can be reviewed and assessed for risk sensitivity:

1. Does the data reflect real-life data sources, users, perspectives, values well enough? Could there be relevant data sources that have been overlooked? Findings might indicate selection bias, confirmation bias or under-fitting.
2. Are data sources and data types equally divided? How many representatives do various types, groups have compared to one another? Findings might indicate under-fitting, selection bias, confirmation bias or outliers.
3. Are the labels a fair representation of real-life groups or types of data? Do the labels match real-life situations or patterns that the system should analyse? Findings might indicate over-fitting, under-fitting or confounding variables.
4. Is the data current enough? What is the desired refresh rate and is this matched? Are there events in the real world that are not reflected well enough in the data?

4.2 Identifying Users

The owner of the system is not the only valuable perspective! AI-systems like search systems are an important part of the world of its users but also of those that are "labelled" by it. The quality of an AI-system can have moral, social and political dimensions and implications so these need to be taken into account.

The users of AI are often diverse and hard to know. They are not a fixed set of trained users, all gathered in a room and manageable in their behaviour and expectations. They could be the whole world, like in the case of a search engine: an American tourist visiting Amsterdam or an experienced art lover in the field at hand have very different needs and expectations when searching for "Girl with pearl" in

the search engine of a museum. The tourist wants to know if a particular picture is for display, the art lover also wants background information and sketches.

Next to that: as the world changes, the users and their expectations could change overnight. Think of what the fire in the Notre Dame did to what users might expect when searching for "Notre Dame" or "fire in Paris". AI recognising viruses in DNA sequences should take into consideration possible mutations that occur constantly.

So testing AI starts with identifying the users or the perspectives from which output from the system will be used. This means studying data analytics on the usage of the system, interviewing process owners or interviewing real users.

4.3 Profiling Users

Identifying users or groups of data is one, determining what they want, expect, need, are afraid of or will behave like, is another. What the tester needs is profiles of the users and perspectives: what is their typical background, what do they want, what turns them off or upsets them and what do they expect?

A technique to create profiles is "Persona". Key to this technique is to not think of an entire group of users but to pick one from this group and make her or him as concrete as possible. The benefit of Persona is that it makes the user come alive. It's a technique to take the perspective of a user from the inside out. For example: the Persona for American tourists could be Joe, a plumber, living in Chicago, white, aged 45, married, two children. He is not well read but loves colourful and well-crafted paintings. His hobbies are fishing and refurbishing old audio equipment. He is turned off by profound theories but likes the human side and background of things (Fig. 1).

4.4 Creating Test Cases

This part is probably where most of the work is for the tester. Per user profile, input and expected output is determined. Good profiles will provide a good basis but will probably need extra information coming from research and interviews.

Identifying test cases will never be complete nor definitive: you can't test everything, also not in AI. The world and the users change so this needs to be reflected in the requirements. It starts with the most important cases; it will grow constantly and needs permanent maintenance.

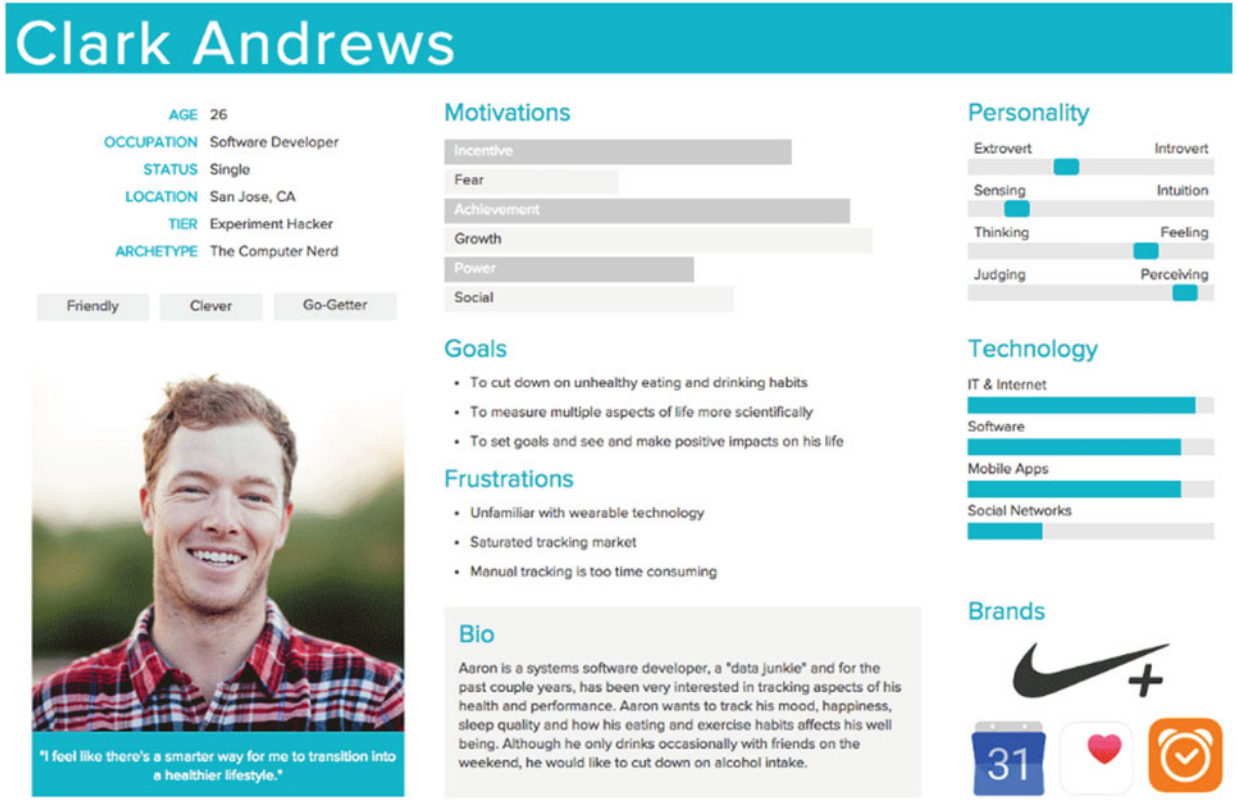


Fig. 1 Profiling users

4.5 Test Data

What test data to use and whether it can be created, found or manipulated depends on the context and the availability of data from production. Data creation or manipulation (like in case of image recognition) is hard to do and sometimes useless or even counter-productive. Using tools to manipulate or create images brings in an extra variable which might create bias of its own! How representative of real-world pictures is test data? If the algorithm identifies aspects in created data that can only be found in test data, the value of the tests is compromised.

AI testers create a test data set from real-life data and strictly separate these from training data. As the AI system is dynamic, the world it is used in is dynamic, test data will have to be refreshed regularly.

4.6 Metrics

The output of AI is not Boolean: they are calculated results on all possible outcomes (labels). To determine the performance of the system, it is not enough to determine which label has the highest score. Metrics will be necessary.

Take, for example, image recognition: we want to know if a picture of a cat will be recognised as a cat. In practice this means that the label "cat" will get a higher score than "dog". If the score on cat is 0.43 and dog gets 0.41, the cat wins. But the small difference between the scores might indicate fault probability.

In a search engine we want to know if the top result is the top 1 expectation of the user, but if the top 1 result is number 2 on the list, that sounds wrong, but is still better than if it were number 3. We want to know if all relevant results are in the top 10 (this is called precision) or that there are no offensive results in the top 10.

Depending on the context we need metrics to process the output from the AI system into an evaluation of its performance. Testers need the skills to determine relevant metrics and incorporate them in the tests.

4.7 Weighing and Contracts

The overall evaluation of the AI system also has to incorporate relative importance. Some results are more important than others as is with any testing. Think of results with high moral impact like racial bias. As part of designing test cases their weight for the overall evaluation should be determined based on risks and importance to users. Testers need sensitivity for these kinds of risks, being able to identify them, translating them into test cases and metrics. They will need understanding of the context of the usage of the system and the psychology of the users. AI testers need empathy and world awareness.

In the movie *Robocop* officer Murphy had a "prime directive" programmed into his system: if he would try to arrest a managing director of his home company, his system would shut down. AI systems could have prime directives too, or unacceptable results, like offensive language, porn sites or driving into a pedestrian. We call these "contracts": possible unwanted results that should be flagged in the test results as blocking issues or at least be given a high weight.

Required contracts have to be part of the test set. Possible negative side effects of existing contracts should be part of the test set too.

4.8 Test Automation

AI testing needs substantial automation. The amount of test cases request it and tests need to be run repetitively with every new version. When the AI system is trained constantly, testing is necessary, as in the case of search engines where there are feedback loops from real data. But even when the AI system is not trained constantly and versions of the system are stable, a changing context demands constant training. Even when the system does not change, the world will.

Test automation consists of a test framework where the test cases will be run on the AI system and the output from the AI system will be processed. Below a basic setup of such a test framework is shown.

4.9 Overall Evaluation and Input for Optimising

The product of testing is not just a list of bugs to be fixed. Bugs cannot be fixed directly without severe regression, as stated above. The AI-system has to be evaluated as a whole since with the many test cases and regression, no version will be perfect. Programmers want to know which version to take, if a new version is better than a previous one. Therefore the test results should be amalgamated into a total result: a quantitated score. For programmers to get guidance into what to tweak (training data, labelling, parametrisation) they need areas that need improvement. This is as close that we can get to bug fixing. We need metrics, weighing and contracts to achieve a meaningful overall score and clues for optimisation. Low scoring test cases should be analysed as to their causes: is it over-fitting, under-fitting or any of the other risk areas?

4.10 Example of AI Test Framework (Fig. 2)

From left up to bottom and then right up:

1. Identifying user groups
2. Creating persona per user group

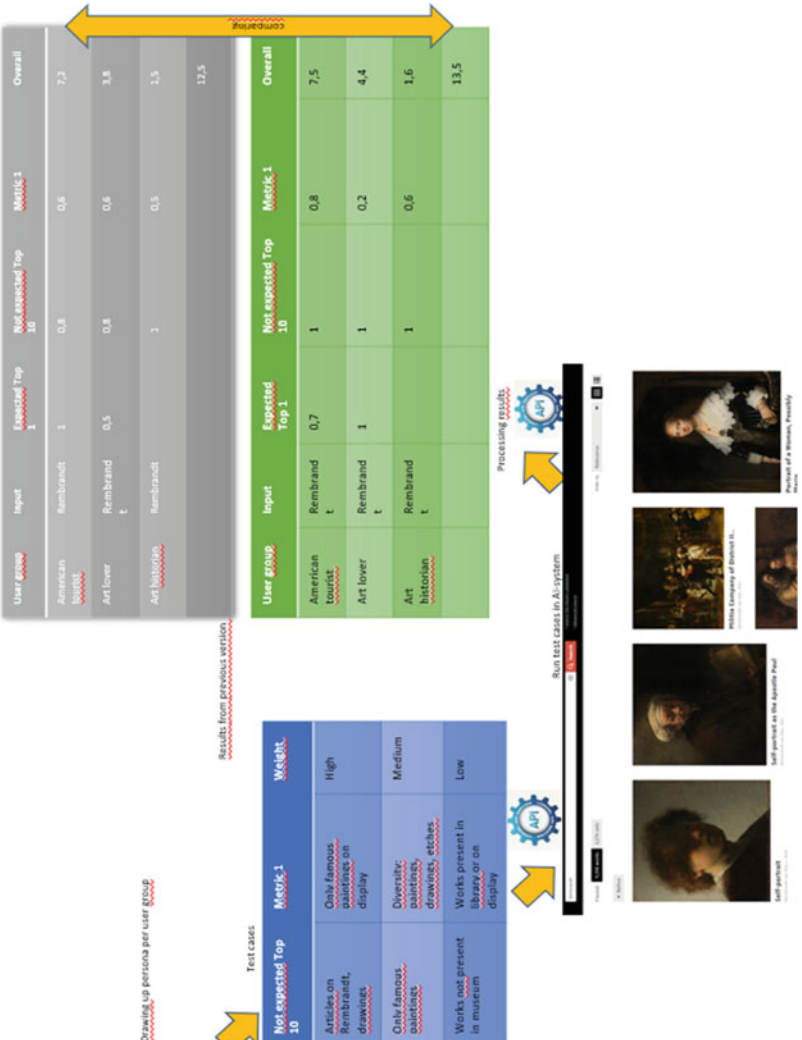


Fig. 2 Example of AI test framework

3. Writing testcases: per user group input with expected top result, not expected, metrics and weight
4. Running test cases in AI system (search engine)
5. Processing results
6. Creating test results per test case with overall weighing
7. Comparing results with results from previous version

5 Conclusions

The world of AI is very dynamic: the algorithm does not equal the code but is a result of training data and labelling. Training data will be updated constantly as the world changes. The output of AI is not Boolean but calculated results on all labels which could all be relevant.

Despite low transparency and risks in Bias, AI is being used for decision making and is an important part of people's worlds. Testers must play a role in creating transparency, by identifying user groups and their specific expectations and needs and showing how the system reflects these. For this purpose an automated test framework is needed to compare the many versions of the AI system, monitor quality in production constantly and give guidance to optimisation.

An AI tester needs skills in data science but most of all moral and social sensitivity!

Responsible Software Engineering

Ina Schieferdecker

Abstract Software trustworthiness today is more about acceptance than technical quality; software and its features must be comprehensible and explainable. Since software becomes more and more a public good, software quality becomes a critical concern for human society. And insofar artificial intelligence (AI) has become part of our daily lives—naturally we use language assistants or automatic translation programs—software quality is evolving and has to take into account usability, transparency as well as safety and security. Indeed, a majority worldwide rejects currently the use of AI in schools, in court or in the army because it is afraid of data misuse or heteronomy. Insofar, software and its applications can succeed only if people trust them. The initiatives towards "responsible software engineering" address these concerns. This publication is about raising awareness for responsible software engineering.

Keywords Software testing · Software quality · Software engineering

1 Introduction

Regardless of whether it is an autonomous vehicle, an artificial intelligence or a mobile robot—software is always steering these systems and is a key component in digital transformation. The same applies to critical infrastructures that shape the lives of millions of people: utilities for electricity, water and gas, but also large parts of the transport infrastructure are already based on information and communication technologies (ICT). Scenarios for smart homes, smart manufacturing and smart cities are even further extending the influence of software on our everyday lives [1]: "The software industry directly contributed €304 billion to the EU economy in 2016, representing 2% of total EU value-added GDP—up 22.4% from €249 billion in 2014. The sector employed 3.6 million people and paid €162.1 billion in wages.

I. Schieferdecker
Federal Ministry of Education and Research, Berlin, Germany

S. Goericke (ed.), *The Future of Software Quality Assurance*,
https://doi.org/10.1007/978-3-030-29509-7_11

Software companies were responsible for an average of 1.8% of total jobs in the seven EU countries in this study."

By that, software leaves the pure technical system management and control and serves today increasingly for decision support and decision control in socially critical contexts. The questions that arise are which processes are necessary, how to make the software comprehensible and who bears the responsibility. And how can companies ensure the reliability, quality and security of software in these increasingly complex environments?

Even further, not only in the use, even in the production and distribution of software, the expectations of customers are exhaustive. Hence, software development is to go faster and faster, the product at the same time getting better and better. And this expectation is not necessarily paradoxical. In many cases, DevOps' holistic approach can actually increase both the speed of delivery and the quality. Where the word "deliver" in times of cloud computing does not really characterize the process anymore. After all, the customer no longer buys a product, but instead access to a service. Even—and especially—such a service will only be accepted if users trust it.

Important elements in establishing trust are so-called testbeds for field experiments and experimental environments for co-innovation. Traceability and transparency should be part of software development. Because the complexity of software-based systems continues to grow, and because data retention and use is often difficult to understand, trust in software is still often fragile. Hence, principles of anti-fragility in software have to be added, which add fault resilience and robustness at run time [2]. Also, the traceability of goals, features and responsibilities need to become part of any software engineering and documentation. Why this is so important is explained in this quote from computer scientist Joseph Weizenbaum: "A computer will do what you tell it to do, but that may be much different from what you had in mind". And although methods and tools for high-quality, reliable and secure software development are available in large numbers, they are to be extended to address runtime faults as well [3].

The chapter starts with a general consideration of software and current pressing issues. This is followed by a review of the current understanding of ethical principles in software engineering and draws conclusions towards responsible software engineering. A summary and outlook complete the chapter.

2 Software and Recent Software Quality Requirements

Software is basically a set of instructions that tells a computer, embedded control systems or a (micro-)processor what to do and how to do it. Software is not just a programming code on firmware, operating system, middleware or application level. It also consists of data that represent the content managed by the programs as well as data that train or steer the programs [4]. In addition, it encompasses meta-data that represent information and documentation of the software [5]. According to ISO [6], software is a "fundamental term for all or part of the programs, procedures, rules,

and associated documentation of an information processing system. ... (It) is an intellectual creation that is independent of the medium on which it is recorded."

Software exists in many types and variants and there is no widely adopted software taxonomy. Rather, there exist surveys for specific fields of software applications or software development tools such as in software-defined networks [7], for user interfaces [8] or for software documentation [9].

Although it is apparently hard to grasp characteristics of software in general, there is a long-lasting and still growing understanding of how software quality is constituted. The ISO 25010 provides an updated set of software quality requirements [10] compared to ISO 9126 or other previously established software quality models, like the one by Boehm et al. [11], FURPS [12], by IEEE [13], Dromey [14] or QMOOD [15].

Still, since software technologies evolve, the understanding of software quality needs to evolve as well. For example, software usability addresses the ease of use of a software [16]. For specified consumers, it seeks to improve their effectiveness, efficiency and satisfaction in achieving their objectives by the given context of use and the usage scenarios for a software.

Another example is the growing importance of data (as software in itself and as a software artefact in use) in big data, Internet of Things or artificial intelligence applications [17]. Data quality assessment [18] aims at deriving objective data quality metrics that resemble also subjective perceptions of data.

Let us also refer to the growing use of software in emulating reality in virtual and augmented reality applications in gaming, for education or for training [19, 20]. By that, multimedia (streams) in 2D, 3D and eventually 4D contexts in presentation, but also in interactive modes, require more elaborated media, interaction and collaboration attributes in software quality.

As a final example let us consider the growing need for transparency of software so that users receive a solid understanding about what a software provides and what it does not provide. The more software permeates into every field of our society, transparency, traceability and explainability become the central quality criteria that also demand attention from software developers [21].

3 Software Criticality and the Need for Responsible Software Engineering

To the extent that society is increasingly dependent on autonomous, intelligent and critical software-based systems in energy supply, mobility services and production, but also in media discourses or democratic processes, new strategies must be found to ensure not only their well-understood quality characteristics such as safety, efficiency, reliability and security, but also their associated socio-technical and socio-political implications and all additional requirements in the context of human-machine interaction and collaboration [22].

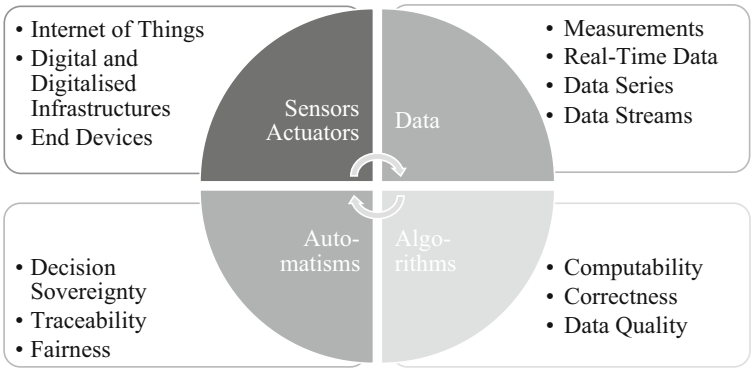


Fig. 1 Elements of software-based systems [23]. Sensors are part of the Internet of Things and generate different kinds of data such as measurements, series of measurements or data streams. Algorithms use these data in their computations or as training data. The algorithms are constrained by complexity, computability and performance limits and possibly by the (in-)correctness of the implemented computation logic and by the (un-)biased (training) data. As a result, software-based systems offer automatisms for which it is essential to agree (and assure) decision sovereignty, traceability and fairness. Any decision in respect to the environment can finally be fed via software (into the cyberspace) and via actuators (into the environment)

Such software-based systems use functionalities as being defined in (meta-) algorithms and steered by data (see Fig. 1).

These software-based systems are also called algorithm-based or algorithmic systems [23]. They are being used for decision-making support or decision-making a in socio-critical context (e.g. in elections), business-critical context (e.g. in online trading) and relevant to self-determination for individuals, organizations and nations. This raises the discussion about the necessary guidelines for the design, development and operation of these software-based systems, which must be understood in the interplay of technological, social and economic processes. They are and become increasingly critical for the whole human society and developed into a public good [23].

In fact, most of the values designed and encoded into these systems stem from the software engineered by the business owners, product owners, software designer and/or software engineers [24]. Software engineering is constituted mainly by (1) defining and constraining the software (requirements engineering and software specification), (2) designing and implementing the software (coding), (3) verifying and validating the software (simulation, model checking, testing, etc.) and (4) operating, maintaining and evolving the software. Software engineering does not need to follow a line of software engineering methods [25], but rather a line of value concerns [23]: Responsible Software Engineering should be constituted by:

1. *Sustainability by Design* by people in power: A critical examination of these value inscriptions should serve as the basis for conscious, reflected valuations, also in order to realize values from the sustainability context. In addition to

the promotion of privacy, safety and security, and quality through appropriate software engineering, sustainability should be anchored in software engineering, for example (1) the ecological sensitivity for energy and resource efficiency of software and (2) the value-sensitivity in data collections, algorithms and heuristics.

2. *Techno-Social Responsibility* by the software community: Not only corporate social responsibility [26] should be addressed by the digital community, but also a techno-social responsibility in the meaning of (1) understanding how digital business models could as well as should not affect society and (2) shaping the digital business models, solutions and infrastructures according to the agreed societal principles.
3. *Responsible Technology Development* by the society: Responsible software engineering should be strategically promoted and supported by appropriate research funding, also known as Responsible Research and Innovation [26] in the meaning of research and innovation (1) based on societal goals, which should also (2) explicitly anchor and demand the UN sustainable development goals [27].
4. *State-of-the-Art Software Engineering* within every software project: It is in the responsibility of the people in power and in action to make use of those software engineering methods and tools that fit the purpose and that fit the level of software criticality. This is not only a matter of tort liability but also of societal responsibilities in light of safety-, security-, environment-, or business-critical software-based systems.
5. Last but not least, such responsible software engineering (see Fig. 2) could be promoted by a *Weizenbaumian Oath* [28] to reflect the professional ethics for sustainable design, development, operation and maintenance, and use of software and of software-based systems. Joseph Weizenbaum (1923–2008) was a computer science pioneer, who critically examined computer technologies and the interactions of humans and machines. He called for a responsible use of technology. Through the Weizenbaumian Oath, all the tech communities could commit to general principles that guide the development and application use of software and of software-based systems. These principles should also become an integral part of the education and training of experts and may constitute a new module in education schemes in software engineering including ISTQB [29].

4 Ethical Principles in Responsible Software Engineering

In responsible software engineering, in addition to software quality matters, the focus is on the comprehensibility, explainability and fairness of software-based systems, and on the ultimate people's decision sovereignty in critical socio-technical contexts. Professional organizations such as the Association for Computing Machinery (ACM) or the German Association for Informatics (GI) already give guidance

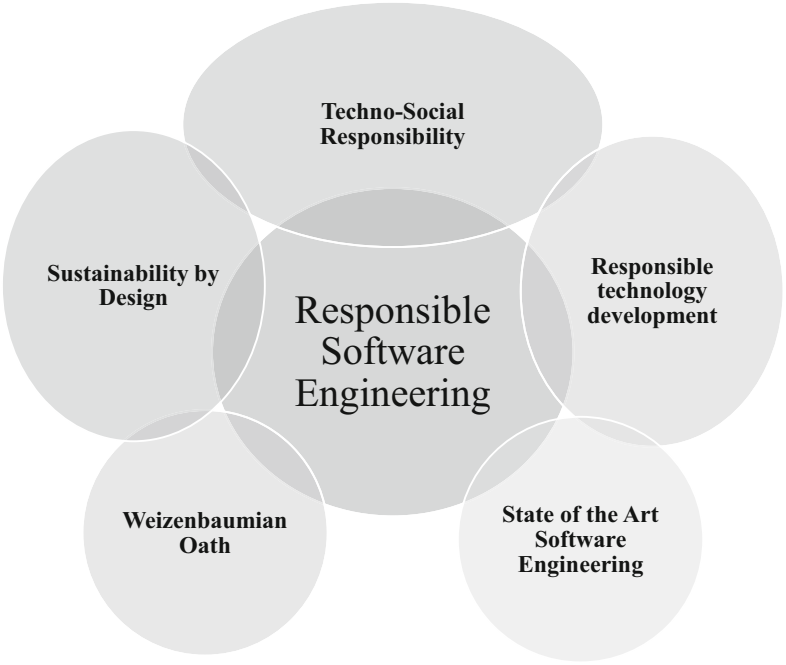


Fig. 2 Constituents of responsible software engineering

to the tech communities through recently updated ethical guidelines [30, 31]. These and similar initiatives provide a solid basis for the extension of professional ethics towards responsible software engineering.

In view of AI, automation and criticality of software-based systems, the initiatives by the High-Level Expert Group on Artificial Intelligence by the European Commission, by the AI4People and by iRights.Lab explained below provide rule sets for coping with software-based systems and have an understanding of dynamically updating these rule sets in view of ongoing socio-technical and socio-political discourses as well as along rapid technical advancements. Its national, European and international operationalization is an open field for implementation and regulation. In addition, all three represent an urgent need for action because the initiatives remain ineffective if they do not lead to the best possible implementations, which prevent side-effects and unintended risks, in a timely manner [23].

The High-Level Expert Group on Artificial Intelligence by the European Commission is working on Ethics Guidelines for Trustworthy AI [32] and has the following normative foundations:

1. "Develop, deploy and use AI systems in a way that adheres to the ethical principles of: respect for human autonomy, prevention of harm, fairness and explicability. Acknowledge and address the potential tensions between these principles.
2. Pay particular attention to situations involving more vulnerable groups such as children, persons with disabilities and others that have historically been disadvantaged or are at

risk of exclusion, and to situations which are characterized by asymmetries of power or information, such as between employers and workers, or between businesses and consumers.
3. Acknowledge that, while bringing substantial benefits to individuals and society, AI systems also pose certain risks and may have a negative impact, including impacts which may be difficult to anticipate, identify or measure (e.g. on democracy, the rule of law and distributive justice, or on the human mind itself.) Adopt adequate measures to mitigate these risks when appropriate, and proportionately to the magnitude of the risk."

Another initiative is AI4People: It is a multi-stakeholder forum that "brings together all stakeholders interested in shaping the societal impact of AI—including the European Commission, the European Parliament, civil society organizations, industry and the media" [33]. The result is a living document with the following preamble: "We believe that, in order to create a Good AI Society, the ethical ... should be embedded in the default practices of AI. In particular, AI should be designed and developed in ways that decrease inequality and further social empowerment, with respect for human autonomy, and increase benefits that are shared by all, equitably. It is especially important that AI be explicable, as explicability is a critical tool to build public trust in, and understanding of, the technology."

The so-called Algo.Rules [34] define a new approach on how to promote software trust systematically. It was developed by the think tank iRights.Lab together with several experts in the field. New rules define how an algorithm must be designed in order to be able to be evaluated with moral authority: above all, transparent, comprehensible in its effects and controllable:

1. "Strengthen competency: The function and potential effects of an algorithmic system must be understood.
2. Define responsibilities: A natural or legal person must always be held responsible for the effects involved with the use of an algorithmic system.
3. Document goals and anticipated impact: The objectives and expected impact of the use of an algorithmic system must be documented and assessed prior to implementation.
4. Guarantee security: The security of an algorithmic system must be tested before and during its implementation.
5. Provide labelling: The use of an algorithmic system must be identified as such.
6. Ensure intelligibility: The decision-making processes within an algorithmic system must always be comprehensible.
7. Safeguard manageability: An algorithmic system must be manageable throughout the lifetime of its use.
8. Monitor impact: The effects of an algorithmic system must be reviewed on a regular basis.
9. Establish complaint mechanisms: If an algorithmic system results in a questionable decision or a decision that affects an individual's rights, it must be possible to request an explanation and file a complaint."

5 Outlook

Software engineering has changed dramatically since 1968, when it was coined for the first time as an engineering discipline [35]. According to [25]), software engineering has been in the Structured Methods Era 1960–1980 and in the Object Methods Era 1980–2000 and is currently in the Agile Methods Era. These eras not only brought "method wars" and "zig-zag-paths" to software engineering, but also put the focus on technical aspects, software features and methodological approaches rather than putting it on the societal impact of software. In fact, along recent digital transformation discourses, not only the central role of software became apparent to the public, but also the need to find a new framing for software engineering. This framing is coined to be "*responsible software engineering*" in this chapter. It is constituted by five central elements, which are sustainability by design performed by people in power, techno-social responsibility by the software communities, responsible technology development by the society, state-of-the-art software engineering within every software project and the Weizenbaumian oath for all experts.

Responsible software engineering is anticipated by several initiatives that arose from discussions around professional ethics in the software communities specifically as well as by addressing grand challenges in research and innovation in general. It will take time till wide-spread acceptance and deployment, but we need to take actions now and develop approaches and programs which are taught at universities and in industry. Along digital transformation, software and its engineering became public goods and have to be addressed and coped with appropriately. It is not any longer a niche concern, but it is in the interest of us all to design and develop the software also on the basis of a public discourse. In this view, software quality is to be extended along societal impact, transparency, fairness and trustworthiness, which will require not only new or extended methods and tools, but also updated processes and regulations.

Acknowledgments This work has been partially funded by the Federal Ministry of Education and Research of Germany (BMBF) under grant no. 16DII111 ("Deutsches Internet-Institut", Weizenbaum-Institute for the Networked Society) as well as by the German Federal Ministry of Education and Research and the Federal Ministry for the Environment, Nature Conservation and Nuclear Safety under grant number 01RIO708A4 ("German Advisory Council on Global Change", WBGU).

The author thanks the numerous discussions with Stefan Ullrich, Jacob Kröger, Andrea Hamm, Hans-Christian Gräfe, Diana Serbanescu, Gunay Kazimzade and Martin Schüssler all from Weizenbaum-Institute as well as with Reinhard Messerschmidt, Nora Wegener, Marcel J. Dorsch, Dirk Messner and Sabine Schlacke at WBGU.

Last but not least, the author thanks the iSQI team for years of successful and pleasant cooperation to make software quality more present and to offer numerous software quality training schemes that improve the knowledge and expertise in the field. Congrats on its 15th birthday, wishing iSQI at least another 15 successful years of extending the body of knowledge in software quality.

References

1. Unit, E.I.: The growing €1 trillion economic impact of software. (2018)
2. Russo, D., Ciancarini, P.: A proposal for an antifragile software manifesto. Procedia Comput. Sci. **83**, 982–987 (2016)
3. Russo, D., Ciancarini, P.: Towards antifragile software architectures. Procedia Comput. Sci. **109**, 929–934 (2017)
4. Osterweil, L.J.: What is software? The role of empirical methods in answering the question. In: Münch, J., Schmid, K. (eds.) Perspectives on the Future of Software Engineering, pp. 237–254. Springer, Berlin (2013)
5. Parnas, D.L., Madey, J.: Functional documents for computer systems. Sci. Comput. Program. **25**(1), 41–61 (1995)
6. ISO/IEC: Information technology — Vocabulary. 2382. (2015)
7. Xia, W., Wen, Y., Foh, C.H., Niyato, D., Xie, H.: A survey on software-defined networking. IEEE Commun. Surveys Tuts. **17**(1), 27–51 (2014)
8. Myers, B.A., Rosson, M.B.: Survey on user interface programming. Paper presented at the Proceedings of the SIGCHI conference on Human factors in computing systems (1992)
9. Forward, A., & Lethbridge, T.C.: The relevance of software documentation, tools and technologies: a survey. Paper presented at the Proceedings of the 2002 ACM symposium on document engineering (2002)
10. Gordieiev, O., Kharchenko, V., Fominykh, N., Sklyar, V.: Evolution of software quality models in context of the standard ISO 25010. Paper presented at the proceedings of the ninth international conference on dependability and complex systems DepCoS-RELCOMEX, Brunów, Poland, 30 June–4 July 2014 (2014)
11. Boehm, B.W., Brown, J.R., Lipow, M.: Quantitative evaluation of software quality. Paper presented at the Proceedings of the 2nd international conference on Software engineering, San Francisco, California, USA (1976)
12. Grady, R.B., Caswell, D.L.: Software Metrics: Establishing a Company-wide Program. Prentice-Hall, Englewood Cliffs (1987)
13. IEEE: Standard for Software Maintenance. (Std 1219). (1993)
14. Dromey, R.G.: A model for software product quality. IEEE Trans. Softw. Eng. **21**(2), 146–162 (1995)
15. Hyatt, L.E., Rosenberg, L.H.: A software quality model and metrics for identifying project risks and assessing software quality. Paper presented at the product assurance symposium and software product assurance workshop (1996)
16. Seffah, A., Metzker, E.: The obstacles and myths of usability and software engineering. Commun. ACM. **47**(12), 71–76 (2004)
17. Wang, R.Y., Strong, D.M.: Beyond accuracy: what data quality means to data consumers. J. Manag. Inf. Syst. **12**(4), 5–33 (1996)
18. Pipino, L.L., Lee, Y.W., Wang, R.Y.: Data quality assessment. Commun. ACM. **45**(4), 211–218 (2002)
19. Di Gironimo, G., Lanzotti, A., Vanacore, A.: Concept design for quality in virtual environment. Comput. Graph. **30**(6), 1011–1019 (2006)
20. Klinker, G., Stricker, D., Reiners, D.: Augmented reality: a balance act between high quality and real-time constraints. In: Ohta, Y., Tamura, H. (eds.) Mixed Reality: Merging Real and Virtual Worlds, pp. 325–346. Springer, Berlin (1999)
21. Serrano, M., do Prado Leite, J.C.S.: Capturing transparency-related requirements patterns through argumentation. Paper presented at the 2011 first international workshop on requirements patterns (2011)
22. Schieferdecker, I., Messner, D.: The digitalised sustainability society. Germany and the World 2030. (2018)
23. WBGU: Our common digital future. German Advisory Council on Global Change, Berlin (2019)

24. Brey, P.: Values in technology and disclosive computer ethic. In: Floridi, L. (ed.) The Cambridge Handbook of Information and Computer Ethics, pp. 41–58. Cambridge University Press, Cambridge, MA (2010)
25. Jacobson, I., Stimson, R.: Escaping method prison – On the road to real software engineering. In: Gruhn, V., Striemer, R. (eds.) The Essence of Software Engineering, pp. 37–58. Springer, Cham (2018)
26. Porter, M.E., Kramer, M.R.: The link between competitive advantage and corporate social responsibility. Harv. Bus. Rev. **84**(12), 78–92 (2006)
27. Biermann, F., Kanie, N., Kim, R.E.: Global governance by goal-setting: the novel approach of the UN Sustainable Development Goals. Curr. Opin. Environ. Sustain. **26**, 26–31 (2017)
28. Weizenbaum, J.: On the impact of the computer on society: how does one insult a machine? In: Weckert, J. (ed.) Computer Ethics, pp. 25–30. Routledge, London (2017)
29. Strazdiņa, L., Arnicane, V., Arnicans, G., Bičevskis, J., Borzovs, J., Kuļešovs, I.: What software test approaches, methods, and techniques are actually used in software industry? (2018)
30. ACM: ACM code of ethics and professional conduct. Association for Computing Machinery's Committee on Professional Ethics (2018)
31. GI: Unsere ethischen Leitlinien, p. 12. Gesellschaft für Informatik, Bonn (2018)
32. EC: Ethics Guidelines for Trustworthy AI, p. 41. European Commission High-Level Expert Group On Artificial Intelligence, Brüssel (2019)
33. Floridi, L., Cowls, J., Beltrametti, M., Chatila, R., Chazerand, P., Dignum, V., Rossi, F., et al.: AI4People—An ethical framework for a good ai society: opportunities, risks, principles, and recommendations. Mind. Mach. **28**(4), 689–707 (2018)
34. iRights.Lab: Algo.Rules: Regeln für die Gestaltung algorithmischer Systeme. Retrieved from Gütersloh, Berlin. https://www.bertelsmann-stiftung.de/fileadmin/files/BSt/Publikationen/GrauePublikationen/Algo.Rules_DE.pdf (2019)
35. Naur, P., Randell, B.: Software engineering-report on a conference sponsored by the NATO Science Committee Garimisch, Germany. https://carld.github.io/2017/07/30/nato-software-engineering-1968.html (1968)

Chasing Mutants

Adam Leon Smith

Abstract This chapter describes mutation testing, how it has developed, the types of tools associated with it, the benefits to a quality assurance process, and the associated challenges with scaling it as a test design technique and as a consumer of resources.

Keywords Software testing · Software quality · Test automation · Mutation testing

1 Introduction

It's hard to prove a negative, and if you are reading this book, you are likely to be aware that testing proves the presence of defects, rather than the absence. Mutation testing turns this principle on its head and asks, if we know there are defects, what do our test results tell us about the quality of our software and tests?

As engineers increasingly turn to more automated software verification approaches and higher quality software outputs are demanded by ever-reducing release timelines, mutation testing helps us take a step back to assess whether we should really be so confident in our tests.

I asked Markus Schirp, author of Mutant [1], a mutation testing tool for Ruby, how he defined mutation testing:

> Mutation testing is the process of heuristically determining semantics of your program that are not covered by tests.—Markus Schirp

With most software testing approaches, it is hard to determine if you can detect failures in the testing process until the failures occur in a later testing activity, or

A. L. Smith
Dragonfly, London, UK

S. Goericke (ed.), *The Future of Software Quality Assurance*,
https://doi.org/10.1007/978-3-030-29509-7_12

worse, in a production environment. This is something familiar to every test manager as they deal with the learning from production releases and feeding information back into the test process. Surely there is a better way to find problems with test coverage?

Mutation Testing is a testing technique that can be traced back to 1971 [2], which has gained more and more attention, now with over a dozen associated tools and used in a range of software contexts. The number of tools available has grown significantly from less than 5 in 1981, to over 40 in 2013 [3].

Mutation Testing sits in a gray area of testing techniques that don't rely on formal specification of the desired behavior of the system, alongside fuzz testing, metamorphic testing, and arguably exploratory testing.

At its core, mutation testing is the practice of executing a test, or a set of tests, over many versions of the software under test—a so-called mutant. Each version of the software under test has different faults deliberately and programmatically injected. Each testing iteration is conducted on a slightly different version of the software under test, with different faults injected based on heuristics, or "rules of thumb" that correspond to common faults. These versions are referred to as mutants, in the context that they are small variations . The purpose of the testing is usually to determine which faults are detected by test procedures, resulting in failures.

The manufactured faulty versions of the software aren't called mutants because they are inferior, but because they are changed in the same way that human genetics mutate as part of natural evolution. This is similar to the use of genetic algorithms in artificial intelligence to solve search and optimization problems.

The benefits of mutation testing can be significant, it can give enormously useful insight into:

- The quality and coverage of automated tests, in particular, coverage of assertions and validations
- The coverage of testing in a particular area of the software
- The success of particular test design techniques
- The complexity and maintainability of different areas of code
- The traceability of software code or components to the overall business functionality, through the tests

It can be used as an approach to improve an existing set of tests, or as a tool to help build a new test suite, either to ensure it has sufficient validations, or to prioritize testing towards areas of the code. Ultimately, it provides a set of facts about quality, information that is not otherwise available (Fig. 1).

2 Automated Testing

Mutation Testing conceptually is not limited to automated tests; in principle it is an approach that could be applied to manual testing. However, the cost associated with running thousands of tests manually (on code that will never reach production, in order to improve the development and testing process) doesn't stack up against the benefit in most cases.

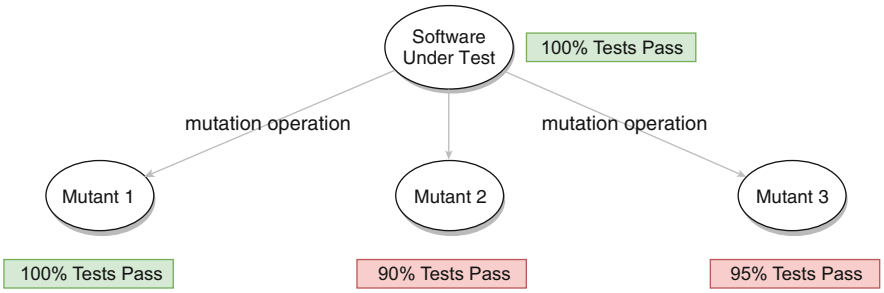


Fig. 1 In this example, Mutant 1 doesn't cause any tests to pass, indicating that the injected faults is not detected, hence there is a fault in the test coverage. Mutants 2 and 3 cause test failures, which is good!

One of the core challenges with automated testing is ensuring that there is the right balance of assertions and validation checks to ensure that software is working, without making the test suite brittle to irrelevant structural changes in the software. Although usually slower, we know that human testers find more defects in software. We know that the reason for this is that the human tester has more context, and uses experience and heuristics to determine correct behavior as well as the stated assertions in any testing procedure. Unlike with manual testing, it is a given that automated test suites are by nature, only going to detect defects within the bounds of the validations they have been setup to perform—in other words, they will only check the things they have been told to check.

In the most extreme example, it is possible to create an automated test that starts the software under test and validates it appears to execute successfully, without making any assertions about the program output. This would be a very maintainable test, as it would rarely need to be updated to stay in line with the software, but a very low value test, as it tells us little other than that the program executes. If this hypothetical software and test were to be evaluated using mutation testing, the score would be very low, as mutated versions of the software would only fail the testing if the injected fault caused a full execution failure.

Taking a counter-example, an automated test against a website that compares every single accessible element of the HTTP responses against a baseline copy. This would be a very unstable test, as it would fail at even the smallest change, perhaps one that a real user or tester could not perceive. If this software and test were evaluated with mutation testing, the score would be extremely high, as virtually any change would cause a test failure.

Mutation testing does not solve the problem of the maintainability of test automation, but it does give useful insight into the value actually offered by individual tests.

This is also important because even in an environment with a high level of automation, exhaustive or full combinatorial testing is often not performed because it is considered an impractical use of resources.

One risk with any test design technique is test case explosion—that is, when you focus on a specific test design technique, and the volume of tests dramatically

increases. This might be because you are just starting to apply format test design techniques, or it might be because you are focusing too much on one technique.

Mutation testing information can help you prioritize resources to maximize test coverage in new ways.

3 Code Coverage and Mutation Operators

On the one hand, mutation testing is a black-box testing technique, as the design and execution of the tests do not require knowledge of the inner workings of the code. That said, the mutation operators and the fault injection process must have very detailed knowledge of the code, or at least, a mutable derivation of it. In fact, the set of mutants generated is inextricably linked to the language and structure of the underlying code, but then again so are real faults.

When the mutation "operator" injects faults into the code, it performs rule-based changes, which can usually be customized by the engineers working on the code. Some example of these include:

- Deleting statements from the code
- Inserting statements from code
- Altering conditions in code
- Replacing variable values

Let's go over some basic programming concepts, and how they link to faults, code coverage and mutation testing. In most programming languages, an executable statement expresses an action to be carried out, such as assigning a value, e.g. true or false, to a variable. The ISTQB glossary defines this as: "A statement which, when compiled, is translated into object code, and which will be executed procedurally when the program is running and may perform an action on data."

For example, in the code below, the statements all of the lines that don't start IF/ELSE are statements:

Pseudo Code

```
allowEntrance = false

if (customerHasMembershipCard or customerHasAccessCard):
allowEntrance = true
    price = 0
else:
   if(weekend):
      allowEntrance = true
      price = 10
```

The degree to which statements are covered by an executable test suite is usually described as statement coverage. Statement coverage is the percentage of executable statements that have been exercised by a test suite.

$$\text{Statement Coverage} = \frac{\text{Number of Executed Statements}}{\text{Total Statements}}$$

To reach full statement coverage on the code above you would need two tests, as there are two exclusive paths through the code required to execute each statement.

Another coverage approach is to cover each branch or decision, essentially wherever a conditional statement such as if, for, while is included, ensure both outcomes of the conditional statement are evaluated. To reach full branch coverage on the example above, you would need one further test, which covered whether the weekend variable was true or false.

$$\text{Branch Coverage} = \frac{\text{Number of Executed Branches}}{\text{Total Branches}}$$

Finally, condition coverage, is a code coverage metric which measures whether each individual condition has been evaluated to true or false. This can be calculated as:

$$\text{Condition Coverage} = \frac{\text{Number of Executed Operands}}{\text{Total Operands}}$$

So referring again to the example above, ensuring that tests cover both membership and access card scenarios, adding one more test to our growing suite.

The problem with solely using code coverage metrics to measure the quality of automated tests is that none of the metrics evaluate whether my tests actually check whether the customer is allowed access, or how much the software calculates as a charge. The verification of these states and variables are not included in the metrics.

Measuring code coverage on your automated tests is great, but it is only part of the picture. Code coverage only tells you the logic and branches which have been executed, it doesn't really measure whether your tests are getting a lot of data of functional coverage, and it doesn't tell you whether your tests are effectively detecting failures.

Validation of the system response (effectively comparing actual to expected results) is a critical part of implementing automated testing, it is straightforward to check that a variable is sensible; however, as interfaces get more complex (think of an XML message or a user interface) the amount of design subjectivity around the validations increases.

Mutation operation engines, which apply mutation operators to code, vary in the operators they support, and indeed the user can usually configure how these are applied. As an illustration, the well-known Mothra study [4] and supporting tool used the operators on Fortran shown in Table 1. These are somewhat outdated, as operators have evolved with the development of object-orientated techniques [5]. Of

Table 1 Mothra operators

Type	Description
aar	Array for array replacement
abs	Absolute value insertion
acr	Array constant replacement
aor	Arithmetic operator replacement
asr	Array for variable replacement
car	Constant for array replacement
cnr	Comparable array replacement
csr	Constant for scalar replacement
der	DO statement end replacement
lcr	Logical connector replacement
ror	Relational operator replacement
sar	Scalar for array replacement
scr	Scalar for constant replacement
svr	Scalar variable replacement
crp	Constant replacement
dsa	Data statement alterations
glr	Goto label replacement
rsr	Return statement replacement
san	Statement analysis
sdl	Statement deletion
src	Source constant replacement
uoi	Unary operation insertion

course, the volume of potential operators and their resulting mutations of software code is enormous, and no implementation can be viewed as exhaustive.

Running the software under test through a mutation operator could result in a number of changes to the code. Four example, rule-based operators are applied to this code:

Pseudo Code

```
allowEntrance = false

if (customerHasMembershipCard or customerHasAccessCard):
   allowEntrance = true
   price = 0
else:
   if(weekend):
      allowEntrance = true
      price = 10
```

In this example, the red items will be changed. The first will be changed to be initialized as **true**, the second will change an **or** to an **and**, the third and fourth will negate a Boolean value by adding a **not**.

As a result, four versions of the system under test will be compiled, and the mutation testing routine should run all automated tests against each version. In order to successfully detect each of the mutants, the test suite would need the following characteristics:

- To test that the customer was not permitted access if the customer did not have either card, and it was not a weekend. This would test a failure to initialize the variable in mutation 1.
- To test where a customer has one card, but not the other (full condition coverage would also require this), and then validate the customer was permitted access.
- To test that a customer was allowed access without cards on a weekend.
- Replacing variable values

As you can see, although a test suite built to achieve code coverage would exercise similar paths through the code, mutation testing metrics allow much more specificity about the verification the tests should perform.

This is useful because, unsurprisingly, many software faults are introduced in the coding process. For example, the common off-by-one error, where a programmer instructs a loop to iterate one time to many, or too few—or miscalculates a boundary condition—is directly addressed by test design techniques such as boundary value analysis and equivalence partitioning. Similarly, this kind of error is commonly injected by mutation test operators.

The mutation testing process can be summarized as follows:

1. First, mutants are created, by inserting errors.
2. After they have been created, the tests are selected and executed.
3. The mutants will be "killed" if the tests fail when executed against the mutant.
4. If the result of testing the mutant is the same as with the base software, the mutant has "survived."
5. New tests can be added, existing tests amended, or code refactored—in order to increase the number of "killed" mutants. Some mutations cannot ever be detected, as they produce equivalent output to the original software under test, these are called "equivalent mutants."

Once the full process has been executed, a mutation score can be calculated. This is the ratio of killed mutants, to the total number of mutants. The closer this score is to 1, the higher the quality of the testing suite and the software.

$$\text{Mutation Score} = \frac{\text{Number of Killed Mutants}}{\text{Total Mutants}}$$

4 Mutation Testing Challenges and Strategies

Mutation testing is inherently scalable as it is usually based on doing something you already do automatically, many times over and independently. That said, this should also be true of automated integration and system testing, but is often not.

The resources required to conduct mutation testing can be significant, clogging up the continuous integration pipeline for hours to run endless iterations of tests on endless iterations of mutated software. Additionally the time required to go through and fix all the issues uncovered, can be daunting.

Three distinct areas of cost need to be considered:

- The compile-time cost of generating mutants
- The run-time cost of running tests on the mutants
- The human cost of analyzing results

Compile Time
As mentioned previously, a major problem with mutation testing is the cost of execution. The number of mutants is a product of the number of lines of code and the number of data objects, but as a rule of thumb the number of generated mutants is typically in the order of the square of the number of lines of code. Some strategies [6] have been tried to reduce the amount of execution:

- Sampling—Executing only a random sample of mutants across a logical area of software and associated tests.
- Clustering—Using unsupervised machine learning algorithms (e.g., K-means) to select mutants.
- Selective Testing—Reducing the number of mutation operators, that is, the heuristics used to inject faults, can reduce the number of mutants by 60
- Higher-order Mutation—First-order mutants are those that have had a single fault injection; second-order mutants have been injected with multiple faults, in multiple iterations of mutation. Higher-order mutants are harder to kill, and focusing solely on second-order mutants has been shown to lead to reduced effort, without reducing coverage.
- Incremental Mutation—Mutating only new or changed code, rather than the whole code base under test.

We know that defects cluster in the same areas of code, and perhaps a simple strategy of applying the techniques to limited, complex, high-risk, and defect-ridden areas of functionality can offer an appropriate balance of cost vs benefit.

Conversely, there are statements in code that we not be concerned about. For example, excluding all logging statements from mutation may be appropriate and lead to less mutants to test.

Another approach used to reduce the amount of time and resources required is direct integration with the compiler. Early mutation testing approaches compiled each mutant individually; however, more modern approaches compile once, and then mutate an intermediary form such as bytecode. This has significant benefits in

terms of compilation performance, but no benefits in terms of the execution time of evaluating each mutant.

Much research has been conducted into one concept called "Weak Mutation." This effectively evaluates mutants at a component level, rather than fully executing each mutant. This has been found to be nearly as effective as "Strong" mutation with reduced runtime cost [6].

Runtime

The runtime cost of running mutation testing cycles can be significant. Of course, the execution can be scaled horizontally in most cases, as tests can be run across multiple machines, or it can be scaled vertically by using more powerful machines. To apply horizontal scalability it is necessary to consider this when selecting the appropriate mutation testing tools, with support for such approaches.

Classical test automation techniques for improving the run-time of tests also need to be considered. Hard-coded waits built into tests may not be noticeable when the test is run once, but scaled to hundreds or thousands of executions, become a significant cost. Optimization of automated tests for performance purposes should be performed *before* attempting to introduce mutation testing.

Analysis

Two related areas of challenge are the Oracle Problem, and the problem of reducing equivalent mutants. The Oracle Problem is far from unique to mutation testing, and applies to any area of testing where it is difficult to ascertain whether a test has succeeded.

This can occur when mutants can't be killed, because the assertions or validations required to be implemented in the tests are too difficult to implement.

It can also occur when software is less deterministic, and when it is difficult to understand whether the failure to detect a mutation is actually meaningful. I asked Alex Denisov, author of a mutation testing tool called Mull [7], what he thought the biggest challenges were:

> As a user, the biggest issue so far is the test oracle problem. It is not a big deal on small or medium projects. However, for a big project, mutation can produce thousands of mutants, hundreds of which are surviving. It is not yet clear how a developer is supposed to process them—manually reviewing those hundreds is simply impractical.—Alex Denisov

The problem of reducing equivalent mutants is also relevant, that is, the mutation does not result in observable change to the output or execution. This can occur because of dead code that isn't used; the mutation only changes the speed of the software; or, the mutation only changes internally used data that does not affect the end state or output.

I asked Markus how he optimizes mutation testing to reduce equivalent mutants, and he disagreed with the metric:

> This cannot be answered as is, the reality is more difficult. The question implies that equivalent mutations are a real world problem, but they are not. The question also implies the tool produces mutations of "low informational value" and not every mutation that comes back alive contains value for the human who reads a report. Both assumptions are false in my world. Equivalent mutants almost never show up, and low value mutations do not exist, hence everything must be killed.—Markus Schirp

To understand what Markus meant, let's look at an equivalent mutation:

Pseudo Code

```
def foo(i):
   return  i + 1 + 0
```

An example of a potentially equivalent mutant would be to remove the "+ 0". While this would change the software under testing, it doesn't change the output, as adding zero to a number has no actual effect anyway. That is exactly the point, the code is irrelevant and should be removed.

5 Mutation Testing Tools

Mutation testing tools are numerous in nature, this is in part, because they are intrinsically linked with the programming languages used in the implementation. You can't generally use a tool designed for C++ with Java. It is crucial to pick a tool that supports the underlying technology stack, has support to allow you to configure the mutation heuristics you want, integrates with your development environment well, and supports parallel and distributed execution. Wikipedia [8] has a good list of mutation engines that you can use with your code. Getting started with these tools can be as simple as injecting a dependency into your build configuration.

Selecting and evaluating tools that suit you is actually more important than with traditional tool selection. One reason is that the tools will come with different default operators and strategies built in. As outlined earlier, these operators and strategies effectively dictate the technical approach and directly affect the results. The tool also may determine how you can scale the run-time execution, and choosing a tool with such limited support may result in impractical execution timelines.

Tools and libraries also exist for mutating data structures rather than code, for instance, applying heuristic operators against standard data structures such as XML.

However it isn't just the core mutation engine that you need to consider:

> As an implementor, the bigger problem I see is the smooth integration. How to include the mutation testing into existing infrastructure: various build systems, test frameworks, CI pipelines, IDEs, etc.—Alex Denisov

There are other types of tools that you might want to look at with regard to your overall workflow. For instance, Pitest, a Java mutation engine, has a Cucumber plugin [9] that allows you to integrate with Cucumber and Behavior-Driven Development. Similarly, SonarQube [10], the popular code quality monitoring suite, has plugins that allow you to display detailed results from your mutation test run. Some mutation testing engines also offer IDE plugins that accelerate the feedback loop and allow you to see the results in the context of your code.

6 Other Applications of Mutation Testing

One way to look at scaling mutation testing is in the test level. It is far easier to apply mutation testing at a unit test level, because individual mutations can be intuitively tracked to effects in the software's operation, less resources are required, and the start-up and execution time for each test is much smaller, leading to faster results.

Mutation testing can still be applied at higher test levels such as functional system testing. Mutated software versions can be deployed to web servers, and typical tools like Selenium can be used to run the test iterations. However the execution time, and amount of setup work, will be considerably higher. Investigating and resolving issues can also require more work to link the mutated line of code back to a functional test verification.

The concept of mutation testing has been proposed to cover other areas of software quality as well. For example, it has been proposed, at least in research [11], that mutation can be used on top of formal specification languages to detect defects at the design stage.

Mutation testing can also be used to inject faults in the running environment. Chaos Monkey, a tool originally developed by Netflix [12] to test the resilience of their environments, randomly injects infrastructure faults, to see how the application under test (or live application!) handles the failure. This can be viewed as mutating the environment the software runs in:

> Failures happen, and they inevitably happen when least desired. If your application can't tolerate a system failure would you rather find out by being paged at 3am or after you are in the office having already had your morning coffee?—Chaos Monkey on github

Fuzz testing, which mutates the input domain of an application, can also be considered a type of mutation testing. Instead of mutating the program code, it mutates inputs to the system under test. This has completely different goals, though, to the mutation testing described above. It is primarily aimed at detecting how your system will behave with unexpected inputs; this might be carried out as part of a security testing activity, or a negative functional test.

Finally, mutation testing can be used to understand the properties of an unknown code base:

> mutation generation, without running the kill phase, allows a nice and unbiased detection of complex code structures.—Markus Schirp

This approach can be used to prioritize regression testing, or refactoring of a large code base.

While mutation testing has been around for a while, it is solidly building support in the engineering community, and it clearly delivers useful information it just isn't possible to get elsewhere.

7 Conclusion

This chapter has hopefully opened your eyes to a completely different approach to looking at test coverage. While mutation testing is something that is currently applied most frequently at the unit testing level, the concepts can be applied, and the benefits realized throughout a full set of software quality assurance practices.

It is not only an effective way to assess your automated tests, but also a way to understand the complexity of your code, and quantitatively understand your code quality and test coverage.

The concepts can also be applied to code, input data, the environment, and no doubt other technical artifacts.

As explained, the biggest challenges are the compute resources required to execute a large number of tests on a large number of mutants, and also the human resources required to analyze equivalent mutants, and solve oracle problems. Finding the right balance between coverage confidence and resource requirements is crucial. Limited research exists, which covers the financial and quality benefits and costs of mutation testing outside of research, and this is clearly an area that needs more analysis.

Maybe it can't help you refactor all your code or improve all your tests, but it can definitely point you in the right direction. No software quality assurance specialist needs less information, and no test suite can ever be fully understood in terms of coverage without some execution results. Mutation testing is no silver bullet, but cannot be ignored by true quality professionals.

Acknowledgements I would like to thank the following people their help: `Markus Schirp` for his advice and quotes as author of the mutation testing tool Mutant; and, `Alex Denisov` from lowlevelbits.org, for his quotes as a user of mutation testing and author of the tool Mull; and, `My better half Julia` for letting me disappear into writing mode on weekends;

References

1. Mutant: Mutation testing for Ruby https://github.com/mbj/mutant
2. Lipton, R.: Fault Diagnosis of Computer Programs, student report, Carnegie Mellon University (1971)
3. Saxena, R., Singhal, A.: A critical review of mutation testing technique and hurdles, in 2017 International Conference on Computing, Communication and Automation (ICCCA), Greater Noida, pp. 887–892 (2017)
4. DeMillo, R.A., Guindi, D.S., McCracken, W.M., Offutt, A.J., King, K.N.: An extended overview of the mothra software testing environment. In: IEEE Proceedings of Second Workshop on Software Testing, Verification, and Analysis, July 1988, pp. 142–151
5. Kim, S., Clark, J.A., McDermid, J.A.: Class mutation: mutation testing for object-oriented programs, p. 15
6. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. **37**(5), 649–678 (2011)
7. Mull: Mull is a LLVM-based tool for Mutation Testing with a strong focus on C and C++ languages. https://github.com/mull-project/mull
8. Mutation Testing: Wikipedia. 03-Mar-2019. https://en.wikipedia.org/wiki/Mutation_testing
9. Pitest Cucumber Plugin: https://github.com/alexvictoor/pitest-cucumber-plugin
10. SonarQube: https://www.sonarqube.org
11. Sugeta, T., Maldonado, J.C., Wong, W.E.: Mutation testing applied to validate SDL specifications. In: Proceedings of the 16th IFIP International Conference on Testing of Communication Systems, Mar. 2004, p. 2741
12. Chaos Monkey github site: https://github.com/Netflix/SimianArmy/wiki/Chaos-Monkey

Embracing Quality with Design Thinking

Mark F. Tannian

Abstract Developed outcomes that depend upon software can be of such quality that end users do not merely accept the product or service, they embrace it. The product or service is of such a quality that end users are excited, inspired, motivated, committed, or possibly relieved to be using the software directly or indirectly (i.e., an embedded component). Achieving embracing quality requires understanding user needs and desires as well as their environmental contexts and accommodating these understandings within the development practice. Design thinking is one approach to deliver products and services grounded in a user-informed development process.

Keywords Human-centered design · User-centered design · Design · Design thinking · Embracing quality · Quality · Prototype · Iterative design · Convergence quality · Collaboration

1 Introduction

Developed outcomes that depend upon software can be of such quality that end users do not merely accept the product or service, they embrace it. The product or service is of such a quality that end users are excited, inspired, motivated, committed, or possibly relieved to be using the software directly or indirectly (i.e., an embedded component). Moreover, the software development outcome has enabled end users to possibly do new things, derive pleasure from owning and using it, or is able to perform personally significant tasks more easily. The level of satisfaction is such that they have an affinity for what your team has produced. Think about those things you have and the activities you do that instill happiness, are fun, or improve your personal and work life. Would you say that is a quality worth achieving? This quality is being called *embracing* for this discussion.

M. F. Tannian
RBCS, Inc., Bulverde, TX, USA

S. Goericke (ed.), *The Future of Software Quality Assurance*,
https://doi.org/10.1007/978-3-030-29509-7_13

Achieving embracing quality requires understanding user needs and desires as well as their environmental contexts, and accommodating these understandings within the development practice. Quality function deployment (QFD) introduced by Shigeru Mizuno and Yoji Akao sensitized designers to the needs of the customer, and this practice helped introduce the term "voice of the customer." The QFD approach brings customer satisfaction forward early into manufacturing or development processes [1]. Another approach to understanding the user is inspired and informed by the design community. This approach is called "design thinking."

In the next section, a brief expansion on embracing quality is presented. The following section will introduce design thinking. The fourth section will describe design thinking's role in bringing about embracing quality. The fifth section raises potential challenges related to design thinking. The final section will end the chapter with some concluding thoughts.

2 Embracing Quality

Embracing quality is a convergence quality that emerges from other emergent qualities, such as reliability, security, privacy, usability, and performance. For our purposes, a convergence quality is a quality that results from the integration of the user with the product or service. The user completes the technology and the resulting combination achieves desired value. The synergy between technology and user is such that the user finds great satisfaction, enjoyment, and may even experience flow [2] from using this technology. This synergy may allow the user to function as if the technology were an extension of her or him.

Embracing as a quality is not static and the level of this quality achieved may not be universal within a market. Culture strongly influences taste, aesthetics, personal values as well as the verbal and nonverbal means by which we communicate. A product or service that achieves embracing quality within a market may lose it over time. Certainly the opposite may happen where a product achieves greater embracing quality in the future. Product and service qualities such as reliability, fitness-for-purpose, security, usability, and performance change due to development practices and external factors, such as market changes, cultural sensitivities, and security threat landscapes. These are significant reasons for why innovation and quality assurance are necessary to achieve embracing quality. This chapter will not explore how to measure or define the gradations within embracing. Suffice it to say that embracing is not strictly a binary value (i.e., rejected or embraced). Some examples of products and services that have achieved embracing at one point in their history are Word Perfect, Sonic the Hedgehog, Apple iPhone, and Uber.

Embracing quality is an outcome that is dependent upon a product or service exhibiting sufficient levels of product or service quality as deemed important by users. ISO 25010:2011 calls out the following as system or software product quality dimensions: functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability, and portability [3]. Figure 1 depicts an alterna-

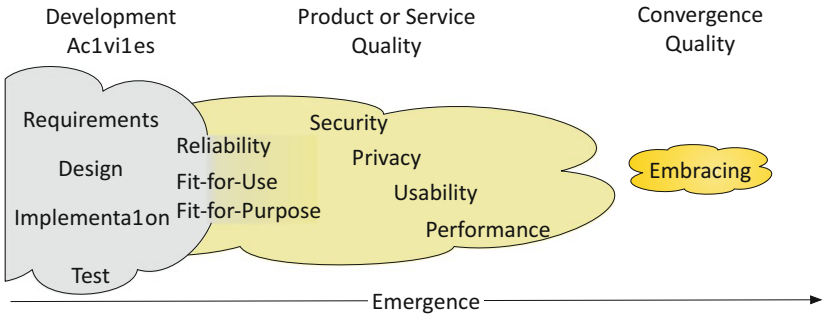


Fig. 1 Relationship of embracing quality to development activities

tive progression of qualities as they result from development activities. Admittedly Fig. 1 is not exhaustive in presenting all software or system qualities, an abbreviated collection was chosen for the sake of visual clarity and space. Figure 1 suggests that development activities primarily focus on reliability, fitness-for-use, and fitness-for-purpose. On some projects, requirements are written and development efforts are allocated to address aspects of the qualities of security, privacy, usability, and performance; however, the cohesive and comprehensive sense of these qualities are emergent. A product or service is unlikely to succeed without sufficient reliability, fitness-for-use, and fitness-for-purpose. These are immediate and necessary qualities to address by any development team. The requirements and efforts to achieve these qualities in turn influence the achievement of overall security, privacy, usability, and performance. The complex interplay among qualities, such as security, usability, and performance, further complicate achieving these quality dimensions.

Social dynamics such as peer pressure, status seeking, opinion influencers, and user ratings influence initial and ongoing interest in a product or service. Product managers and development teams must seek to achieve adequate levels of and balance among product or service qualities (e.g., security, privacy, usability, etc.); however, embracing quality is not completely in their control to achieve.

The "lean" software and business development concepts of minimal marketable feature (MMF) and minimal viable product (MVP) are by their definition minimizations of time, resources, and investment. Teams working towards these optimal outcomes are essentially seeking the tipping point of what is minimally marketable and minimally viable and what would be insufficient. This tight margin between sufficiency and insufficiency increases the sensitivity of qualities like fitness-for-purpose to the judgment of those responsible for defining what is minimally marketable and minimally viable for the audience being anticipated. The "voice of the customer" is essential for these judgments to be grounded in actual expectations.

Although a team's efforts may be deep within the technology stack, each work product influences qualities such as security, privacy, usability, and performance that will in turn influence the user's experience. It is important for team members to recognize how each team's contribution fits into the overall design and what role

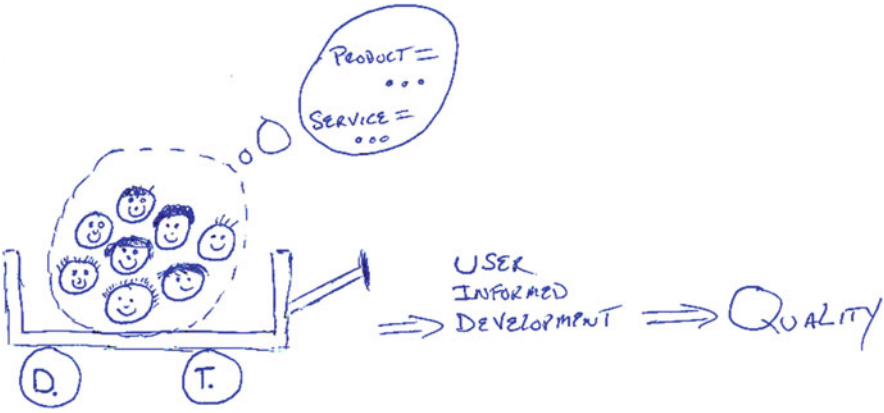


Fig. 2 Design thinking as a vehicle to achieve user-informed development and quality [in a visual thinking style]

the product or service plays in the end user's life. This understanding is needed to influence the numerous subtle decisions being made as they implement and test. Promoting user and overall solution awareness within development teams will assist members to recognize the impact of their efforts (Fig. 2).

Design thinking is a promising method for delivering products and services grounded in a user-informed development process. A brief introduction to design thinking is provided next.

3 Introducing Design Thinking

There are a number of books available that explore design thinking, such as [4, 5]. After reading Nigel Cross's *Design Thinking* [6] and Jeanne Liedtka and Tim Ogilvie's *Designing for Growth: A Design Thinking Tool Kit for Managers* [4], you may wonder what design thinking really means. There are two schools of thought that have adopted the term "design thinking." Cross is a member of a discipline of inquiry that explores design and how designers (e.g., architects, industrial designers) do what they do. The second school attempts to provide a bridge between designerly thinking and successful business innovation. To be clear, design thinking in the remainder of this chapter is that which enables businesses to be successful innovators. An accessible introduction to design thinking is the *A4Q Design Thinking Foundation Level* syllabus [7].

3.1 Fundamental Concepts

There is no single approach to design thinking. However there are common aspects among the well-established approaches. Design thinking at its core is a user-

centered design approach. A critical objective is to understand users and their objectives, needs, and impediments as they relate to what is being designed. Design is iterative and eventually converges on a final design by repeating the pattern of learning-making-evaluating as well as the pattern of divergent thinking–convergent thinking. The design team implements multiple alternative designs and allows users to experience these alternatives. The team adopts user insights as the design progresses. Design thinking relies heavily on the team's ability to work, explore, fail, and learn together. An underlying assumption is that initial designs are somehow wrong. Each design alternative is a good guess, but only a guess to which the team should not be overly committed. The principle "fail faster, succeed sooner" (attributed to David Kelley) is inherent in design thinking [8]. Teams should recognize the limits of analysis and rely on experimentation to achieve understanding. These experiments utilize prototypes that progress in fidelity from paper drawings, Lego blocks, and pipe cleaners to functional product as the team approaches its goal of delivering a product or service design.

3.2 Design Thinking Resources

Innovation is necessary for nearly all businesses, but it is not guaranteed to occur for all. Fundamentally, innovation is finding a viable solution to a problem for the first time. A design team may not be the first to try. However, if they are successful, they will be the first to discover a previously unknown answer. Innovation using design thinking results from a clever fusion of resources. People, place, parts, and partnership are necessary for design thinking to thrive.

In practice, designing innovative products and services requires a team of designers. In this context, the term "designer" is a role, not a designation, resulting from training in the design fields. Each member should be an expert in relevant technologies or fields of study or areas of the business. Diversity among members enables the team to propose a variety of and identify potential in candidate solutions. Each member needs to possess or be willing to develop the qualities of being observant, empathetic, attentive, and humble. These qualities are essential for each team member to learn throughout the design process as well as for the team to form common understandings.

With language and speech being limited forms of communication, visual thinking is an essential skill when communicating concepts and relationships that linear-vocabulary-bound communication struggles to convey. Pictorial communication is less hindered by differences in personal history, culture, and language.

There will be mistakes and misunderstandings. It is important to "fail fast," learn, and adapt. The team should encourage thoughtful risk taking. Each member's contribution has the potential to tip the effort into a successful direction. The converse is also true in that suggestions may prove to send the effort astray. Therefore, the team needs to provide a safe supportive environment and be able to work together with shared purpose, flexibility, and urgency.

Where this work is performed is integral to the process. This space is ideally well lit and flexible in terms of furniture and work surfaces. The workspace should facilitate collaboration and immersion. While onsite team members are actively working on the project, they should work in this space in order to be present for informal and possibly spontaneous exchanges. Having a stable location allows the team to place various project artifacts (e.g., prototypes, charts, drawings) in proximity to each other. Working in this space allows team members to recall and dwell on previous outcomes as they progress further along in design development. Prototype assembly and evaluation often takes place in this room. Specialized equipment may require prototype subassemblies to be constructed elsewhere. This may be a place where cooperating users meet with design team members. Depending on which design thinking techniques are used, it is helpful at times to have a space in which to assemble a gallery of ideas for collaborating colleagues and users to explore and consider. Anchoring the gallery in the team's designated space will likely improve activity logistics.

There is a strong drive within design thinking to "do" or "make." This mindset reinforces "fail faster, succeed sooner." When one is exploring the unknown, multiple quick experiments may quickly yield useful signals that will guide the design to success. In order to make, the team needs parts, materials, tools and talent. The talent component is addressed in part when forming the team; however, certain specialties may not be in-house or are not available for extended commitments to the project. Given the need for speed, variety, and volume of prototypes, the fidelity of the prototypes change as the design progresses. Fidelity is a term that relates to a prototype's approximation of a final finished product or service. A low-fidelity prototype focuses on large conceptual questions and often consists of rough drawn ideas or simple three-dimensional mockups. There is quite a bit of engineering undone at low fidelity. As the team makes deeper commitments to design alternatives the degree of fidelity increases, which is reflected in the level of engineering investment and operational sophistication of the prototype.

The team must consider the tradeoff between sophistication and conceptual agility as fidelity increases. Design thinking's goal is a user-centered nuanced understanding and design, and often the result is not a market-ready product or service. The team may realize significant shortcomings exist as user consultations progress. Investments in sophistication that does not directly influence the user experience or their task objectives are unlikely good design thinking prototype features. In many cases, design thinking yields mature materialized or implemented user requirements, but not necessarily a final product or service. Manufacturing engineering and production-grade software development processes are expected to follow.

Parts and materials are often generic or are repurposed store-bought items. Large format paper, white boards, colored pens and markers, paints, stickers, sticky notes, pipe cleaners, and Lego blocks are often used at low fidelity. Software development may start earlier than other technical disciplines because of the flexibility of programing and computing platforms. The team may seek to test a final design under close to real-world conditions prior to submitting the design and closing the design-

thinking project. To that end, custom parts or subassemblies, such as enclosures and circuit boards, may be needed.

There are two essential partnership types. The first is related to the sponsoring organization. The design thinking team needs the support of the sponsoring organization in terms of budget, approach, schedule, and outcome expectations. The design thinking team is tailored to the project. The appropriate personnel may not initially work in the same business units or functions. The broader organization needs to be willing to direct talented individuals to the team. Failure to develop a viable design is a possibility the sponsor must accept. A well-run design-thinking project will yield intellectual capital in terms of market understanding, promising solution approaches, and awareness of technical challenges. If the sponsors remain committed to addressing the same problem or a redefinition of the problem, the knowledge gained has the potential of directing the next effort away from pitfalls and towards promising notions that otherwise would not be known.

The second type of partnership is between the design thinking team and cooperating users. Users are essential. They are the source of the signals that the team collects and analyzes. These signals will indicate what does not work, nudge the design towards what does, and expose realities of the problem that could not be discovered without them. Users must be candid, honest, and be of goodwill when communicating their impressions and insights. Depending upon the nature of what is being designed, the sponsoring organization and design team are placing significant trust in these users. This trust relates to the relevance and reliability of their feedback as well as their discretion. Most likely each user will sign a confidentiality agreement; however, enforcing it may be difficult. In some cases, the loss from a breach of confidentiality may be beyond any realistic compensation from a jury award or legal settlement.

3.3 Design Thinking Approaches

There are multiple documented approaches to design thinking. The Design Council believes there is no one ideal design thinking approach [9]. The primary reasons for this are that business environments are undergoing continual change that prevents an ideal approach from emerging, and the need for businesses to adapt design thinking to their business makes their practical design thinking approach unique. There are several well-known general design thinking approaches. The Design Council introduced the Double Diamond approach [10]. Stanford d.School introduced their 5-stage approach [11]. Liedtka and Ogilvie introduced the Designing for Growth approach [4]. Although each organization will likely approach design thinking differently, it is informative to explore established general approaches.

The Double Diamond approach developed by the Design Council has four stages. The diamond in the approach title refers to a visual metaphor that represents the natures of the four stages. The depiction in Fig. 3 provides a visual that expands upon the diamond metaphor. Each stage of the four stages of Discover, Define,

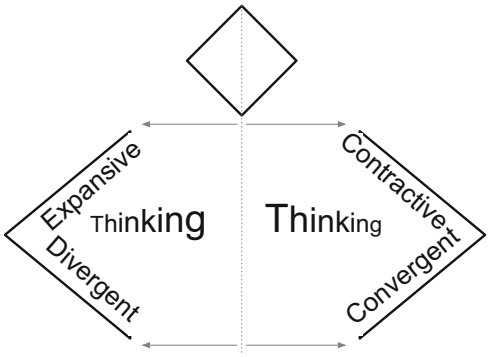


Fig. 3 Explanation of the diamond metaphor

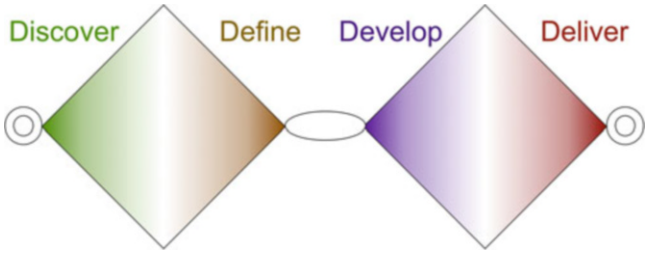


Fig. 4 Double Diamond approach

Develop, and Deliver conceptually align to each of the four left-right halves of two diamonds (Double Diamond) as shown in Fig. 4. The Discover stage is expansive in approach, seeking out and gathering what is known and knowable about the problem to be addressed by the intended innovation. After having assembled a broad range of inputs, the Define stage is engaged and the content is distilled and synthesized into a Design Brief. Informed by the Design Brief, the Develop stage uses expansive or divergent thinking in order to create, evaluate, and refine multiple design options. In the final stage, Deliver, a design is selected and is made ready for use. Design Council provides a collection of process methodologies to facilitate the completion of each stage.

The Stanford 5-Stage d.School approach is a bit fluid. At times practitioners of the Stanford approach introduce a sixth stage. Adaptability is an inherent spirit among those who apply Stanford's approach. One popular layout of the approach consists of the stages Empathize, Define, Ideate, Prototype, and Test [11]. An optional addition is the stage called Notice. This stage is oriented to identifying or noticing the initial problem to be explored before one can proceed to empathize, define, and so on. The overall flow through the stages is as previously listed. However, at a given time within a project it may be appropriate to revisit or jump ahead along the sequence of stages. Beyond describing the mechanisms with which to execute design thinking, Stanford d.School promotes the need for an appropriate

mindset among team members. The mindset animates their process and is useful to consider for adoption generally. Some of the mindset attributes are "human centered," "bias toward action," "radical collaboration," "show don't tell," and "mindful of process."

The last approach we will explore is Designing for Growth developed by Jeanne Liedtka and Tim Ogilvie [4]. This approach is designed to ask and answer four questions in dedicated stages. These questions are in order "What is?," "What if?," "What wows?," and "What works?." Their approach and their motivations for developing this process are closely aligned with the business need to generate new business value. Like the other two general processes, the starting point is the first stage listed and the successful stopping point occurs when "What works?" is truly completed. Iterations within and between these stages are likely necessary. In order to make these stages actionable, a set of ten tools have been described [4]. These in turn have been expanded upon by techniques that have been documented in a workbook [12]. Unlike the Double Diamond process, the result of this process is not a market-ready product or service. Liedtka and Ogilvie are committed to the need for learning up until the very end. The "What works?" ends with a limited and controlled market test that emulates practical conditions that may uncover significant challenges. A highly functional preproduction prototype is developed that enables the team to uncover flaws that need to be corrected prior to fully committing to a market or production-ready offering.

4 Design Thinking's Role in Quality

By using design thinking to focus on users and their experiences, the team has a means to gauge the design's potential along the embracing quality continuum. Iterating multiple design approaches through the make-learn-evaluate cycle with users provides the design team multiple opportunities to identify disagreeable aspects of designs, understand users' interests, calibrate outcome expectations, and prioritize promising elements of a design. By remaining agile and open to change, design thinking teams avoid prematurely committing to assumptions, understandings, and preferences that do not align with actual use, expectations, and users.

Design thinking directs development efforts to produce tangible design alternatives for user evaluations. Having a concrete representation of design ideas allows users to experience ideas; provides a common visible point of reference from which to offer and interpret feedback; and acts as a baseline from which to suggest revisions or alternatives. Figure 5 depicts active design thinking efforts as a central motivating force for development activities. Various quality objectives must remain unaddressed or limited in order to maintain speed and responsiveness. Many underlying infrastructure components up to the point of a premarket test will unlikely experience load and diverse usage patterns that deviate from guided user evaluation session objectives. The limited range of usage allows for various quality

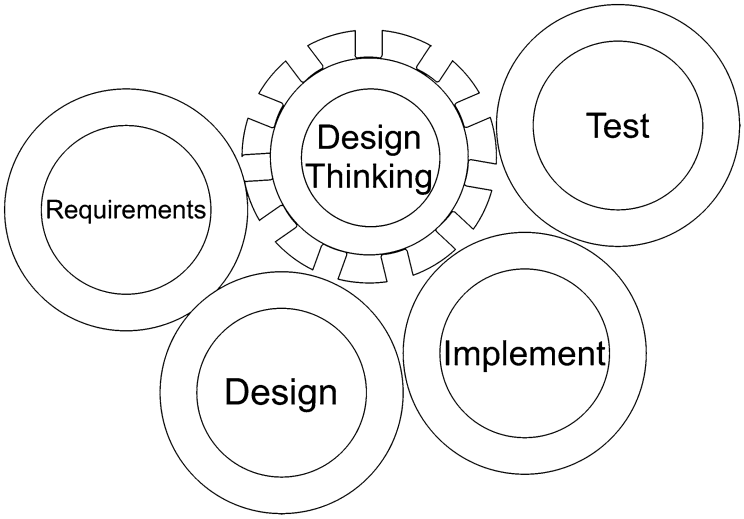


Fig. 5 Design thinking is central to the design candidate development processes

objectives to be avoided or postponed. Only in the limited premarket evaluation will users have an opportunity for intense independent, self-directed use of the product or service.

Assuming all goes well, the design thinking outcome is a highly promising prototype or proof of concept. The result will likely be limited in production level qualities, such as resiliency, reliability, and performance. Production shortcomings in these behind-the-scenes functional qualities will undermine the goodwill provided by a design's potential. Imagine if your ride-hailing app failed to coordinate a ride successfully 20% or more of the time. The novelty of a ride hailing app will compensate for premarket quality during final user evaluation. However, if the app is being marketed beyond a controlled evaluation, poor reliability will not be well received. How many times can users be disappointed in performance or reliability before the level of embracing quality slips towards rejected? Consider proposals to deploy design-thinking prototypes for general use carefully.

Having obtained experimental evidence of what feature functionality and feature sets are minimally sufficient, the user experience-oriented specifications of MMFs and MVP are likely to align with the market. Ideally the outcome of utilizing a MVP strategy is to earn revenue sooner, minimize costs, and maximize return on investment. However, there is likely to be significant tension as to what constitutes minimally necessary and sufficient for requirements that will address qualities like reliability, fitness for use, security, privacy, and performance. Design thinking is unlikely to produce much more than limited considered opinion on the significance and character of qualities users do not directly experience or recognize in context. Products and services that can be managed with continuous deployment may be

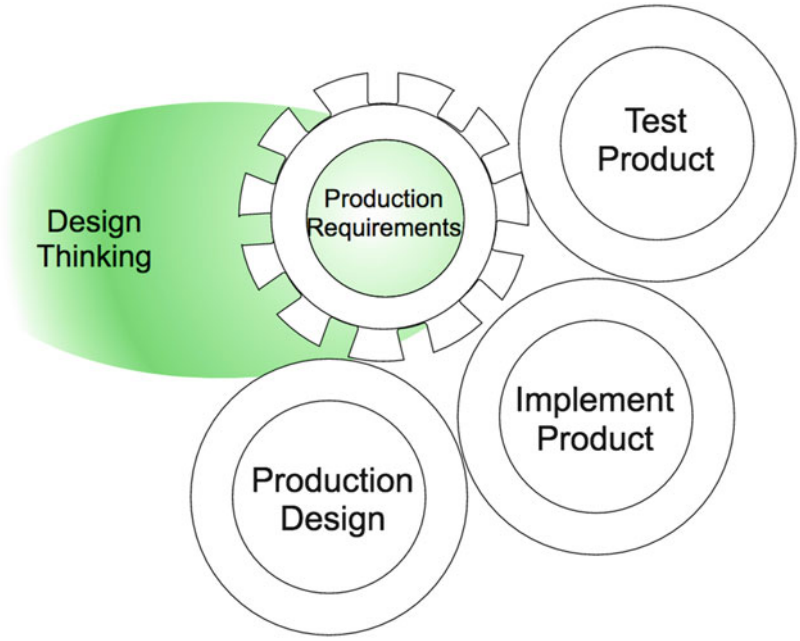


Fig. 6 Design thinking informs production requirements that govern production design, implementation, and testing

deployed relatively early with regular revision. Continuous deployment is less viable for products that at best can be patched or updated once a month.

Once the design thinking phase has completed, the interactive dynamic between users and the design for the most part ends. The design outcome provides an established set of user-facing requirements that need to be implemented and tested to accepted production quality. Given the speed at which prototype code is written, its base may only be partially salvageable. The languages and libraries chosen may have enabled prototyping speed and flexibility. Now knowing what needs to be implemented, the language and library selections may need to be revisited in order to provide a more manageable, reliable, secure, and computationally efficient foundation. By having senior developers participating on the design thinking project, various development conventions and tool chains used in standard development projects are more likely to be used during prototype implementation, thus making reuse more feasible. Figure 6 shows design thinking as a launch point that influences production requirements, but it is no longer a dynamic component among production development activities. The central motivating force is the production requirements activity, which will develop new requirements necessary for the result to be of sufficient reliability, fitness for use, security, privacy, usability, maintainability, and performance to be successful in the marketplace or in production.

5 Challenges of Design Thinking

Design thinking does not mitigate all the risk of innovation. At times it may appear to make innovation more risky. An innovation that involves people is best done with input from the affected users. Avoiding the user community results in a design that is based on assumptions that lack validation prior to release. These assumptions start with the problem definition and continue to the ultimate design outcome. Design thinking provides an approach to gathering, interpreting, and responding to user insights. This section explores some of the challenges design thinking either introduces or does not fully resolve for the design team.

Design thinking is not a cooking recipe (i.e., gather listed measured ingredients, prepare ingredients as specified, assemble, cook, and enjoy) for innovative success. The dimensions of problem domain, team composition, level of design thinking expertise, cooperating users, available materials and parts, supplier relationships, market timing, market conditions, budget, business climate outside of the team, and the state of the art individually and in combination may fatally hinder the design project or the ability to realize monetary value or brand elevation from its result.

Diversity of users within the targeted market is desired in order to recognize the breadth of preferences, task variety, and user challenges within the problem domain as well as challenges with the design alternatives. Some of the most informative users are those who "suffer" from "restless-user syndrome." This nonmedical condition presents itself as perpetual dissatisfaction with the products and services at hand combined with imaginings of capabilities, features, and qualities not yet considered or have been discarded. An "afflicted" person understands the task objectives and that these tasks are largely independent of technology, but technology strongly influences the manner in which a task is performed. Locating restless users for your cooperating user pool may not be easy; however, they are extremely helpful. Regular users are needed as well. They are able to share with the team reasonable and useful perspective on what is present in an alternative. However, a restless user will also guide the design team deeper into the problem domain or suggest things not previously considered. A challenge is to find a diverse set of restless users.

Basing design decisions on user input can be puzzling. Unlike electrical and mechanical measurements like speed, distance, height, or voltage, locating and interpreting relevant insights and relating them to each other is relatively more complicated. User-based signals are embedded in noise of uncertainty, unique personal preferences, distracted thought, limited imagination, inconsistent rapport, language shortcomings, and inter- and intra-user inconsistency. Team members familiar with interpreting user session results will be needed to identify commonality, useful suggestions, and guiding direction.

6 Conclusions

Embracing quality is a convergence quality that results from the integration of the user with a product or service. This quality emerges as a result of the user's intellectual and emotional response to accomplishing desired tasks using the product or service. Moreover, embracing quality is an aggregation of the responses of multiple users over multiple uses. This quality indirectly influences revenue, brand, efficiency, and productivity.

Embracing quality is difficult to design for directly. This challenge results from it being an emerging aggregation of service or product qualities (e.g., performance, security) as well as being highly dependent on users. The greatest influence the design team may have on the user is through user experience design and instructional resources.

Often the design team must anticipate what the user truly wants, needs, and desires. Understanding these things requires that designers engage users during design. By utilizing design thinking, a design team is able to augment their efforts by allowing user insights to influence design by incorporating their preferences, criticism, and suggestions.

In order to achieve embracing quality, seek out users and allow them to: inform the problem definition, create an affinity with the designers, and help you help them. In other words, center your design on the user by utilizing design thinking or other user-centered approaches in order to improve the likelihood of market success.

References

1. Mazzur, G.: History of QFD – Introduction. http://qfdeurope.com/en/history-of-qfd/ (2015). Accessed 14 June 2019
2. Csikszentmihalyi, M.: Flow: The Psychology of Optimal Experience. HarperCollins, New York (2008)
3. ISO/IEC: ISO/IEC 25010:2011 - Systems and software engineering – Systems and Software Quality Requirements and Evaluation (SQuaRE) – system and software quality models. In: International Organization for Standardization (2011)
4. Liedtka, J., Ogilvie, T.: Designing for Growth: A Design Thinking Tool Kit for Managers. Columbia University Press, New York (2011)
5. Lewrick, M., Link, P., Leifer, L.: The Design Thinking Playbook: Mindful Digital Transformation of Teams, Products, Services, Businesses and Ecosystems. Wiley, Hoboken (2018)
6. Cross, N.: Design Thinking: Understanding How Designers Think and Work. Berg, Oxford (2011)
7. A4Q Design Thinking Foundation Level. https://isqi.org/us/en/a4q-design-thinking-foundation-level (2018). Accessed June 14 2019
8. Manzo, P.: Fail faster, succeed sooner. https://ssir.org/articles/entry/fail_faster_succeed_sooner (2008). Accessed June 14 2019
9. Design Council. Eleven lessons: Managing design in eleven global companies - desk research report. Design Council (2007)
10. The Design Process: What is the double diamond?. https://www.designcouncil.org.uk/news-opinion/design-process-what-double-diamond. Accessed June 14 2019

11. Doorley, S., Holcomb, S., Kliebahn, P., Segovia, K., Utley, J.: Design Thinking Bootleg. Hasso Plattner Institute of Design at Stanford. Stanford (2018)
12. Liedtka, J., Oglivie, T., Brozenske, R.: The Designing for Growth Field Book: A Step-by-Step Project Guide. Columbia University Press, New York (2014)

Developing Software Quality and Testing Capabilities in Hispanic America: Challenges and Prospects

Ignacio Trejos-Zelaya

Abstract This chapter presents a summary of the Hispanic America region status and prospects in software engineering—particularly those regarding software quality and software testing.

Keywords Software testing · Software quality · Software tester · Software testing skills · Software engineering

1 Introduction

By the turn of the century, humankind had already entered the information age. Modern societies have grown increasingly dependent on information and technology for their proper functioning, growth and survival. For over two decades, developed countries have found that shortages of capable Information Technology (IT) workers may hinder their prosperity [1].

In the last years of the twentieth century, the IT sector was experiencing an abrupt growth [2, 3]. The *Computer World* weekly reported an exceptional growth of IT employment positions: from 160,000 jobs in 1997 to 800,000 in year 2000 [4].

In emerging economies such as those in Spanish-speaking America, *software* development has opened opportunities for establishing export and outsourcing services industries. For example, during the same period, Costa Rica had software companies and start-ups exporting successfully to Latin America and North America, growing at a rate of 40–60% in headcount [5, 6].

A successful software ecosystem requires that all aspects of software development, integration, reuse, and maintenance be considered and performed with excellence. Most computing higher-education programmes in Latin America lean either towards *Computer Science* or *Information Systems* [7]; although most include several programming courses in their curricula, few of them embrace *Software*

I. Trejos-Zelaya
Universidad Cenfotec & Tecnológico de Costa Rica, San José, Costa Rica

S. Goericke (ed.), *The Future of Software Quality Assurance*,
https://doi.org/10.1007/978-3-030-29509-7_14

Engineering as their main subject, and still less do focus on software quality and/or software testing.

We shall present a view on the challenges ahead for the development of software quality and testing capabilities in Spanish-speaking America and prospects for its expansion and progress.

After this Introduction, the chapter is organised as follows: the next section provides a background on the evolution of computing in Latin America since the 1950s to the present day, then goes into describing workforce demand and supply and skill sets required for IT and particularly in software engineering, emphasizing those related to individual and social competencies. Examples of the Hispanic America software industry are provided, followed by a discussion of the impact of professional certification schemes on software quality and testing on the region. Then, the situation of Hispanic America post-secondary education on software engineering is presented, with specific analysis of the author's environment—Costa Rica. The chapter concludes with a view on the challenges ahead and promising prospects lying in the future of software quality and software testing in Hispanic America.

2 Background

Computing in Latin America started in the late 1950s when the first computers were introduced in countries such as Mexico, Brazil, Colombia and Chile [8]. Da Costa [9] asserts that, since those times, computing in Latin America has been influenced mostly by the USA and, to a lesser extent, Western Europe. Economic, political and social matters were diverse during the 1960s and 1970s, and had an influence on the development of computing-related higher-education programmes and local IT ecosystems; there are significant variations in the evolution of IT among countries. Larger countries, such as Mexico, Brazil and Argentina, started assimilating computing technologies earlier than smaller ones. First users were universities, governments and large corporations (some state-owned).

First degree programmes appeared close to engineering, mathematics or science faculties. Among the first were Argentina's Universidad de Buenos Aires, which established in 1962 a scientific computing programme as a sequel to their *Instituto de Cálculo* (Calculation Institute). National Polytechnic Institute founded their degree programme in computing engineering in 1965, followed by the National Autonomous University of Mexico (UNAM). Venezuela, Chile, Colombia and Peru followed suit in the late 1960s. About the same time, in Costa Rica, industrial IT players (IBM and Burroughs) offered training courses in programming and system administration to engineering and science students and professionals to grow local capabilities for operating and developing computing systems. During the 1970s, most Latin American countries established educational programmes related to computing. Terminology was diverse; frequently appearing terms were 'Informática' (*Informatics*), 'Computación' (*Computing*) or 'Ingeniería de Sistemas' (*Systems Engineering*). Colombia's Universidad Nacional established the

first master's program in Computer Science in Latin America. Brazil was first in creating doctoral programmes and, together with Chile, has grown scientific capabilities more widely than all other Latin American countries.

Latin America has experienced the same IT transitions that have occurred in more developed countries, typically with a lag of 1 to 5 years as compared to the originating technology's country. Overall, Latin America has been mostly a follower in computing hardware innovation. Brazil set trade barriers to protect its industry, then funded doctoral scholarships for studies abroad with a view to build its scientific and technological research and development capabilities. Upon returning, scholars help build national research laboratories or centres at universities. Brazil currently incorporates digital technologies to create an innovative environment to foster economic development.

Chile stimulates Digital Government and nurtures public–private collaboration for innovation in education and business. Smaller countries, such as Costa Rica and Uruguay, obtained loans from the Inter-American Development Bank aimed at supporting their software industries' export capabilities through improving quality and productivity in companies, updating university curricula, building institutional competencies and promoting innovation and entrepreneurship. Colombia and Mexico sponsor technology parks and regional initiatives for industrial clusters that include digital technologies and services companies.

Nowadays, due to the proliferation of the Internet (and the World Wide Web) and mobile communications, digital technologies have become pervasive in Latin America. Broadband access, together with smart phones, laptops and digital TV sets, has opened a market of enormous proportions, in a very short time frame: "442 million unique mobile subscribers across Latin America and the Caribbean, accounting for 68% of the [sub-continent] population" [10]. Digital Transformation is on the rise in the Americas, demanding more and more software applications to enable and sustain transformed business processes or new endeavours.

3 Skill Sets

In the report 'The Future of Jobs', The World Economic Forum [11] states:

> Disruptive changes to business models will have a profound impact on the employment landscape over the coming years. Many of the major drivers of transformation currently affecting global industries are expected to have a significant impact on jobs, ranging from significant job creation to job displacement, and from heightened labour productivity to widening skills gaps. In many industries and countries, the most in-demand occupations or specialties did not exist 10 or even five years ago, and the pace of change is set to accelerate.

Digital technologies have been instrumental to changes impacting nearly every human activity worldwide. Those technologies, themselves, have made great strides in the last 60 years, influencing and accelerating scientific, technological and business development overall.

The 'IT skills' gap refers to the shortage of qualified candidates to fulfil open positions in occupations related to digital technologies. The problem has been reported several times [12, 13, 14], and it appears in many ways [8, 15]:

- A deficit of educated candidates to fulfil job openings
- Candidates who are inadequately prepared for certain IT positions
- New occupations generated by technological change and innovation
- Novel business models, products, and services
- Need to retrain personnel for assimilating acquired technologies
- Requirement to operate and maintain information systems built on technologies no longer taught at universities
- Impaired diversity due to underrepresentation of women and ethnic minorities in the IT workforce, which negatively impacts innovation, productivity, creativity and prosperity
- Institutions slow in designing and updating curricula to educate future technologists and developing programmes to retrain those already employed

Most Latin American countries report that demand of IT skills exceeds supply. However, there is more to those situations:

- Diversity of skills required in the labour market is scarce or even unknown (e.g. storage, digital forensics, data science, cloud computing, blockchain, artificial intelligence, Internet of Things).
- Quality of the education of IT graduates is uneven: range of knowledge, depth of know-how, significance of experience, certification of skills, mastery of domain, in addition to an understanding of their discipline's foundations.

3.1 Software Engineering

As a computing sub-discipline, *Software Engineering*'s origins can be traced back to the 1960s, particularly the first Software Engineering Conference sponsored by the NATO in October 1968 [16]. Here we quote some relevant parts of the report:

> The discussions cover all aspects of software including
>
> – Relation of software to the hardware of computers
> – Design of software
> – Production, or implementation of software
> – Distribution of software
> – Service on software
>
> Some other discussions focussed on subjects like:
>
> – The problems of achieving sufficient reliability in the data systems which are becoming increasingly integrated into the central activities of modern society
> – The difficulties of meeting schedules and specifications on large software projects
> – The education of software (or data systems) engineers
> – The highly controversial question of whether software should be priced separately from hardware

In 1975, the IEEE Computer Society started publishing the *IEEE Transactions on Software Engineering* as a bimonthly peer-reviewed scientific journal and also sponsored the first edition of what became the International Conference on Software Engineering; both continue to this day. It was followed by the *IEEE Software* magazine in 1984. The British Computer Society sponsored publication of the *Software Engineering Journal* (1986–1996, now available via IEEE Xplore), and several more conferences and refereed journals related to facets of Software Engineering have appeared since the 1980s.

By year 2000, Finkelstein and Kramer defined *Software Engineering* as "the branch of systems engineering concerned with the development of large and complex software intensive systems. It focuses on: the real-world goals for, services provided by, and constraints on such systems; the precise specification of system structure and behaviour, and the implementation of these specifications; the activities required in order to develop an assurance that the specifications and real-world goals have been met; the evolution of such systems over time and across system families. It is also concerned with the processes, methods and tools for the development of software intensive systems in an economic and timely manner" [17].

Recognising the need to establish software engineering as a profession, the Association for Computing Machinery (ACM) and the IEEE, established the Software Engineering Coordinating Committee in 1993 to jointly work on:

- Defining a Body of Knowledge on software engineering (SWEBOK, now in its third version [18])
- Agreeing on a Software Engineering Code of Ethics and Professional Practice [19]
- Developing curriculum recommendations on software engineering for undergraduate [20] and graduate education [21]
- Describing a set of competencies, at five levels of competency [22] (SWECOM)

SWEBOK's objectives are:

1. To promote a consistent view of software engineering worldwide
2. To specify the scope of, and clarify the place of software engineering with respect to other disciplines such as computer science, project management, computer engineering, and mathematics
3. To characterize the contents of the software engineering discipline
4. To provide a topical access to the Software Engineering Body of Knowledge
5. To provide a foundation for curriculum development and for individual certification and licensing material

Some organisations have sponsored the development of professional certifications on software quality and software testing:

- *American Society Software Quality Engineering* (ASQ): Certified Software Quality Engineer [23] since 1996
- *International Software Testing Qualifications Board* (ISTQB): Founded in 2002 by 8 country members (Austria, Denmark, Finland, Germany, the Netherlands,

Sweden, Switzerland, and the UK), it now includes 59 member boards. Based on pioneering work by the British Computer Society's Information Systems Examinations Board (ISEB) and the German Testing Board, ISTQB has been developing a comprehensive set of qualifications assessments related to software testing—both technical and managerial—in various levels [24]:

  - Foundation level
  - Foundation level extension (agile, model-based tester, automotive software, mobile application, acceptance, performance, usability)
  - Advanced level (test manager, test analyst, technical test analyst, agile technical tester, security tester, test automation engineer)
  - Expert level (improving the testing process, test management)

- *Institute of Electrical and Electronics Engineers* (IEEE), *Computer Society* (IEEE/CS): associate software developer, professional software developer, and professional software engineering master. IEEE/CS offer review courses based on the SWEBOK that help candidates prepare for certification assessments. Two such courses cover software quality and software testing.
- *International Software Quality Institute* (iSQI): develops its own professional certifications and exams supporting other organisations' certification programmes. Areas covered include: software testing, management, software architecture, requirements engineering, software development, usability, and agile methods.

Standards for software development started to appear in the 1960s. The IEEE has been developing the most comprehensive family of civilian Software Engineering Standards since the 1980s. Also relevant are standards promoted by the ISO and the IEC. Some of those standards are particularly relevant to software quality and software testing:

- Software Quality Assurance Processes (IEEE Std 7304)
- System, Software, and Hardware Verification and Validation (IEEE Std 1012)
- Software Reviews and Audits (IEEE Std 1028)
- Software Testing (ISO/IEC/IEEE 29119)
- Configuration Management in Systems and Software Engineering (IEEE Std 828)
- Classification for Software Anomalies (IEEE Std 1044)
- System and software quality models (ISO/IEC 25010, preceded by ISO/IEC 9126)
- Software Requirements Specifications (IEEE Std 830)
- Software Life Cycle Processes (ISO/IEC/IEEE Std 12207)
- System Life Cycle Processes (ISO/IEC/IEEE Std 15288)

3.2 Human Competencies

At the turn of the present century, industry and professional associations, as well as academic forums recognised the need to better prepare future professionals for life in the workplace. Some engineering educators called for a more rounded and integrative approach, as opposed to the prevailing analytical and reductionist model [25]. The USA's National Academy of Engineering stated these needed attributes of engineers by year 2020 [26]: strong analytical skills; practical ingenuity; creativity; communication; business and management; leadership; high ethical standards; professionalism; dynamism, agility, resilience, and flexibility; lifelong learning.

Those attributes agree with the findings reported in what employers express in surveys presented in [7] and [27], where the *human*, non-technical, attributes mentioned by 50% or more of the respondents include: leadership; ability to work in a team; communication skills (written and verbal); problem-solving skills; work ethic; initiative; analytical/critical/quantitative skills; flexibility/adaptability; interpersonal skills; organizational ability; strategic planning skills.

There is a trend in engineering education and accreditation schemes to include 'soft skills' in addition to more 'engineering skills' as part of graduate attributes from engineering degrees. The foremost example is the International Engineering Alliance (IEA) Graduate Attribute Profile [28], "Graduate attributes form a set of individually assessable outcomes that are the components indicative of the graduate's potential to acquire competence to practise at the appropriate level"; these comprise: (1) engineering knowledge, (2) problem analysis, (3) design/development of solutions, (4) investigation, (5) modern tool usage, (6) the engineer and society, (7) environment and sustainability, (8) ethics, (9) individual and team work, (10) communication, (11) project management and finance, (12) lifelong learning.

Professional work in software quality assurance or software testing improves when individuals have developed their communication and teamwork capabilities. For example, we highlight the matter with quotations taken from [29]:

- Errors may occur for many reasons, such as [. . .] Miscommunication between project participants, including miscommunication about requirements and design
- A tester's mindset should include curiosity, professional pessimism, a critical eye, attention to detail, and a motivation for good and positive communications and relationships
- In some cases organizational and cultural issues may inhibit communication between team members, which can impede iterative development
- Additional benefits of static testing may include: [. . .] Improving communication between team members in the course of participating in reviews
- Potential drawbacks of test independence include:
  - Isolation from the development team, leading to a lack of collaboration, delays in providing feedback to the development team, or an adversarial relationship with the development team
  - Developers may lose a sense of responsibility for quality
  - Independent testers may be seen as a bottleneck or blamed for delays in release [. . .]

With regard to personal competencies, two project management professional associations, the International Project Management Association (IPMA) and the Project Management Institute (PMI), have developed competency models that include 'behavioural', 'people' or 'personal' competencies: IPMA Individual Competence Baseline and PMI Project Manager Competency Development Framework, respectively. Key traits identified: leadership (leading), relations & engagement, teamwork, managing, motivation, assertiveness, self-reflection & self-management, personal communication (communicating), relaxation, openness, creativity, cognitive ability, resourcefulness, results orientation, efficiency, effectiveness, consultation, negotiation, conflict & crisis, values appreciation, personal integrity & reliability, ethics, professionalism.

4 Industry

For nearly 30 years, Computer Science programmes have sported 1 to 4 courses (typically 2) related to software development, sometimes including one or two 'capstone' project(s). A major problem faced by the industry worldwide is that university-level degree programmes on software engineering, as something distinct from Computer Science or Information Systems, have appeared only recently (Rochester Institute of Technology's was the first such in the USA, in 1996).

Industry has had to cope with the situation and invest in training their personnel beyond the programming skill-set that recent graduates bring when recruited. Fresh graduates lack the subject knowledge and the discipline required for professional software engineering, but can learn quickly.

Latin America has not been a major worldwide player in the product-oriented software industry. Each country has developed its own software industry, serving local needs, and some companies have been successful in growing software services for export. Aiming at strengthening their export capabilities, smaller countries, such as Costa Rica and Uruguay, reached the Inter-American Development for funding improvements of their software companies' productivity and quality, upgrading and updating university curricula, building institutional competencies, fostering innovation and entrepreneurial initiatives, in addition to expanding industry-academia collaboration.

Software services comprise development, maintenance, migration, support and testing. Mexico's *Softtek* (www.softtek.com) is the largest Latin American IT company. Softtek offers diverse technology and business transformation services, and is credited to inventing the *nearshore* concept—serving mainly North America in compatible time zones. Softtek adheres to the Capability Maturity Model (CMMi) and has achieved its level five. It also uses Six Sigma as a customer-focused and data-driven management method for problem resolution, business process excellence and process improvement. Within an extensive service portfolio, Softtek offers QA and software testing services using their own software QA & validation methodology and quality assurance and validation maturity model. In QA and

testing, their services comprise: quality assurance, software testing, test automation, performance testing, mobile software testing, security and penetration testing and agile software testing. The company is led by Blanca Treviño since year 2000 and has subsequently experienced steady growth. Ms. Treviño has also been a role model for women in technology, helping initiatives to attract female talent to IT. In order to overcome limitations of new hires in software testing and quality assurance, Softtek has a wide-ranging training and career development programme, which also includes bridges to industry-recognised professional and technical certifications.

Notable amongst Latin American companies is *Choucair Testing* http://www.choucairtesting.com Founded in 1999 by María Clara Choucair, the company focusses on software testing and quality assurance services. Their approach, known as *Business Centric Testing* (BCT), concentrates on designing an appropriate service adapted to each client company's business model, competitive strategy and user experience. Choucair Testing has become the largest Hispanic America provider of software quality assurance and testing services. Building upon its experience in the banking and finance industry, Choucair now covers more application and business domains with diverse technologies. Their testing services comprise: functional testing, performance testing, mobile testing, web testing, business intelligence testing, usability testing, accounting & finance testing, migration testing, comprehensive system testing, comprehensive acceptance testing, automation testing, SAP automation testing, payroll testing, security testing, transactional switch testing, internationalisation and localisation testing, testing environment preparation and continual improvement, technology and knowledge transfer.

As the need for controlling, testing and improving software quality grows in Hispanic America, the region has witnessed the appearance of more companies specialising in those services. Countries such as Mexico, Costa Rica and Uruguay have been successful in attracting foreign direct investment from technology companies who either establish their own units or centres of excellence on software testing, or seek to outsource or out-task to specialist companies. Thus, the demand for professionals knowledgeable on software quality and testing has grown steadily during the last decade.

5 Impact of Professional Certification Schemes

Motivated by the need to grow and improve her company's chosen domain of expertise, María Clara Choucair sought out training frameworks and certification schemes in order to develop Choucair Testing's specialist workforce. She found the ISTQB certification schemes and, with help from the Spanish Software Testing Qualifications Board (SSTQB) and the International Software Quality Institute (iSQI), led the establishment of the *Hispanic America Software Testing Qualifications Board*, as a regional member of the ISTQB.

In Costa Rica, companies such as Electronic Data Systems (EDS) and Hewlett-Packard prompted their engineers to seek certifications by the BCS's Information Systems Examination Board (ISEB), prior to becoming a member country of the HASTQB Regional Board within the ISTQB. Before that, from 2003 to 2010, courses for preparing for ASQ's *Certified Software Quality Engineer* were run by Cenfotec (www.ucenfotec.ac.cr); some 120 professionals hold the CSQE in Costa Rica, most of them working in the biomedical device sector. ASQ's impact has been wider and long-standing in the Manufacturing and Life Sciences sectors in Costa Rica (certifications on *Quality Auditor*, *Quality* Engineer, *Six Sigma Black Belt*, *Six Sigma Green Belt*, *Six Sigma Yellow Belt*).

ISTQB professional certifications in software testing have been growing steadily in Hispanic America in recent years. At the close of 2018, 7499 professionals had earned one or more ISTQB certifications. This can be observed in Fig. 1.

Hispanic America follows the worldwide trend of Foundation being the most sought after certification level, as can be seen in Fig. 2.

Figure 3 shows the distribution of certified professionals per country in the Hispanic America region.

It is clear that Colombia leads the pack in Hispanic America. Costa Rica's progress since 2013 has been steady, due to the growth in demand for certifications and market recognition of software testing and software quality assurance as professional fields of endeavour. This can be seen in Fig. 4.

Costa Rica's growth can be traced to a very productive collaboration between academia and industry. Starting around 1999, some universities have offered scant and scattered training on software quality assurance and software testing, but more frequently since year 2014—with Universidad Cenfotec taking the lead since 2004.

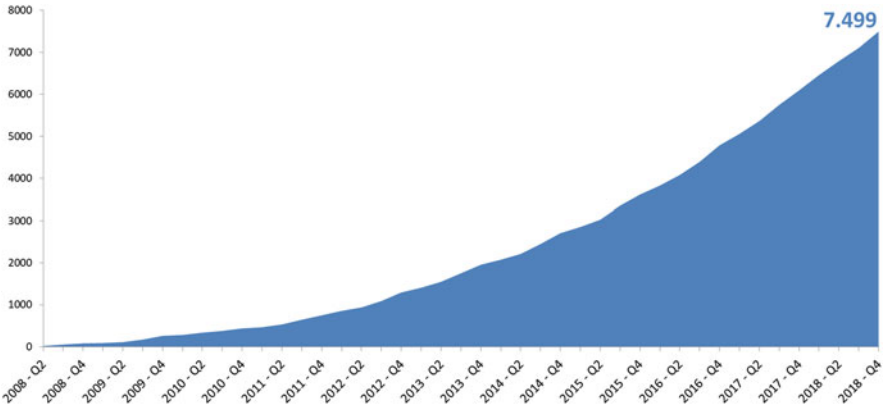


Fig. 1 Evolution of ISTQB Software Testing certifications in Hispanic America 2008–2018

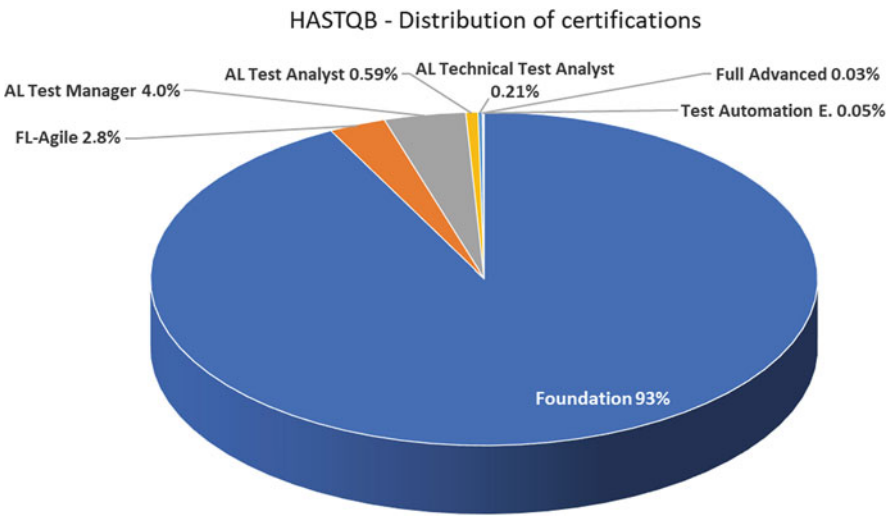


Fig. 2 Hispanic America distribution of software testing certifications, per ISTQB levels

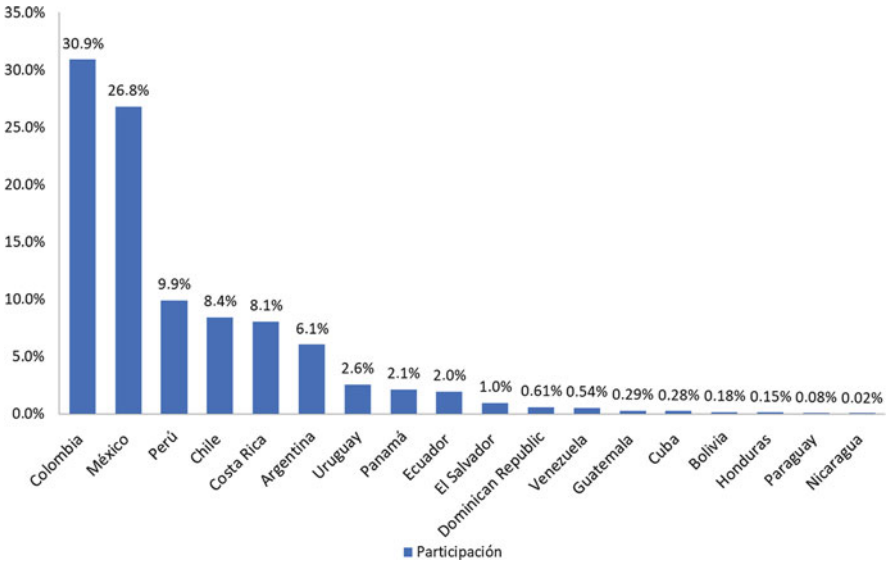


Fig. 3 Distribution of ISTQB's certification per Hispanic American country

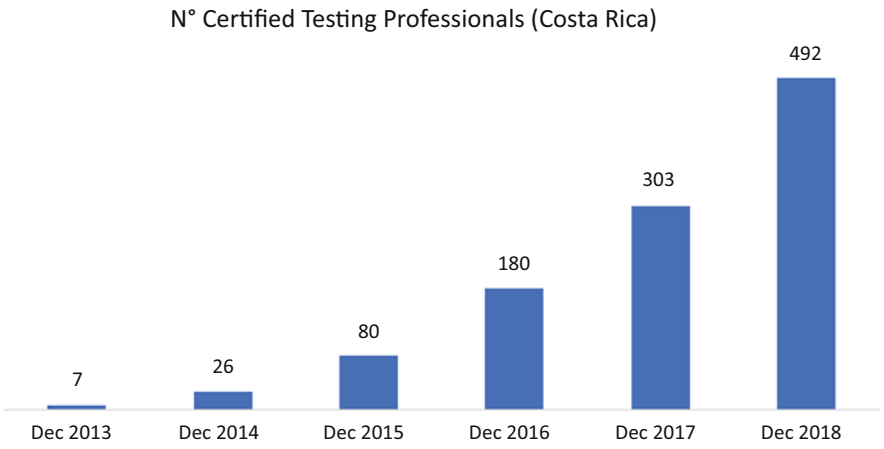


Fig. 4 Growth of certified testing professionals in Costa Rica, 2013–2018

6 Higher Education on Software Quality and Testing

In Hispanic America there are very few degree programmes on Software Engineering, as such. Costa Rica and Panama were among the first countries to develop full degree programmes focussing on Software Engineering. Other countries have 'Systems Engineering' programmes that blend computer science, information systems and software engineering knowledge areas. The result is that few degree programmes on computing provide sufficient space for software quality assurance processes, software verification and validation, software or software testing.

It not just a matter of *knowledge*, for *experience* is lacking as well as *attitude*. This explains the difficulties faced by employers in hiring recent graduates with basic competencies on software quality and testing. Large companies such as Softtek and Choucair Testing have developed training programmes that start with foundational courses and workshops functional software testing, and then progress towards testing analysis, technical testing, security testing, performance testing, usability testing, Web testing, mobile testing, testing management, agile testing practices, testing process improvement and software quality assurance—among others. Smaller companies have a reduced assortment of training courses or workshops, and resort to guided self-study or mentored study groups. Syllabi from the ISTQB and ASQ (CSQE) certifications have provided guidance for organising and aligning course content or studies.

The above poses challenges for designing innovative curricula at universities and colleges. The following section offers examples from university experiences in Costa Rica.

6.1 Curriculum Design and Development Experiences in Costa Rica

Professors from the Costa Rica Institute of Technology (TEC, Tecnológico de Costa Rica) and the University of Costa Rica (UCR, Universidad de Costa Rica) developed extension training courses (continuing education) and consultancies on software quality assurance, software processes and methodology and software testing around 1995. Course contents and consultancies were mostly based on IEEE's Software Engineering standards. By 1997, Prof. Marcelo Jenkins developed software metrics and software processes courses MSc in computing. Open and in-house training courses on software quality assurance were offered by TEC in 1999 and 2000.

6.1.1 Cenfotec

In year 2000, Cenfotec was founded as a technical college by a group of software development entrepreneurs and academics interested in contributing to Costa Rica's knowledge-based economic development. Cenfotec started offering a 2-year programme on software development, grounded on Software Engineering. The first three terms of the original programme included a software engineering project, three technical courses and one English for IT course (nine computing science and software engineering courses, three English courses, three software engineering projects). A fourth term comprised a practicum (internship) and two technical elective courses. The three integrative software engineering projects were the curriculum backbone; each exercised requirements elicitation, analysis and specification, software design, construction and testing, with different technological mixes and processes. The projects were incremental, progressively integrating knowledge and skills required for problem-solving using systematic engineering processes, employing an approach that helped grow intertwined 'hard' and 'soft' skills required for teamwork and future professional endeavours. Experience-driven *learn-by-doing collaboratively* computing products or services was implemented within projects by teams of students that performed roles inspired by Watts Humphrey's *Team Software Process* [30] and the *Rational Unified Process* [31]. Actual teamwork was achieved: 4 to 6 team members collaborate, performing roles for which they are accountable, employing emotional and social intelligences as cornerstones [32], and developing good work habits [33]. Benchmarking against the SWEBOK v0.95, revealed the need to improve software testing education.

In July 2003, Cenfotec agreed with Universidad Latina (Costa Rica's largest private university) to develop there Costa Rica's first Software Engineering degree programme, as a continuation of Cenfotec's 2-year programme. The knowledge content was validated against a draft of the *Software Engineering Education Knowledge* (SEEK), that was published by a year later by the Association for Computing Machinery and the Institute of Electric and Electronic Engineers's Computer Society [34]. That Software Engineering degree had improved coverage

of software process, software quality assurance, software testing and software configuration management, in addition to reinforced strengths in requirements, design, construction and the project approach. Figure 5 shows a view of the knowledge mapping performed against the SEEK.

As stated above, most computing degrees typically lack software testing and quality assurance content. This prompted curriculum design work towards specialisations on software quality and testing. Between years 2002 and 2005, Cenfotec developed continuing education courses on software quality assurance and software testing, and also offered preparation towards ASQ's *Software Quality Engineer* certification (years 2003–2010). With the help of Costa Rica's Investment Promotion Agency (CINDE, www.cinde.org/en) and collaboration from 13 companies[1], Cenfotec designed and validated a *Software Testing and Quality Technician* programme in 2009, targeted at people with some education in programming; the programme failed to attract sufficient students. Curricula for postgraduate programmes on software quality engineering and testing were drafted between 2004 and 2008. From 2009 to 2011, Cenfotec pursued authorisation to obtain formal university status.

In order to improve continuing education and design postgraduate specialisations, Cenfotec contacted the Hispanic America Software Testing Qualifications Board (HASTQB) in 2009, and Costa Rica gained membership in 2010. For curriculum design, knowledge input was sought at ISTQB's syllabi (Foundation and Advanced levels) and CSQE's Body of Knowledge. Another valuable source was O$_*$NET [35], a database with standardized descriptors on about 1000 occupations in the USA, provided knowledge, skills, tasks and work characterisations related to software testing and quality engineering. Curricula for software quality engineering and software testing were designed and validated during 2010 and 2011, including coverage analysis of syllabi from ISTQB's advanced-level certification schemes (test analyst, technical test analyst, test manager). A six-course specialisation was launched in late 2011. Figure 6 illustrates some of the task abilities identified and validated for graduates of the specialisation ('E') or a future Master's ('M') programme.

Cenfotec was granted university status in December 2011, and started operating as Universidad Cenfotec in January 2012. By now, it has several postgraduate studies programmes, which include an MSc in database technology (comprising data analytics and data management) and an MSc in cybersecurity. Although a full-fledged MSc in Software Testing & Quality Engineering was designed, market research revealed that it was not advisable to launch it yet. A forthcoming MSc in Software Engineering will have software quality engineering and advanced software testing among its concentrations. The current degree programme in software engineering includes four software engineering projects where students apply the basics of software processes, software configuration management, software testing and

[1] AdvanceMe, Avántica Technologies, Babel Software, Experian, Hewlett-Packard, Intertech International, L.L.Bean, Qualitas Factory, RoundBox Global, Schematic (now Possible Worldwide), TestingSoft, Via IT, Wal-Mart.

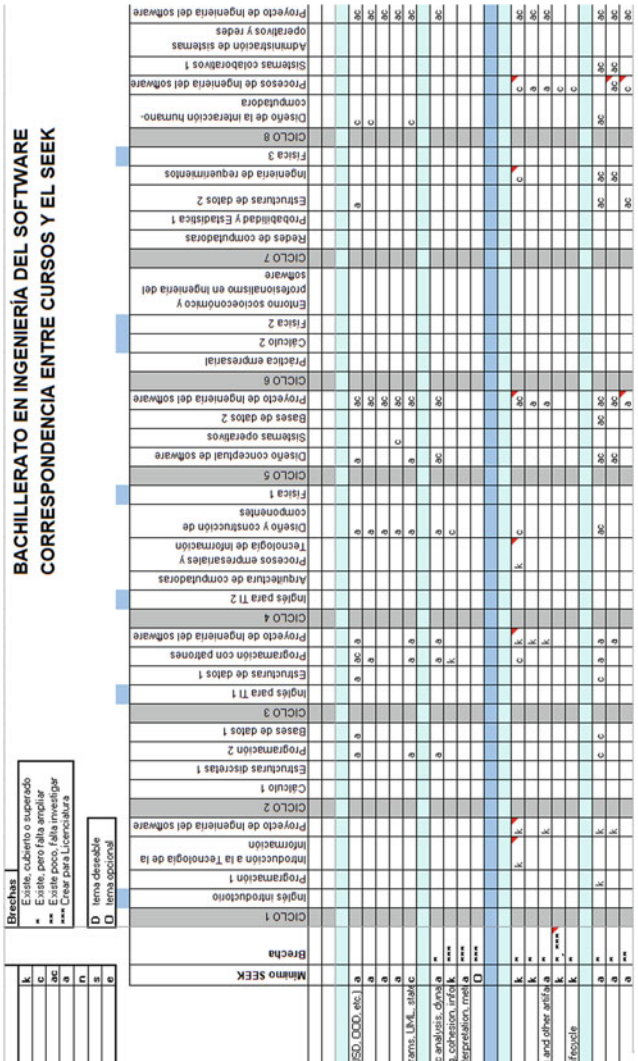


Fig. 5 Mapping Cenfotec's Software Engineering to the SEEK (2003)

E	Design test plans, scenarios, scripts, or procedures.
E	Test system modifications to prepare for implementation.
E	Develop testing programs that address areas such as database
E	Document software defects, using a bug tracking system, and report
E	Identify, analyze, and document problems with program function,
E	Monitor bug resolution efforts and track successes.
E	Create or maintain databases of known test defects.
M	Plan test schedules or strategies in accordance with project scope or
E	Participate in product design reviews to provide input on functional
E	Review software documentation to ensure technical accuracy,
E	Document test procedures to ensure replicability and compliance
E	Develop or specify standards, methods, or procedures to determine
E	Update automated test scripts to ensure currency.

Fig. 6 Fragment of software quality engineer graduate profile (task abilities). Cenfotec (2009)

quality assurance; in addition to the projects and several courses on programming, requirements, design and construction, there are courses going deeper on software engineering processes and software verification and validation.

As of this writing, Cenfotec is launching two new programmes on software testing: *Software Testing Technician* (four terms, nine courses), *Software Test Analyst* postgraduate programme (five courses, five certifications aligned with ISTQB, IREB and Selenium).

6.1.2 State Universities

Since 2008, the Distance Education University (UNED) offers a Postgraduate Diploma ('Licenciatura') on *Informatics Engineering and Software Quality* ('Ingeniería Informática y Calidad del Software') which includes nine courses and a final project (either an internship or a directed research). The courses include: IT systems quality management, advanced requirements engineering, quality systems, change management, configuration management, IT marketing, analytical methods and software quality metrics, software quality control and software quality certification models.

The Costa Rica Technical University started offering a degree programme on software engineering in 2011, which includes one course on software quality and another on software quality management and validation. The Costa Rica Institute of Technology started in 2012 a new version of its reputed *Computing Engineering* degree ('Ingeniería en Computación'), which included, for the first time since 1976, a course covering the foundations of software testing, software quality assurance and software configuration management; input from employers and graduates were the main reason for including such a course in the curriculum. In 2017, the University of Costa Rica made a major change in its prestigious *Computer Science*

and Informatics 4-year degree programme. After completing 2 years of studies, students of computing can choose to pursue two further years in one of these concentrations: *computer science*, *software engineering* or *information technology*. The software engineering concentration offers a course on software quality and another on software testing. The remaining state-sponsored university, National University (Universidad Nacional), does not include software quality or software testing as separate subjects within its degree or postgraduate programmes.

6.1.3 Other Private Universities

Software quality and software are not common in other computing degree programmes in Costa Rica. Universidad Invenio includes a course called *Quality and Productivity in Enterprise Information and Telecommunication Technologies* in its Enterprise ICT degree programme. Universidad Fidélitas launched a Postgraduate ('Licenciatura') Diploma on Software Quality Management that includes eight courses, a Research Seminar and a Graduation (thesis) Project; the courses include: quality management systems, knowledge management, software engineering, project formulation and evaluation, ICT quality models and standards, information system measurement, ICT quality auditing, management of ICT resources. The U. Fidélitas programme is not particularly geared towards software quality and less so towards software testing.

6.2 Challenges and Prospects

The author, being a university professor, inevitably biases his views from an educational and research perspective.

6.2.1 Challenges

In the author's view, these are some of the major challenges faced by the Hispanic America region with regards to software quality assurance and software testing:

- Faculty and university acknowledgement of software engineering as an important computing discipline, distinct from computer science and information systems [36] and recognising software quality, software testing and software maintenance as subjects that need study and space in curricula. Faculty members interested in software engineering are few compared to those specialised in more traditional areas of computer science or information systems. Those interested in software engineering do not lean particularly towards software quality or Software testing, usually preferring software technology, software design and software construction.

- Educating future software professionals requires more than technical knowledge, it requires practice and experience. There are excellent resources available for teaching and organising courses related to programming, data structures, algorithms or databases—to name subjects which are quite mature and longstanding in computing curricula. In software testing, professors face the challenge of have concrete artefacts (code, models, plans, asset versions) for their students to test and analyse. Tool selection, for practical workshops is very challenging—as per its availability or suitability for teaching and learning—and also in terms of affordability by educational institutions. Software processes related to software testing, quality management, version control and configuration management need to be scaled down to educational settings, yet not be made too simple to become irrelevant or trivial. Measurement of quality and productivity needs product assets, project data and process history metrics. The challenge of curriculum-in-the-large can be faced successfully (as illustrated above), there is a major challenge of curriculum-in-the-small design and production (textbooks, laboratories, assets to be assessed or measured, presentations, software standards, workflow definitions, etc.).
- Industry recognition of software quality as subject of foremost importance. IT and software managers graduated when software quality and software testing were not part of their computing education (even less so if transferring from management or classic engineering degrees into computing). They do not know how to manage testing or quality assurance, they do not usually include the subjects in budgets, nor assign time and resources to activities related to them. Software testing and software quality assurance are absent, generally.
- Organisations more mature and knowledgeable about the costs of poor software quality will press suppliers, outsourcing companies and universities. Governments, banks and large corporations immersed in digitisation of their processes and businesses need to ensure the reliability of all software-intensive systems that support them.
- As with the rest of the world, Hispanic America will need to develop capabilities to address the challenges raised by mobile computing, usability, accessibility, Internet of Things, cyberphysical systems, safety-critical systems, secure systems, data-intensive analytic systems, artificial-intelligence powered systems, robotic and mechatronic systems, and much more.

6.2.2 Prospects

- Software engineering degree programmes will become more frequent in Hispanic America in the coming decade, as more people—from industry and academia—recognise that need for having an educated workforce developing and maintaining good-quality software systems. In 2011, the IEEE organised a workshop in Peru on computing curricula and nomenclature, with representatives from eight Latin American countries and also from Spain, USA and UK; they discussed nomenclature and approaches [37], and described common competencies for

graduates from computing sub-disciplines, plus specific outcomes of education for graduates of computer science, information systems, software engineering, computer engineering and information technology programmes.

- On the supply side: Universities and training organisations interested in developing education or training programmes on software quality or software testing can look at, select, adapt or adopt: bodies of knowledge, software engineering standards, certification schemes, curricula guidelines and recommendations, (academic) accreditation standards and regulations. This may impact degree programmes, postgraduate diplomas, master's and specialty programmes, continuing education and training.
- Industry and academia can collaborate more. For example:
  - Developing a common understanding of competencies, using frameworks such as SWECOM [22] or SFIA [38]
  - Sharing information on technical skills and knowledge needed for good performance on specific jobs
  - Validating what workplace-relevant, non-technical skills are more important for employers
  - Developing internships, dual education, co-op terms, study visits
  - Working together to support economic and community development in their areas
  - Showing organizations' outreach and social responsibility
  - Guest lecturing or workshop tutorials on practices, tools, standards and processes
  - Providing artefacts for reviews (inspections), testing and analysis
  - Opening suitably anonymised data on defects and productivity
  - Faculty performing research with professionals at companies
  - Companies' personnel undertaking postgraduate studies and developing theses on subjects relevant to their employers
  - Co-leading comparative studies on tools, methods, standards, etc.
  - Validating curricula development, updates or upgrades
  - Reviewing course development, alignment, update or upgrade—at the curriculum level or in continuing education contexts
  - Joint definition of integrative courses or projects for solving authentic problems in team-based efforts
  - Interacting with advisory boards related to academic programmes or schools
  - Sponsoring innovative graduation projects and providing incubation opportunities or mentoring
  - Jointly sponsoring groups or communities interested in software quality and software testing
- The academic communities in Latin America interested in software engineering are growing, and will probably develop more specialist tracks or symposia in regional conferences.
- More scientific, engineering and industrial research will be needed in the field of software engineering, particularly related to software quality, software

testing, software analysis, software integration, software aging and software maintenance.

As the 'digital transformation' of organisations progresses, more challenges—and *opportunities*—lie ahead for those in Hispanic America interested in software quality and software testing.

Acknowledgements I thank Stephan Goericke and Agustina Gay of iSQI for the invitation to contribute, Santiago Castaño of the HASTQB for sharing statistical data, Carolina Triana for documents on Choucair Testing, colleagues of the JIISIC and SLISW Latin American conferences on software engineering and my family—for their patience (one more time!).

References

1. Barlas, S.: How critical is the shortage of IT workers? IEEE Computer. **30**(5), 14–16 (1997)
2. Hoch, D., Purkert, G.: Secrets of Software Success. McKinsey & Company (2000)
3. National Research Council: Building a Workforce for the Information Economy. National Academy Press (2001)
4. Computer World, USA/North American editions. Years 1997-2000
5. Mata, F., Jofré, A.: Estudio de oferta y demanda del recurso humano en la industria de software [Study of supply and demand of human resources by the software industry]. Pro-Software (Inter-American Development Bank, Caprosoft, Procomer, FunCenat) (2001)
6. Brenes, L., Govaere, V.: La industria del software en Costa Rica [The software industry in Costa Rica]. Comercio Exterior. **58**(5), 303–311 (2008)
7. ACM/IEEE-CS Joint Task Force on Computing Curricula. The overview report: covering undergraduate degree programs in Computer Engineering, Computer Science, Information Systems, Information Technology, and Software Engineering. ACM Press and IEEE Computer Society Press (2005)
8. Sabin, M., Viola, B., Impagliazzo, J., Angles, R., Curiel, M., Leger, P., Murillo, J., Nina, H., Pow-Sang, J. A., Trejos, I.: Latin American perspectives to internationalize undergraduate information technology education. In Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE 2016). ITiCSE '16 Working Group Reports, July 09–13, 2016, Arequipa, Peru. A., Clear, E., Cuadros-Vargas, J., Carter, Y. Tupac (Eds.). Association for Computing Machinery (2016)
9. Marques, I.d.C.: History of computing in Latin America. IEEE Ann. Hist. Comput. **37**(4), 10–12 (2015)
10. GSMA The Mobile Economy Latin America and the Caribbean (2018)
11. World Economic Forum: The Future of Jobs. Employment, Skills and Workforce Strategy for the Fourth Industrial Revolution. World Economic Forum, Geneva, Switzerland (2016)
12. Experis: Tomorrow's Talent: Plugging the IT Skills Gap. Experis, London, UK (2013)
13. Capgemini Consulting: The Digital Talent Gap - Developing Skills for Today's Digital Organizations. London, UK, Capgemini Consulting (2013)
14. The Bureau of National Affairs: Building Tomorrow's Talent: Collaboration Can Close Emerging Skills Gap. Bloomberg BNA, Arlington, Virginia (2018)
15. Trejos, I., Cordero, A.: Learn-by-doing-collaboratively across the curriculum: Integrative projects at UCenfotec. In: IEEE World Engineering Education Conference – EDUNINE 2017. IEEE, Santos, BRAZIL (2017)
16. Naur, P., Randell, B. (eds.): Software Engineering. Report on a Conference sponsored by the NATO Science Committee. Garmisch, Germany, 7th to 11th (1968, October)

17. Finkelstein, A., Kramer, J.: Software Engineering: A Roadmap. In: Proceedings of the Conference on The Future of Software Engineering. ACM (2000)
18. Bourque, P., Fairley, R. (eds.): Guide to the Software Engineering Body of Knowledge Version 3.0 (SWEBOK). IEEE Computer Soc, Los Alamitos, Calif. (2014)
19. Gotterbarn, D., Miller, K., Rogerson, S.: Software Engineering Code of Ethics and Professional Practice v5.2. ACM & IEEE, New York, NY (1997)
20. ACM & IEEE Computer Society: Software Engineering 2014 - Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering. ACM & IEEE Computer Society (2014)
21. Integrated Software & Systems Engineering Curriculum (iSSEc) Project. Graduate Software Engineering 2009 (GSwE 2009) - curriculum guidelines for graduate degree programs in software engineering. Stevens Institute of Technology (2009)
22. IEEE. Software Engineering Competency Model Version 1.0 (SWECOM). IEEE Computer Society (2014)
23. https://asq.org/cert/software-quality-engineer
24. https://www.istqb.org/certification-path-root.html
25. Smerdon E.: An Action Agenda for Engineering Curriculum Innovation. 11th IEEE-USA Biennial Careers Conference, San Jose, California (2000, November 2–3)
26. National Academy of Engineering: The Engineer of 2020: Visions of Engineering in the New Century. National Academies Press (2004)
27. National Association of Colleges and Employers. Job Outlook (2019)
28. IEA. Graduate attributes and professional competencies, Version 3. International Engineering Alliance (2013)
29. ISTQB. Certified tester foundation level syllabus, Version 2018. ISTQB (2018)
30. Humphrey, W.: Introduction to the Team Software Process. Addison-Wesley, Boston, MA (2000)
31. Kruchten, P.: The Rational Unified Process: An Introduction, 3rd edn. Addison-Wesley, Boston, MA (2003)
32. Cherniss, C., Goleman, D.: The Emotionally Intelligent Workplace. Jossey-Bass, San Francisco, CA (2001)
33. Covey, S.: The Seven Habits of Highly Effective People. Free press, New York (1989)
34. ACM&IEEE/CS: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering. New York, NY, ACM & IEEE/CS (2004)
35. O*NET. O*NET Resource Center. See www.onetcenter.org/overview.html, www.onetonline.org/, www.mynextmove.org/
36. Parnas, D.: Software engineering programs are not computer science programs. IEEE Softw. **16**(6), 19–30. IEEE (1999)
37. Ramos, T., Micheloud, O., Painter, R., Kam, M.: IEEE Common Nomenclature for Computing Related Programs in Latin America. IEEE (2013)
38. SFIA: Skills Framework for the Information Age, SFIA 7 - The Complete Reference. SFIA Foundation, London (2018)

Further Reading

1. IPMA. Individual competence baseline for project, programme & portfolio management. Version 4.0. International project management association (2015)
2. PMI. Project manager competency development framework, 3rd Ed. Project Management Institute (2017)

The Future of Testing

Digging in the Past of Software Testing and Unearthing the Future

Kaspar van Dam

Abstract Many articles have been written on this topic. Many people have been trying to predict the future of software testing. Only time can tell who's right and who's wrong. Being a software tester today and a former archaeology student, I've decided to start digging in the past of Software Testing in an attempt to predict the future of this profession. What will change? What will stay the same? And what things from the past might resurface? Let's make a timeline starting around 20 years ago in 1998 (which is the moment when I myself started digging around in the world of computers and software) and try to stretch this all the way to 20 years in the future with 10-year intervals.

Keywords Software testing · Software quality · Software testing history · Software testing future

1 Introduction

Many articles have been written on this topic. Many people have been trying to predict the future of software testing. Only time can tell who's right and who's wrong. Being a software tester today and a former archaeology student, I've decided to start digging in the past of Software Testing in an attempt to predict the future of this profession. What will change? What will stay the same? And what things from the past might resurface? Let's make a timeline starting around 20 years ago in 1998 (which is the moment when I myself started digging around in the world of computers and software) and try to stretch this all the way to 20 years in the future with 10-year intervals.

K. van Dam
Improve Quality Services, Zwolle, Netherlands

S. Goericke (ed.), *The Future of Software Quality Assurance*,
https://doi.org/10.1007/978-3-030-29509-7_15

2 The Past

1998

Steve Jobs has returned at Apple just a year ago and presented the first iMac. Google is founded and the first MP3 is created. It's also the year in which MySQL and XML 1.0 are introduced. Windows 98 is released and Bill Gates gets a pie in the face. Seti@Home is started up and Internet Explorer becomes the most popular internet browser. The world gets to know blogs and it's also the start of ISEB Software Testing Certification (IEEE 829) and the third anniversary of TMap, which has changed the world of software testing quite a bit.

From the first day software was being developed, software was also being tested. However, around 1998 the first little steps were being taken in an effort to make software testing into a serious profession and people became aware quality assurance was something of importance in an IT project. But in 1998 a lot of software was still being released without any mentionable testing. At IT-related study programmes, software testing wasn't even part of the curriculum. But the introduction of TMap and other methodologies made an impact in the software development industry. People were trying to get a grip of what software testing was all about. They tried learning and discovering which steps to take to get software tested and get a grip on the quality of the software that was being developed. In the years that followed, software testing became an actual profession. However, there weren't yet any real requirements for the job. If someone could read and write he/she could become a software tester. If you failed at developing software you could always give software testing a shot. But most software testers back then didn't even have a background in IT. Strangely enough (looking back at it), this was actually considered something good. The consensus was that software testers should be strictly independent. Therefore they should not be hindered by any technical knowledge. They should focus on the functional requirements which were being written down in massive documents and by trying out newly developed software as an end-user, they should try and find out if these requirements were being met. In reality this often meant software testers considered it a sport to find as many bugs as possible. Some software testers might even remember the bumper stickers that were popular back in the days stating things like "Software Testing: You make it, we break it!" or "Every tester has the heart of a developer, in a jar on his desk".

2008

The world has changed. Computers and the World Wide Web are now commonplace. The first iPhone is barely a year old and Android and Chrome are being born. Facebook has been available to the general public for 2 years and no one can imagine a world without the Google search engine anymore. In 2008, both Spotify and GitHub are founded and it will be another year before the Bitcoin appears. TMap Next has been around for a few years now and ISTQB is celebrating its sixth anniversary. The Agile Manifesto is already 7 years old, but is now starting to get some feet on the ground. Also test automation is emerging more and more.

Most probably for most testers 2008 just went by like every other year. However, it has been quite an important period for the profession. Testing became more about involving people instead of endlessly scrutinizing the different documents and making it a sport to file as many bugs as possible in some bug tracking tool. From 2008 onwards, testers (and other people in the IT work field) slowly started embracing the agile methodology. After spending years and years making software testing into a serious, but especially an independent, profession all of a sudden, the profession started moving back to its origins. Because in the 1980s (and also before that) software testing wasn't considered a specialist job at all. A software developer wrote some software, tested it and went on with the development. As long as the system didn't crash and seemed to be doing what it was supposed to be doing it must be working as intended. With the introduction of Agile, SCRUM and later on things like DevOps, testing went back from being a strictly independent specialism to something that's part of the entire software development process. Looking back at it you may say the profession of software testing had lost its way for a decade or two and from around 2008 onwards it has found its purpose again. Software testing isn't about finding bugs. It isn't about being independent and having a developer's heart in a jar on the desk. It's about working together to create the best possible software. Software quality isn't about finding out what's wrong with the software (compared to what was written down in a functional design document). It's about the software meeting the wishes and demands of the person who will be using the software, the end-user. Instead of being this separate testing department spitting out bugs, software testers became part of the development teams. This also meant more and more software testers actually did get the heart of a developer . . . Until ca. 2008 it was said software testers tolerated boredom while developers automated it. When developers and testers started working together it turned out there wasn't any need to keep repeating the same manual testing tasks over and over. They could be automated! And many testers embraced this, they no longer needed to tolerate boredom! To many it must've seemed like test automation was born around 2008, in reality test automation is as old as software development itself. It was just forgotten (by most) for a period of time . . .

3 Present Day

We can't imagine a world without the internet anymore. Smart phones are part of everyday life and it's hard to even start comparing software development with what it was back in 1998. Hardly any IT project is done without at least some elements of the Agile way of working and things like DevOps and Continuous Development and Continuous Integration are now becoming commonplace. Words like communication and teamwork are (almost) considered being magic words in the world of software development nowadays. So, what have we learned from the past, implemented in our present and what can it tell about our software testing future?

Software testing has come a long way, but right now developments in this field of work are gaining momentum. Things are changing. Fast. And the people involved need to change along with it or be left behind. Most testing professionals we see today are nothing like the software testers from back in 1998. A question however could be: aren't we changing too fast? Aren't we forgetting the things we learned during the last 20 years (and before that)?

Today test automation may very well be the most sought after expertise when it comes to quality assurance in software development. Software development is all about continuously producing sufficient quality software. Time is everything, we want to release new functionality as soon as possible without being hindered by exhaustive manual testing. Therefore test automation is the only way to go. However, as some people may have thought for a while, tooling will not replace the software tester entirely. Simply because tooling isn't able to test software at all, it can just execute certain pre-programmed checks on the software. This means it's still up to a specialist to come up with the right test cases (and then decide if they should be automated or not). For software testers this means they should shift their attention from just software testing (e.g. TMap) to a much broader area. They should know about automation, tools and frameworks. But they should also keep a focus on their (testing) skills, including soft skills. Instead of following some testing technique out of a book, a software tester today should possess a certain mindset in which they understand people, product and process (P3). And besides just looking at functionality testers are often expected to keep an eye on non-functional requirements like performance, security and reliability as well. The shift from waterfall to agile software development has had a major impact on everyone involved with IT; however, it may have had the biggest effect on software testers. And I think this is just getting started . . .

4 The Future

2028

Let's just try and predict the future. At least, when it comes to software testing. Continuing with 10-year jumps we go to 2028. If we take a look at the origin of software testing and how it has developed the last few decades, than where do we expect the expertise to go next? From what mistakes have we learned? What things will we still be doing in 2028? What will probably change?

One thing is certain: We cannot predict the future accurately but we can be pretty sure that a software tester 10 years from now will still have to be flexible. During the last few decades software testers have changed from being a bit of rigid personalities who required to be independent on their own little testing island into possibly the most flexible people participating in any IT project. It's often the tester who makes the connection between businesspeople and IT people. It's often the software tester who starts working together with business analysts to change the way

requirements are written down (in order to get a better shared understanding of what needs to be built and tested by the development team). A software tester is required to understand what the business is talking about but should also be able to spar with a developer on how to implement some code and how to automate the tests that should check if the code is actually working. Therefore a software tester in 2028 might not even be 'just' a software tester. The role might be more about being the linking pin between business and IT and being the quality conscience of the entire BizDevOps team. This is something entirely different than back in the days. In 1998–2008 everyone with some analytical skills could become a software tester. If you failed at being a good software developer, you could still make a career move to software testing. But in the upcoming 10 years the role current software testers occupy might even become the most challenging role within software development. Being the quality conscience, you should always be adapting yourself to new realities. You should constantly be changing strategies and always keep track of things changing. Both from a business perspective and a technical perspective.

And even though this may sound like a lot of fun to many people, changing is actually the hardest thing to do for a human being. Simply put, our brains aren't built for coping with change. We were built to be organized, to find structure and patterns. To do things exactly like we always did them. To understand this we need to look a bit further back than 1998. More than 10,000 years ago we humans were still hunter-gatherers. The only thing we needed to worry about was surviving and the best way to do that was to follow strict patterns. We often went to the same places to hunt or gather food and we could trust our brain would warn us if anything was out of the ordinary. Because that's how the human brain has developed. It's always more or less in a relax mode, until something changes, when a chemical called cortisol is released, also known as the stress hormone. It induces a state known as the fight-or-flight reflex. Adrenaline starts pumping, our field of vision narrows, muscles contract. We get hyper-focussed on the one thing that has changed. Was it a threat? A predator? An enemy? Some other form of danger?

Nowadays we're working in our twenty-first century offices, but our brains are still the same as they were 10,000 years ago. They haven't yet adapted to this new environment. Evolution doesn't like to be rushed. So, even though we might think we embrace change. Even though we think constant change makes our day-to-day work more fun. Our brains don't like it at all! Knowing this, it's no wonder people on the work floor are clinging to old patterns. There is a reason we often still prefer to have a structured Ways of Working, preferably documented in some handy Excel checklist ... There's a reason we may feel lost when things all of a sudden change when we're in the middle of something. The Agile way of working is all about adapting to change. But we should be more aware that this is actually something we humans aren't really good at! (And if any reader is now shaking his/her head mumbling 'no, no, no, that doesn't apply to me!', trust me: it applies to you as well. It's simple biology! [1, 2]).

So, will we be abandoning the Agile way of working in 2028? I don't think so. However, I do suspect we will change it quite a bit. When looking at software development teams I see a lot of developers struggling with all these Agile meetings,

things changing all the time. When they just started putting a shoulder to the wheel with some technical challenge there's someone at their table inviting them to some refinement or 3-Amigo session because some new insights have popped up and everything needs to be different. Once again! The problem is, this process of continuous changing will remain in the future. I think it will even get worse. Technology had many limitations in the past, but they're vanishing really quickly. In the past you could ask an end-user to wait a year or two before expecting some new functionality. But in 2028 (or really even today already!) when an end-user wants something of the software, he/she expects it to be there tomorrow. Or possibly even sooner. And if not: there's always someone else who can make it happen that fast. This means in 2028 we IT people need to be even more flexible and more adaptive to change than we are today. And someone needs to make sure the people writing the actual code can keep their focus on the code. To make sure the developer knows where to pick the berries and where to hunt a rabbit without constantly having to look over his/her shoulder for some predator approaching, figuratively speaking. And that someone might very well be the person we call a software tester today.

This means software testers have a lot to do in the upcoming 10 years. Because as Darwin stated: "It is not the strongest of the species that survives, nor the most intelligent; it is the one most adaptable to change." If you still want to be part of the game, if you want to survive in the world of software testing, a software tester should keep learning, training and adapting to change. Even though our very own brain is resisting this. The difference between a 1998 software tester and a 2028 software tester may be comparable to the difference between a 'common' soldier and a commando. A soldier is trained to do as he's being told, a commando however is trained to stay alive and accomplish his mission by constantly adapting to change. This doesn't mean one of these two is better than the other, but it does require a different type of person, a different training and especially a different mindset to be a commando instead of a 'common' soldier. The software tester of 2028 might very well be the commando, while the software developer may remain a soldier.

In this (brave?) new world of software development in 2028 I don't think we'll still have the job title 'Software Tester'. The role will be about so much more than just testing the software. It might even be probable that the actual testing of software isn't the main focus anymore in 2028. Nowadays a lot of manual testing is already being replaced with automated testing. However, as stated before, test automation today is just about executing pre-programmed checks on the software. I'm not sure if we're there already in 2028, but I do believe artificial intelligence will influence test automation a lot in the future. Which means that in time a computer might actually be able to at least do a small amount of actual testing of software instead of just checking some predetermined things. This would mean that the actual testing of software would become even less important within the job description. So what shall we call this 'software tester' in 2028? Maybe 'Quality engineer'? Or 'Change specialist'? Maybe even 'Dev Commando'? I personally wouldn't vote for any of these new job titles. I think in time some new term will pop up describing this new

role. After all, back in 1998 who would have thought we would be having business cards today with terms like 'Scrum-master' or 'Product-owner' . . .

2038

It may be near impossible to predict the future 20 years ahead. But let's give it a try. In 2038 we're probably driving autonomous cars, and robots may very well be part of everyday life. Artificial Intelligence will be a lot more intelligent than it is today and may even be replacing knowledge workers. Will there still be software development as we know it? And will there be software testing?

I personally believe we will still be needing people to develop software even though it's probable we'll be able to produce a lot more software with a lot less people. It is possible software has become intelligent enough to be testing itself. Software may be continuously running self-diagnostics which indicate when something's going wrong and the software in question might even be able to fix itself, at least to a certain degree.

But I don't think computers and robots will have replaced everyone involved with the development of software. Simply because of one thing: Artificial Intelligence is only intelligent about certain things, but really unintelligent when it comes to some other things . . . This is clearly visible today, but I don't expect it will be all that different in 20 years from now. Take a look at current-day tools like Google Assistant or Siri. These AI's know a lot more than any human being knows (because they have an endless supply of information constantly at their disposal). Some robots today are amazing at interpreting their surroundings and figuring out what's expected of them. Current prototypes of autonomous driving cars may very well already be safer than human drivers. However, even with all this computer power and all this data and intelligence there's still one thing at which every AI sucks: understanding human behaviour.

A great example is Honda's humanoid robot Asimo, which was developed a few years ago. In every single way this was a great feat of engineering. However, during a demonstration it failed horribly because of one simple misunderstanding of human behaviour. The robot didn't understand why people would want to take pictures of it and thus concluded that when people were raising their camera or mobile phone to take a picture, they were raising their hands to ask a question. The robot froze, repeating over and over "Who wants to ask Asimo a question?" [3]. And this interpretation of human behaviour is something current AI still doesn't know how to do, even though it's something we humans find very easy! We immediately understand what's happening when someone is hanging out of the window of a train that's about to leave to hug someone outside. They're saying goodbye. Pretty straightforward, right? However, an AI might mistake it for someone trying to pull another person out of the train. There might be something wrong in the train, an evacuation might even be necessary . . . . I don't believe AI will be much better at interpreting human behaviour in just 20 years from now. And therefore it's very likely we'll still be needing humans to develop software that will actually do what an end-user is expecting from it. And when we're still building software we'll also still need someone to act as the earlier mentioned quality conscience. Someone who's

able to translate what an end-user wants from the software and who's capable of telling if the software is actually serving its purpose. And even though computers might take over a lot of work in the software development industry, I think there will still be a lot of work to be done by human beings, especially when it comes to quality assurance and software testing. However, it will most probably be a completely different job compared to the job today.

5 Conclusion: This Is the Future

To conclude things. During the last few decades the world of a Software Tester has changed dramatically. Looking at the history of software testing and developments in the work field today, I don't expect the upcoming 20 years will be any different. Just like many software testers today can't even imagine what they were thinking back in 1998, I believe that in 2038 people will have a hard time imagining what software testers are doing today. Things like Artificial Intelligence will change the software development industry completely. And it is expected AI will, in time, be a part of about every piece of software. We will find ways to have software adapting itself to the world around it. The software can change according to the needs of people using it and it might even become able to test itself and fix possible bugs it detects in its own lines of code. However, we will still be needing people to act as a quality conscience. People who can make sure software being developed is meeting the requirements of human beings using the software. Human beings whose behaviour will most probably remain a mystery to even the most intelligent AI. It might even be the software testers, or whatever we will call them in the future, who could one day prevent AI from becoming too intelligent (Skynet, anyone?). But what does all this mean for current day software testers? What should we be focussing on? For what things should we be preparing ourselves?

First off, we should keep investing in test automation. It can only be expected test automation is something that will stay and it will keep improving itself until eventually test automation frameworks will actually be able to do (some) testing instead of just checking some pre-programmed test. But, more importantly I believe we should keep investing in communication and collaboration. It's incredibly difficult to get a good understanding of what an actual end-user of a piece of software wants and needs. And it might be just as difficult to make sure everyone involved with creating that software shares the same understanding of what it is the software should be doing to meet the requirements from this end-user. Even today software testers should already act as a quality conscience and make sure software is meeting the requirements. This means the tester is already a linking pin between business and IT today, and will be this linking pin even more in the (near) future. Software testers today might consider broadening their expertise and start looking more at non-functional requirements like performance and security. These things are already really important today but will be even more important in the (near) future. We don't want our current-day laptop to be hacked, nor do we want it to fail at

certain tasks because it's incapable of executing some calculations within a required time frame. However, when something similar would happen to an autonomous car driving at a great speed on the motor way in 20 years from now it would predict disaster! It's up to the software testers of the future to make sure chances are as small as humanly possible that something like that can happen with software that might be responsible for human lives. I don't know what the future of software testing will be bringing us, but I do know it will require us to keep staying ahead of the game. To keep investing in knowledge, to keep improving in our skillset and to remain able to adapt to change. Whatever change that may be. Only then will we be able to survive in this future world of software testing!

P.S. Anyone up for a cup of coffee somewhere in 2038 and a look back with me at the things I've written down here?

Acknowledgments The author would like to thank Joris Meerts, Pascal Maus, Pieter Withaar, Piet de Roo, Huib Schoots and Berry Kersten for their input and feedback on this chapter, which was loosely based on an earlier published interview with some of these people.

References

1. Gray, K.: What Goes on in Brains: How an Understanding of Neuroscience Makes a Difference When you Advocate Agility; P3X Conference November 8th 2018, London (2018)
2. Levitin, D.J., Luke, D.: The Organized Mind. Penguin Group US, New York (2014)
3. https://www.theverge.com/2013/7/6/4498808/honda-asimo-disappoints-when-it-confuses-phones-for-hands. Accessed June 19 2019

Subconscious Requirements: The Fright of Every Tester

Hans van Loenhoud

Abstract Subconscious requirements are requirements that stakeholders do not explicitly ask for, but still expect to be present in a system. Such requirements are problematic to testers as they will easily be overlooked and not be included in the specifications and therefore may not be tested at all. Over time, less notable (non-) functional requirements tend to end up as subconscious. The development of ever more complex IT solutions adds to the occurrence of subconscious requirements. They do need to be tested, as ignoring them will lead to expensive rework or rejection. Due to the lack of specifications, tests can only rely on experience-based techniques. To acquire the necessary experience, testers can turn to proven techniques from the requirements engineering discipline.

Keywords Software testing · Software quality · Subconscious requirements · Software tester

1 Introduction

As testers, we all have the same experience once in a while: a few days after the go-live of a new release, a user comes up with a serious bug. Usually, it is not about the core functionality, because everything works fine there, but more likely an issue on non-functionals like usability, security, or performance. That may seem less important, but in fact such issues can be quite nasty, difficult and expensive to solve. And then, the project manager comes to *your* desk and asks: "Why didn't you find this bug in your tests? You spent more than xxx hours on testing and still the #!@! thing has bugs in it!!!"

If you are a beginner in testing, you turn red in the face, try to hide under your desk and consider looking for another job. If you are an experienced tester, you remain calm, look the project manager right into her eyes and say: "Nobody told

H. van Loenhoud
Taraxacum, Kockengen, Utrecht, The Netherlands

S. Goericke (ed.), *The Future of Software Quality Assurance*,
https://doi.org/10.1007/978-3-030-29509-7_16

me that this was a requirement, so how could I know that it needed testing? Go to the BA and ask why this has not been specified in the first place." So, off goes the project manager, but still you feel uncomfortable about the situation.

And although you did not know about this requirement, you still consider the absence of defects in the system to be your responsibility, no matter that you learned from the "seven testing principles" that software never will be 100% defect free (see [1]).

What actually happened was that you were struck by a subconscious requirement: a requirement that everybody, at least at the user's side, knows about, but that is considered to be so self-evident that no one takes care of telling, capturing, or documenting it. In the remainder of this chapter, we will look into a bit of theory about these subconscious requirements and see if we can find ways to deal with them.

2 What Are Subconscious Requirements?

The concept of subconscious requirements stems from the work of Noriaki Kano, professor at the Tokyo University of Science. In the early 1980s Kano laid the foundation for a new approach to modeling customer satisfaction and developed a customer satisfaction model (now known as the Kano model) that distinguishes between essential and differentiating attributes related to concepts of customer quality [2].

Although originating from marketing research, the Kano model has become one of the fundamentals of business analysis and requirements engineering in IT, see for instance [3, 4]. The Kano model (see Fig. 1) discerns three categories of factors that are relevant for customer satisfaction:

- Performance factors

  Performance factors relate to features that the customer explicitly asks for. They have a linear relationship with customer satisfaction: the more these factors are present in a product or service, the higher the satisfaction. Kano called this "one-dimensional quality," and in requirements engineering, they are usually referred to as "satisfiers" or "conscious requirements."
- Basic factors

  These are factors that customers implicitly expect to be present in a product or service. This is what requirement engineering calls "subconscious requirements," because customers consider these features self-evident or even are unaware of them. They are also called "dissatisfiers": when such features are present, no one will notice them, so they don't contribute to the customer's satisfaction. However, when they are missing, the customer will consider the product or service to be unusable and will be very dissatisfied. Kano used the term "must-have quality."
- WOW-factors

  The third category concerns features, that the customers do not consider to be possible, so they will never ask for them. Therefore, they are called "unconscious requirements." If they are absent in a product or service, the customer will

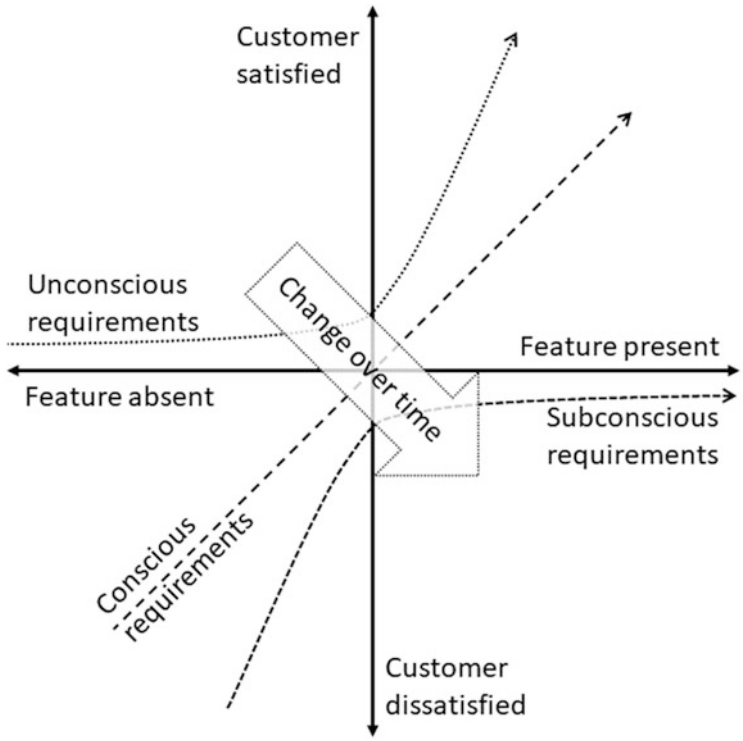


Fig. 1 The Kano model

not notice it, so it does not affect satisfaction. On the other hand, when they are present, the customer will be pleasantly surprised: "delighters," or in the terminology of Kano, "attractive quality."

One interesting observation by Kano was that requirements tend to change over time. New features start, almost by definition, as unconscious requirements. As customers discover, experience and like a new feature, it becomes a conscious requirement that is explicitly asked for. Gradually, as all similar and competitive systems implement the same feature, customers forget that originally systems did not have such a feature and start to take it for granted, turning it into a subconscious requirement. That is why many systems contain features that users consider as indispensable without knowing why and thus without explicitly asking for them.

A good example is the camera function on mobile phones, where this process took less than 20 years. The first time a camera was introduced as part of a phone, most customers were puzzled: no one had asked for this unconscious feature. But they liked it as a delighter and all brands started to implement it in their phones, turning it into a conscious requirement. Nowadays, when buying a new

phone, everybody takes for granted that it will have a camera, so it has become a subconscious requirement.

Conscious requirements are the easiest category to deal with for both the analyst and the tester. They are explicitly mentioned by the stakeholders, so they will clearly be present in the specifications of the system under test (assuming that the business analysts and designers did a good job).

Unconscious requirements are also relatively simple to test. Stakeholders did not ask for them, so the designers have incorporated them deliberately in the system and will have added them prominently to the specifications. The only problem for the tester is that these requirements are based on assumptions of the designers about the system's attractiveness for the customer. These assumptions may be wrong, so they should be tested too. Unfortunately, designers often forget to specify their assumptions, so as a tester, you should ask for them. However, even if an unconscious requirement remains untested and subsequently fails in production, the users will not complain because they did not ask for the feature in the first place (but be aware that such failure may involve extra risk that actually does need testing).

The challenge for the tester lies in the subconscious requirements. The stakeholders do not ask for them, so they are quite likely to be missing in the specifications, provided by analysts and designers as input for the tester. But they still need to be tested, because failure of a system to meet the subconscious requirements will almost certainly lead to rejection. Therefore, the tester faces the task of testing features that are not present in the specifications.

In practice, this usually does not concern the main functionality of a system, being described in detail in the specifications and thus belonging to the conscious requirements. Often, subconscious requirements relate to some "nitty gritty" functionality details and exceptions, to user-related quality criteria like usability, security, or performance, or to implicit technical, infrastructural, organizational, and legal constraints. Of course, depending on the domain knowledge of the analysts and designers, a certain part of all subconscious requirements will still be present in the specifications of a system. But on average, one can be quite sure that a significant portion is missing. If a certain requirement is missing from the specifications, the chance that it is implemented in the system is minimal.

3 How to Deal with Subconscious Requirements?

The problem for the tester lies in the (mostly subconscious) requirements that are missing from the specifications of the system under test. In such a case, common black-box testing techniques cannot be used, as they depend on the analysis of the specifications in the test basis.

White-box techniques, depending on the documented structure of the system are equally inapplicable, as the structure of a system normally is derived from its specifications.

The only thing that is left is to apply experience-based techniques: error guessing, exploratory testing and checklist-based testing (see [1]). In experience-based testing, tests are derived from a test basis consisting of the knowledge and experience of the testers themselves. Therefore, the big question is: how can a tester acquire the necessary, relevant knowledge and experience?

The only answer to this question is: as a tester, apply requirements engineering techniques that are suitable to uncover subconscious requirements. Such techniques can be found in the IREB body of knowledge (see [3, 5]):

Observation Techniques

- Field observation

  This technique is about observing the users during their work in their usual environment without interfering. The tester may notice certain unexpected or undescribed behavior, strange sequences in activities, or manual sidesteps. These are strong indicators for subconscious requirements.
- Apprenticing

  Apprenticing goes a step beyond field observation. Here, the tester conducts a short hands-on training in the environment in which the system is to be deployed. Key users teach him their work processes to better understand the domain. When actually participating in the work to be done, subconscious requirements will easily come to the surface.
- Contextual inquiry

  Contextual inquiry is an iterative, field data-gathering technique, studying a few carefully selected users in depth to arrive at a fuller understanding of the work practice across the entire user base.

Artefact-Based Techniques

- System archaeology

  System archaeology is a technique to gather information regarding a new system from the documentation, user interface or code of a legacy or competitor system. Of course, most of the requirements found by this technique will be present in the specifications of the system under test. The remainder will either not be relevant in this particular case or turn out to be subconscious.
- Perspective-based reading

  In this technique, the tester uses a specific perspective—in this case looking for unfamiliar requirements—in order to retrieve information from documents that are relevant for the domain in which the system will be used, e.g., process descriptions, company regulations, and applicable legislation.

These techniques have one major drawback: they are usually quite time consuming. However, this time investment is inevitable to acquire the thorough domain knowledge necessary to recognize subconscious requirements and to design tests to mitigate related risks.

A quite different approach to find and test unconscious requirements is Specification by Example (see [6]). This works best when applied by the whole development team throughout the development process, as it promotes shared understanding and trust within the team. But even for testing itself, it can be very useful.

In this approach, test cases are not derived from specifications, but from an iteratively growing set of real-life examples of inputs and outputs from the work processes that the system is intended to support. Specification by Example will yield tests for all requirements, conscious, unconscious, and subconscious alike, and may, in the long run, be more time-efficient than the techniques mentioned earlier. An additional benefit of Specification by Example and similar approaches like Behavior-Driven Development (see [7]) is that the resulting set of examples/test cases ultimately forms a complete and up-to-date summary of the system and its behavior, the so-called "living documentation."

Yet another approach that deserves mentioning here is Design Thinking (e.g., see [8]). Several variants of Design Thinking exist, but they all focus on understanding the true needs of the users by introducing human-centered techniques like Persona's and Empathy Mapping. Especially the exploration of the "pains" and "gains" of different user groups can help the tester identify previously undetected subconscious requirements.

A common technique from most Design Thinking variants is prototyping. Prototyping offers future users the opportunity to gain hands-on experience with early versions of a new system and to provide feedback on its behavior, thus easily uncovering subconscious requirements. Especially the so-called low-fidelity prototypes like UI-sketches, storyboards, and customer journeys are very useful in this respect. Like Specification by Example, Design Thinking is in fact a whole-team effort, involving analysts, designers, developers, and testers.

4 What About the Future?

One might hope that by the growing professionalism and maturity of IT analysis and design, the tester's problems with subconscious requirements will gradually disappear, as nowadays most product owners, business analysts, and requirements engineers learn about techniques to elicit and communicate them.

However, trends in IT work in the opposite way. To illustrate these trends, the Cynefin Framework (see Fig. 2) is very useful.

The Cynefin framework [9] is a conceptual framework used to aid decision-making as a "sense-making device." It was created in 1999 by Dave Snowden working for IBM. Cynefin draws on research into systems theory, complexity theory, network theory, and learning theories and offers five decision-making contexts or "domains":

- Obvious
- Complicated
- Complex
- Chaotic
- Disordered

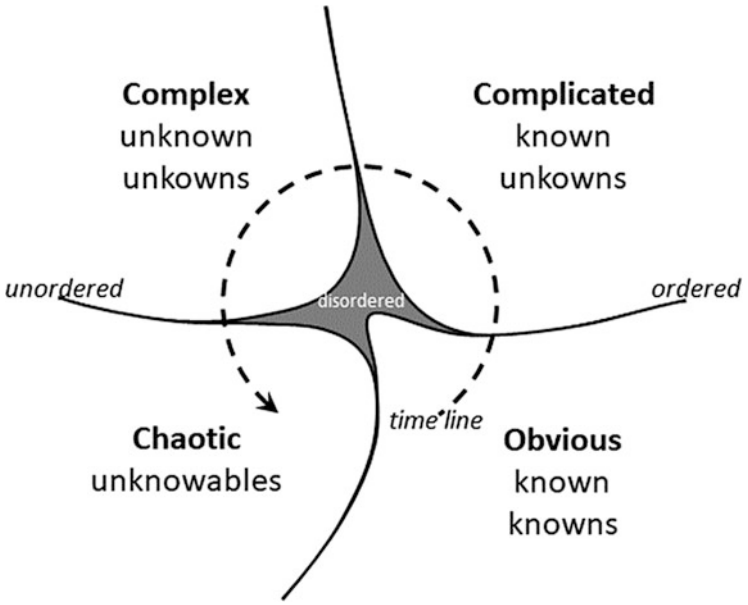


Fig. 2 The Cynefin framework

Although initially intended for decision-making, it can also be used to categorize IT projects with respect to the level of uncertainty that they must deal with (ignoring "disordered," a temporary state that changes into another after clarification):

- Obvious projects develop IT systems that offer support for clear, straightforward business processes, like order processing or accounting. Everything relevant to this development is known in advance ("known knowns"), so even subconscious requirements will be documented in the specifications. In these projects, developers and testers can easily rely on best practices.
- Complicated projects are, as the name implies, a bit more complicated. They often relate to the integration of different business processes, for instance, the development of ERP systems. There, developers and testers may be confronted with unexpected and subconscious requirements that are not present in the specifications. However, the type of requirements itself is known ("known unknowns"), so with a sufficient amount of domain knowledge testers will be able to deal with them by choosing from existing good practices.
- Complex projects are projects that develop IT systems for innovation, with well-known examples like Spotify and Uber. Here, developers and testers face "unknown unknowns." Cause and effect can only be deduced in retrospect, and there are no right answers upfront. Development teams will have to develop their own emergent practices to deal with the inherent uncertainties. Design Thinking offers clues to find the inevitably missing subconscious requirements.

- The most "problematic" category is that of the chaotic projects: not chaotic in the sense of bad project management, but chaotic in relation to the topic that has to be developed. This is about the "unknowables." The team explores completely new fields of IT, like the Internet of Things, Artificial Intelligence or machine learning. In these kinds of projects, not only the exact features, but even the future users are uncertain. But anyhow, these systems will interact with users, so the chance of meeting subconscious requirements after implementation is 100%. The only thing that can help you there may be a quote from Steve Jobs [10]: "Have the courage to follow your heart and intuition." As a tester, you will have to detect your own subconscious requirements and make them conscious in your test cases.

The Cynefin framework can also be read as a timeline. In the early decades of IT, most projects were in the obvious quadrant, automating "islands" of single, strictly demarcated business processes. Somewhere from the 1990s onward, we started trying to integrate these islands, and projects became complicated. With the breakthrough of internet and mobile apps, we saw the arrival of complex projects that created and triggered completely new business processes.

As IT extends its scope beyond the original domain of computers, invading products like cars, refrigerators, light bulbs, or weapons, the part of chaotic projects is growing rapidly. So as a tester, be prepared to deal with the unknowables; you will find more and more subconscious requirements on your path. Consider, for instance, the requirements for a "friend-or-foe" system in an autonomous combat robot, let alone the challenge of testing such a system . . .

5 Conclusion

Subconscious requirements are requirements for a feature that users in a certain domain take for granted or only become aware of when it is not implemented. Such requirements are easily overlooked, even by experienced analysts and designers, so there is a good chance that some of them will be missing from the specifications for a system.

Testers still have the responsibility to test the relevant subconscious requirements, but fear overlooking some of them, because they cannot rely on the specifications. The absence or failure of such a feature will often seriously affect the system, leading to expensive repairs or even rejection. To test subconscious requirements, testers cannot rely on their commonly used specification- and structure-based testing techniques, so they switch to experience-based techniques.

This raises the question of how testers should acquire the necessary domain knowledge and experience. In the first place, testers should (be able to) apply observation and artefact-based techniques from the requirements engineering discipline. Such techniques are especially suitable for finding subconscious requirements. Learning and applying these techniques is very relevant but will significantly add to the workload of the tester.

Agile approaches like Specification by Example, Behavior-Driven Development, and Design Thinking also significantly reduce the risk of overlooking subconscious requirements, as they heavily rely on real-life examples, customer empathy, and early feedback. These approaches work best when applied by the whole team throughout the entire development life cycle, so they are not suitable for testing in isolation.

Subconscious requirements are here to stay. The Kano model shows us that many requirements in the long run will end up as subconscious ones. From the Cynefin framework we learn that in the future, even more development will take place in the complex and chaotic domains, where the "unknown unknowns" and "unknowables" predict many subconscious requirements.

The time has gone for testers to derive their tests from a solid and extensive test basis. The future of testing will be predominantly experience based!

References

1. Olson, K., et al.: Certified tester foundation level syllabus version 2018. International Software Testing Qualifications Board. https://www.istqb.org/downloads/send/51-ctfl2018/208-ctfl-2018-syllabus.html (2018)
2. Kano, N., et al.: Attractive quality and must-be quality. J. Jpn. Soc. Qual. Control (in Japanese). **14**(2), 39–48 (1984)
3. Stapleton, P.: Agile Extension to the BABOK® Guide V2. International Institute of Business Analysis and Agile Alliance, Toronto, ON (2017)
4. Frühauf, K., et al.: IREB Certified Professional for Requirements Engineering - Foundation Level – Syllabus Version 2.2.2. https://www.ireb.org/content/downloads/2-syllabus-foundation-level/ireb_cpre_syllabus_fl_en_v222.pdf (2017)
5. Häusser, D., et al.: IREB Certified Professional for Requirements Engineering – Advanced Level Elicitation – Syllabus Version 2.0.1. https://www.ireb.org/content/downloads/7-syllabus-advanced-level-requirements-elicitation/cpre_elicitation_al_syllabus_en_2.0.1.pdf (2019)
6. Adzic, G.: Specification by example – How successful teams deliver the right software. Manning Publications, Shelter Island, NY (2011)
7. North, D.: Introducing BDD. https://dannorth.net/introducing-bdd/ (2006)
8. Alliance for Qualification: Design thinking foundation syllabus. https://isqi.org/nl/en/index.php?controller=attachment&id_attachment=27 (2018)
9. Snowden, D.J., Boone, M.E.: A leader's framework for decision making. Harv. Bus. Rev. **85**, 69–76 (2007)
10. Jobs, S.: You've got to find what you love. https://news.stanford.edu/2005/06/14/jobs-061505/ (2005)

The Why, How and What of Agile Transformations

Rini van Solingen

Abstract Agile can be compared to fitness. It means being fit enough as a team, department or organisation to be able to deal with all circumstances. Being able to react rapidly and nimbly when the situation demands it. And that is a very important skill in a time of digitalisation, disruption and rapid change. In this chapter, we explain the why, how and what of agile transformations, introduce a step-wise approach to undertake such transformations and discuss the most common pitfalls observed in practice.

Keywords Agile methods · Agile management · Agile project management · Agile transformation

1 Introduction

Agile is a mindset that embraces change, and is about delivering results quickly and learning from that. Agile working is about giving autonomy to people and teams, with clear room for decision-making and a great deal of self-organisation. Continuous improvement is paramount, as well as making gradual efforts to generate even higher customer value and surpass existing performance. Learning step by step and improving by doing. Delivering results and learning how to improve, together as a team. That is agile.

Agile works best in situations where a lot is changing and still being discovered. Ideas may have been formulated in advance, but a great deal still needs to be reflected upon, learned and changed. Making a plan in such a situation only makes limited sense, because things always turn out differently than expected. A clear goal

Translated from the Dutch Original book: "AGILE", © 2018, Rini van Solingen & Management Impact - translation by tolingo GmbH, © 2019, Rini van Solingen.

R. van Solingen
Prowareness, Delft, The Netherlands

S. Goericke (ed.), *The Future of Software Quality Assurance*,
https://doi.org/10.1007/978-3-030-29509-7_17

is necessary, but how it is reached is largely left open. And even the goal can be regularly reviewed, since that, too, may well alter in our rapidly changing world. And the more agile you are, the easier it is to deal with change.

The starting point of this chapter is that above all agile is a broadly applicable mindset that will find its way into many different environments. And this isn't such a crazy idea. After all, society is rapidly accelerating as a result of digitalisation and new ways of working together. Agile helps you, in cooperation with others and in small, clear steps, to achieve objectives that can be adjusted at any time.

This makes agile particularly suitable for what we often call 'knowledge work': cooperation between people in which the work itself and the results are often volatile, uncertain and consist of information, data and the like. Knowledge work is non-physical and is therefore fundamentally faster than physical work. After all, you can send messages, documents and files to the other side of the world in digital form in seconds (and for free!). This acceleration makes hierarchies within organisations unsuitable for rapid operational decisions. The speed and dynamism of change are simply becoming too great to ask permission from the boss each time. As a result, operational decisions and choices are increasingly made at the operational level, usually in teams that are allowed to organise themselves.

Agile working is, as such, a response to a rapidly changing and complex world. And it turns out to be effective rather than a short-lived hype. Many organisations are actively engaged in increasing their agility—whether they are large or small, commercial or public, young or old, technical or administrative in nature. They all struggle with the dynamism of the outside world, and they all see many advantages in making their working methods more agile and nimble. How they do this will vary from one organisation to another. It depends on their situation, their customers and their people. But striving for faster results and more flexibility is a uniform change in almost any organisation.

The essence of agile working lies in the acceptance of the fact that, as far as the future is concerned, it is never exactly clear what you can achieve and when. The reality is that so much is changing that we cannot actually make agreements far in advance. An important personal threshold for achieving agile working is therefore daring to let go of the idea that a detailed plan is necessary in order to be successful in complex situations. Learning to trust that taking the first step is the most important, because it is only during that first step that you will discover what the best next step is. Work experimentally and step by step. Agile teams do not plan too far ahead, and they jointly deliver results at a gradual pace within short periods of time, with the aim in particular of learning from each step. Learning how to do things better, learning what customers really need and thus discovering together where the highest customer value lies.

The structure of this chapter is the following. In Sect. 2 of we present the key characteristics of agility: iterations, validation and step-wise improvement. In Sect. 3 we present the key dimensions along which to decide whether agile ways of working make sense, depending on the amount of unpredictability. Section 4 introduces the reasons why more and more organisations start to implement large-scale agile transformations. In Sect. 5 we introduce a step-wise approach to

use when leading such a transformation. Finally, Sect. 6 presents the seven most common pitfalls observed in practice during agile transformation. We round off with a short conclusion in Sect. 7.

2 Agile Is About Short-Cyclic Working and Iterations

The essence of agile is easily explained by the difference between a submarine and a dolphin (Fig. 1). Project-based working is similar to a submarine: the boat is invisible and dives under water. It can stay there for a long time—just like a big project. Only towards the end do you become restless; as the deadline approaches, urgency and activity increases. A submarine comes up at the end with the result. For the first time. You then hope that everything is okay. That customers will be happy and that your result will prove valuable. Unfortunately, in practice things are different. The hope then often turns out to be deferred disappointment. And this makes sense, because this is when the very first feedback is received. You find out about everything that's not right at the same time. And unfortunately, you don't really have time to do anything about it anymore. After all, the project is over and the budget is spent. This submarine is sometimes also called the *see-you-later* model.

The alternative is the dolphin. A dolphin also dives under water. But a dolphin soon rises up to the surface again. This is because dolphins need air. With the dolphin approach you dive under water, but you quickly emerge with the first result. It is of course smaller than the total result you have in mind, but it is something you can test at least. You can test if it has value, test if it works and test if it does contribute to making the goal a reality. In this way you discover whether it delivers

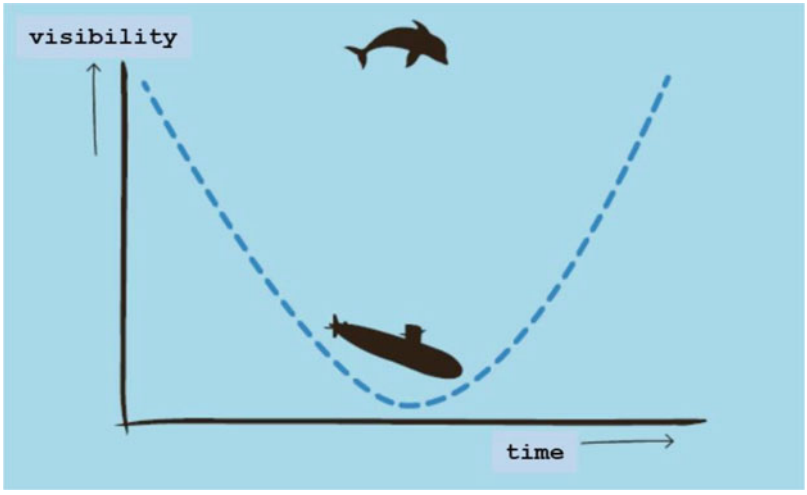


Fig. 1 Submarine vs. dolphin

the value you expect and you get feedback on what is sensible and what is not. With this knowledge, the dolphin dives under water again and emerges a little later. A dolphin works with so-called iterations or sprints (repetitions)—always diving and surfacing again. Breathe, check if you're still going in the right direction or decide to make adjustments. And then quickly dive under the surface again and re-emerge. The dolphin approach is also called the *see-you-soon* model. Try to deliver results in everything you do as quickly as possible and ask for feedback. You'll see that you'll get results sooner and that you'll also get to know much sooner which parts of your original plan aren't needed at all. And that's how agile speeds things up. It's not about working harder, it's about working smarter. Discovering what you don't have to do because it has no value, saves a lot of time. And it gives you the opportunity to deliver earlier or to deliver additional value.

The two different approaches function in a totally different way:

- It's finished at the end versus it's always finished.
- Feedback at the end versus feedback from the beginning.
- No interim adjustments versus constant adjustments.
- Not being able to stop halfway versus always being able to stop.
- Value only comes when everything is finished versus the most valuable thing comes first.
- Accumulating risks versus highlighting risks.

In short, working in long cycles versus working in short cycles. Agile working really is fundamentally different. In a dynamic and complex world where a great deal is changing and under discussion, it is smarter to work in short cycles. That works much better here. Agile working means swimming like a dolphin: always resurfacing and adjusting on the basis of concrete results and new insights.

3 When to Be Agile and When Not?

Agile is not a silver bullet. It is not the solution to all problems. Agile is suited to situations where there is a lot of uncertainty. Where a lot is still changing and a lot is still being discovered. These are also what we call *complex* situations. You know beforehand that many things will still change and afterwards you always know how it should have been done. It's about the degree of (un)certainty in this choice of whether agile will help or not. Agile helps in particular when work is complex and cannot be planned in advance. An alternative to agile is lean. Lean is especially helpful when the work is clearer. It may be complicated, but through repetition it is possible to master the work and make it plannable.

The best way to explain when agile makes sense and when it doesn't is to use the model created by Ralph Stacey (Fig. 2). This model describes the different situations that can arise when both the certainty about what is needed and when it can be achieved decreases. Whether or not agile working is a good choice is strongly dependent on this degree of (un)certainty:

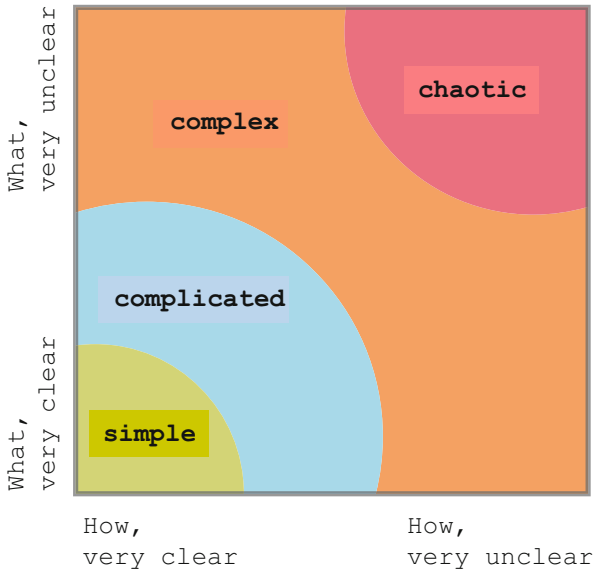


Fig. 2 The Ralph Stacey model

- *Simple* situations: Here it is clear what is needed and how it can be achieved. Simple situations include baking a cake, riding a bicycle or learning to swim, for example. There is a very clear relationship between the what and the how. If you follow a fixed number of steps, you will get the result you want. This is a known method, so there are best practices you can learn from an instructor. And then it works—all the time in fact. If you don't yet know how to deal with simple situations, find a trainer or instructor to teach you.
- *Complicated* situations: These arise when the uncertainty around the what and how increases. It is fairly clear what is needed and how this can be achieved, but rock-hard guarantees are no longer possible. It is therefore worthwhile making a detailed analysis in advance of what exactly is needed. The more precisely something is thought out and specified, the greater the chance of success. Take choosing a technical platform, for example, or making a medical diagnosis or repairing a large machine. All these tasks may seem very difficult beforehand, but if well-trained people carry out a thorough analysis first, they will almost always succeed in achieving a good result. The success rate of the how can be increased by additional attention, training, the use of expertise, automation and/or standardisation. Complicated work is usually repetitive. You do it more often and get better at it. Optimisation with lean, therefore, is perfect in complicated environments. Experts and consultants who carry out analysis and propose a specific solution are therefore suited to complicated situations. Complicated work can be planned in advance, but needs expertise, analysis and preparation.

- *Complex* situations are those in which the what and how become a little more uncertain. The characteristic of a complex situation is that it always turns out differently to what you expected. There is more uncertainty than certainty beforehand. There are too many variables involved that are interdependent. Think, for example, of a large project involving many people and parties, the creation of new IT systems or the merger of two companies. You have an idea of what you want to achieve and how it could work, but things always go differently to how you thought they would. However, in complex situations you can always give a good explanation afterwards of why they went this way. And in retrospect you always know, with your current knowledge, how you should have tackled things. In a complex situation it is therefore best to discover in small steps exactly what is needed and how you can achieve this: experimenting, discovering, learning and making adjustments based on intermediate results. Complex situations are therefore extremely suitable for agile. Daring to discover and getting a coach to help with this are suited to complex situations. Complex cannot be planned in advance, but can be explained afterwards.
- *Chaotic* situations: These are situations in which the what and the how are totally uncertain. Think, for example, of major accidents, Brexits or war situations. Such situations are unpredictable and can only be explained to a limited extent afterwards. Terms such as bad luck and good luck then play an important role. An agile way of working might help, but in a chaotic situation it is mainly a matter of action. Doing something. No matter what it is. You want to get out of the chaos, so you take action in as coordinated a way as possible to reach a different state as quickly as possible. Leaders therefor play a crucial role in chaotic situations.

The answer to the question as to when to be agile and when not depends purely on the situation. Is it complicated or complex? Complex situations are not repeatable. Things always turn out differently. Then agile comes into its own. Agile helps you discover a route when you do things for the first time. Agile is applied in situations that are not repetitive, where only afterwards do you know how it should have been done and what is actually needed. Take small steps and thus make the learning process short cycle and repetitive.

If it is complicated and therefore repetitive, lean is more useful at first. Lean helps to optimise things you do more often and to learn from them. Lean is applied in situations that are repetitive in themselves—think of operational processes or production lines, especially in the manufacturing or service industries. The goal of lean and agile is basically the same: to be successful and to improve on the basis of experience.

However, our society is changing in such a way that more and more complex situations are arising. Everything is accelerating and also becoming increasingly digital. A lot of simple and complicated work is disappearing because it is being automated. And automating something is again complex. As a result, more and more environments are becoming complex and there is also increasingly complex work. This explains why agile is being used more often and more widely. Agile is capable of dealing with complexity, unpredictability and change.

4 Agile Transformations

Because of the growing dynamism in the market and the need for far-reaching agility, many organisations are rigorously transforming themselves into agile organisations. They are introducing agile, top-down and organisation-wide, as the standard working method. Such transformations are usually introduced on a large scale with very large budgets and a great deal of management attention. Besides a new way of working, also being introduced are reorganisations, new career paths, social plans, retraining programmes, etc. These are complex projects that take many months or even years to complete. They are also extremely radical and risky.

Most large organisations have a great deal of experience with radical change. It is carried out in a planned and controlled manner using best-practices, project plans, experienced programme managers, complicated presentations, Gantt charts, and much more. With an agile transformation, such a systematic approach doesn't work. This is because there is so much dynamism and uncertainty that a traditional plan very quickly becomes obsolete and needs to be adjusted very frequently and rigorously. In practice, with agile transformation such a plan is only useful in order to involve all the parties beforehand and to give them the feeling that their interests are being looked after. But you're quite sure that the transformation will not go according to plan.

What does work is: carrying out the agile transformation in small steps. This way, both controlled and agile changes are made. This means that an outline plan is drawn up, which is divided into small steps, with only the short term being worked out in detail. Each step produces concrete intermediate results, on the basis of which the plan can be adjusted again and again. This means that adjustments are made on the basis of concrete results and experiences.

However, this requires a great deal of knowledge and experience with agile on the part of those who lead the transformation. Line and programme managers generally have insufficient resources and are therefore unsuited to carrying out the transformation themselves. VersionOne's annual study (*The 11th State of Agile Report*, 2017) into the global application of agile in 2017 revealed the top 3 bottlenecks when it comes to agile transformation:

1. The current culture (63%)
2. A lack of experience with agile (47%)
3. The current management (45%)

Points 1 and 2 are also the responsibility of the management and this research shows that the management is mainly the biggest bottleneck when it comes to agile transformation.

Large-scale, top-down agile transformations require an approach that is itself agile. This makes it possible to achieve results quickly and to adjust them on the basis of concrete results and changing needs. A plan can be used to do this, provided that it is set up iteratively. Such a step-by-step plan describes a rhythm on the way to the goal, with interim adjustments based on what does and does not work.

And that is actually the essence of successful agile transformation: learning by doing. After all, you learn more from doing things than you do from thinking about them.

5 Agile Transformation in Eight Steps

Although a repeatable recipe to agile transformation does not work because of the high degree of dynamism and changeability in this type of process, it is possible to carry out a transformation in a rhythmic and step-by-step manner.

The following step-by-step approach has already proven itself in practice many times.

- *Step 1—Perform an initial assessment.* Each agile transformation is unique. It is therefore advisable to start with an assessment that maps out the current status, where the introduction of agile is most opportune and which specific obstacles are present. The result is a zero-measurement with insight into the current state of affairs and clarity about where to start.
- *Step 2—Formulate the why and the urgency.* Successfully implementing an agile transformation is only possible if it makes sense and there is sufficient urgency. It is therefore necessary to specifically articulate the reasons for the transformation, so that all those involved gain a clear picture of it. Specify the target quantitatively, so that actual progress can be measured during the transformation.
- *Step 3—Work out a blueprint.* Transformations need direction. So make the ideal final situation (or blueprint) explicit. In order to achieve this situation, it is often necessary to divide the organisation into market- and customer-oriented teams, which work completely autonomously and independently.
- *Step 4—Determine the change strategy.* Will the transformation be carried out in phases, via an oil slick or perhaps with a big bang? All three (and hybrid) variants are possible. It is essential that a fixed change strategy is used. The one that fits best always depends on the environment and the people in that environment. It is important that there is a strategy and that it is consciously chosen.
- *Step 5—Create a transformation roadmap.* This determines the order in which the changes will be implemented. Usually, such a roadmap is worked out on a large board or brown paper on which the various transformation themes and actions are set out over time. For example, with columns such as this sprint, next month, this quarter, next quarter, rest of the year. This is the plan to deviate from and that will be revised and adapted continuously.
- *Step 6—Implement the roadmap iteratively in sprints.* Set up a tight rhythm for the transformation team. Experience has shown that 1-week sprints are very effective, because they result in fast and agile working. In addition, they help keep actions small and result oriented.

- *Step 7—Measure and revise the roadmap.* It is crucial to measure progress explicitly, especially the extent to which the actual goal is achieved. In addition, the roadmap will need to be updated regularly. This is done in detail on a weekly basis (step 6) during the refinement and planning, but it is also important to adjust the roadmap more intensively on a quarterly basis and generally (step 7).
- *Step 8—Integrate through governance and culture.* Changes are immediately anchored in the normal way of working. There are two ways of doing this. Firstly, by making them part of the governance—for example through practices, KPIs or procedures. It is much better to anchor changes through culture. This is because they are then directly integrated into the actual behaviour of teams and people.

6 Pitfalls of Agile Transformations

Becoming truly agile involves much more than learning a new trick. Agile working requires the transformation of firmly anchored structures, processes and systems. However, the existing culture, the underlying views, principles, norms and values also shift. This complicates matters, and even with a quick start it can take several years before an agile transformation is fully implemented. Because agile working is much broader than initially imagined, its actual impact is always underestimated. As a result, many organisations repeatedly make the same mistakes when seeking to become agile. The seven most common pitfalls are:

- *Pitfall 1—No focus on interim results.* An agile transformation requires agility. That is why it's important that whatever is changed really is *changed*. This requires a continuous focus on the implementation of large ideas through small, noticeable steps during the planning and implementation of the changes. In fact, this is the basis of any form of agile: making things small, following things through and learning by doing. Agile transformations themselves are no exception in this respect. However, in practice we regularly see a detailed schedule for an agile transformation with interim milestones and a fixed end date. The most important condition for successful agile transformations is nevertheless agile execution. Step by step, implementing the most valuable change first and making adjustments based on learning experience.
- *Pitfall 2—The 'why' is not measured.* Agile is not an end in itself. Introducing it serves a higher purpose. Many organisations do not make that goal explicit. This creates confusion about the motives, and everyone creates their own interpretation. Subsequently, the extent to which the transformation has the desired effect is often not measured. Without measurable progress, the usefulness of the investments will sooner or later be called into question. Making the goal explicit and measuring whether it is being achieved is therefore very important.
- *Pitfall 3—Management support is at a too low level in the organisation.* Transformations are often made by one specific manager or director. However, practice shows that the desired changes always have an impact on adjacent

departments or other companies in the same ecosystem. This is often forgotten, causing senior management to make the wrong interventions at the wrong time. This in turn is a hindrance and counterproductive. It is therefore necessary to have active support from at least two management layers higher than the place where agile is implemented. But there comes a time when it is the CEO him/herself who will lead the transformation. Ultimately, an agile transformation is an integral transformation of the entire organisation. It can't really succeed without the full support and involvement of the highest executive.

- *Pitfall 4—The impact is heavily underestimated.* Transformations to agile are far-reaching and usually require adjustments far beyond the area in which they begin. Agile teams are set up, for example, and the individual assessment of employees is quickly called into question—after all, it's all about the team's result, right? Before you know it, HR systems and assessment processes are transformed. It is therefore worthwhile to learn from the experiences of other organisations through company visits, for example. In this way, it is possible to anticipate what is yet to come. It is also advisable to include, involve and inform top management. Ultimately, agile will affect the entire organisation. And not only in terms of working methods, but also in terms of processes and structures and even culture.
- *Pitfall 5—Fear of failure rules, and there are too few experiments.* Agile working requires short cycles and is therefore able to handle the unknown well. This does require an organisation to learn how to work with uncertainty, mistakes and experiments. A fear culture makes it extremely difficult to implement agile. As long as people are afraid to make mistakes, experimenting is extremely difficult. But it is precisely experimentation that is needed to be able to work in an agile manner. After all, the faster one learns, the more agile one becomes. A focus on learning experiences and the future in the case of mistakes is then important. As long as there is a fear of failure or in the case of failure, the culprit is sought in the past, an agile transformation will prove difficult. This is because agile is all about learning by doing, and successful learning is impossible without mistakes. Fear of error is perhaps the greatest assassin in any agile transformation.
- *Pitfall 6—The importance of a new rhythm is not understood.* Rhythm is essential for agile working. Many organisations have a hard time with this and add agile meetings as an extra, whereas they should be the basis of everything. The organisation then remains very ad hoc and there is less time for the real work. By establishing a rhythm of recurring meetings, adjustment is provided for. If something unexpected occurs, escalation is no longer necessary, and questions and problems can be dealt with in existing rhythms. This provides security and structure. It does require a new rhythm to schedules and the structural cancellation of the usual meetings as they were held.
- *Pitfall 7—Attention is paid solely to the process.* With a focus on the mechanical process, too much attention is paid to the procedural side of agile and not enough to the side that has real impact. Agile working requires much more than a set of meetings and working arrangements. It requires a different approach to organisational issues. Agile transformations require a change of culture, beliefs

and prevailing values and norms. Make sure, therefore, that the transformation to agile is much broader than just the process side.

7 Conclusion

Agile goes beyond a wish for novelty. Agile working is often started because the market and the environment require more speed and agility than can currently be delivered. Something therefore has to change. Generally, a number of pilots or initial projects are cautiously started. Once confidence has been established, the strategic choice is quickly made to switch completely to agile. This doesn't just happen in smaller companies. Increasingly, large organisations are also switching to agile as standard. A number of multinationals, banks, manufacturing companies and telecom giants are currently carrying out transformations in which agile working will become an important basis for the future.

What does work well in practice is to structure the transition to agile as a separate change process. The first step in such a transformation consists of strategy and planning. Questions are answered such as: Why do we want/need to change, what will the organisation look like in the future, what are good objectives, what problems have we solved? And naturally also: What is the business case for this change, and what does the planning look like? Such questions sometimes do not seem very 'agile' at first glance, but practice has shown that an agile transformation begins by first properly adapting to the current situation.

Implementing an agile transformation is not an easy task. What usually starts with a single agile team soon results in all fixed values being called into question. Despite the good intentions, organisations regularly find themselves mired in an agile transformation. At the same time they often have no choice but to attempt it, since the market demands far-reaching agility and speed. Doing nothing is therefore not an option. Lessons learned and mistakes from the past can help make agile transformations run better and faster.

An agile transformation is best carried out via a step-by-step rhythm of iterative changes, i.e. agile is best introduced agilely!

Next-Generation Software Testers: Broaden or Specialize!

Erik van Veenendaal

Abstract Software has become immensely important for today's society. Despite many quality initiatives, the IT-industry is still far from being able to deliver zero-defect software. In recent years the way software is being developed has changed dramatically in most world regions. In addition to the rapid and dynamic changes currently in the software development arena, there is an increased growth in innovation, new technologies, and expansion of IT throughout most industries. There has been a large shift towards adopting an Agile and/or DevOps way of working. Agile typically provides benefits such as the ability to better manage changing priorities, improved project status visibility, increased team productivity and better delivery predictability. However, many organization are struggling with Agile and scaling Agile, and it has become apparent that moving towards Agile does not automatically also guarantee improved software quality. Testing, although in Agile organized differently compared to traditional organizations, is still and will remain an important part of software development. This is not only due to the importance of software in today's society, but also due to the many (technical) challenges that IT project are facing, e.g., increasing complexity, new technologies, systems-of-systems, variety of devices and OS's and security vulnerabilities. What does all of this mean for the software tester, and to the knowledge and skill set that is expected of a tester? This chapter will look in detail at the knowledge and skill set required for a tester to add value and survive in the rapidly changing IT world. Two options will be provided to the tester:

- Broaden your skill set and become a so-called T-shaped tester.
- Choose a test specialization, but choose this with great care and a vision towards the future.

Keywords Software testing · Software quality · Software tester · Software testing skills

E. van Veenendaal
TMMi Foundation, Dublin, Ireland

S. Goericke (ed.), *The Future of Software Quality Assurance*,
https://doi.org/10.1007/978-3-030-29509-7_18

1 The Future of Testing

Before looking at answers and solutions regarding the knowledge and skill set required for a tester, let's first briefly look at where we are today and what the future of the testing profession looks like (if at all possible). Most people perceive Agile as something that is the common way of working around the globe. However, there are also countries were the number of people or organizations actually practicing Agile are still a minority. Perhaps these countries are just "behind" and they will be Agile a few years from today, but it can also be the case that Agile does not always have a perfect fit within every culture.

There are keynotes at international testing conferences claiming that testers will soon disappear. According to them, there will be no more, or at least very few, dedicated testers in the near future. Interestingly, numbers from survey reports, e.g. the World Quality Report, show exactly the opposite. Also test certification schemes such as ISTQB® and TMMi® are still showing a strong growth and uptake. Surprisingly contradicting talks, signals and facts. Which of them are true, and what is the future for the software testing profession?

In most western societies Agile seems to be a good fit. Generally speaking, there are many people who tend to be communicative, liberal, open minded, love working in a team and less focused on formalities. Mind you, not all parts of the world and cultures are like this. Even within Europe there are huge differences, sometime even between neighbouring countries. Agile has shown to deliver many advantages over the years, but also in a traditional environment delivering a quality product is certainly not impossible.

With all of these opposite and different trends it is quite difficult to accurately predict the future of the testing profession. However, I strongly believe it is safe to state that with the current state of the practice in terms of software quality being delivered and its criticality, the need for testing as a discipline and as a profession will remain (at least for short to medium term). Looking at testing today and tomorrow, there is a firm tendency towards two main options for the tester:

- Become a so-called *T-shaped person* (tester), by changing your attitude and by *broadening* your knowledge and skills. Knowledge and skills will be a challenge in the near future for many testers. It is just not good enough anymore to understand testing and hold an ISTQB certificate. Testers will most often not work in their safe independent test team environment anymore. They will work more closely together with business representatives and developers helping each other when needed and as a team trying to build a quality product. Besides strong soft skills, it is also expected from testers to have knowledge of the business domain, requirements engineering, scripting, etc. Become a "tester plus", someone who can test, but also organize testing and support others in testing. A tester that can do much more than just test.
- Become a *test specialist*. As products are becoming more and more complex, and are integrated in an almost open environment, many so-called non-functional testing issues have become extremely challenging. At the same time the business,

users and customers do not want to compromise on quality. To be able to still test non-functional characteristics such as security, interoperability, performance and reliability, or other complex aspects, e.g. systems-of-systems, highly specialized testers will be needed. Even more so than today, these specialists will be full-time test professionals with in-depth knowledge and skills in one specific (non-functional) testing area only.

The concept of a T-shaped person is of course popular in the Agile world and refers to the need for cross-skilled developers, business analysists and testers in an Agile team, e.g. a Scrum team. In practice, many talk about being a T-shaped tester, but only a few truly are. Let's try to define the knowledge and skill set required to be a true T-shaped tester, but before we do let's look into more detail on what the concept of T-shaped persons actually means and stands for.

2 The Concept of T-Shape

The concept of T-shaped skills, or T-shaped persons, is a metaphor originally used in job recruitment to describe the abilities of persons in the workforce. The vertical bar on the T represents the depth of related skills and expertise in a single field, whereas the horizontal bar is the ability to collaborate across disciplines with experts in other areas and to apply knowledge in areas of expertise other than one's own (see Fig. 1). More in detail the horizontal stroke is composed of two things. First, empathy. It's important because it allows people to look at a problem from another perspective—to stand in somebody else's shoes. Second, they tend to get enthusiastic about other people's disciplines, to the point that they may actually start to practice them. T-shaped people have both depth and breadth in their skills.

To better understand what a T-shaped person is, it is perhaps easier to first understand what the converse, a so-called I-shaped person, is. An I-shaped person is one who is a functional expert—their functional expertise being represented by the

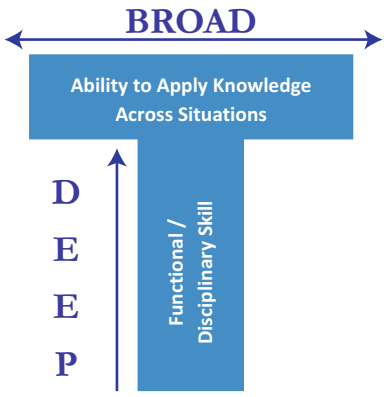


Fig. 1 T-shaped person

vertical stroke in the letter I. There is of course nothing wrong in being an I-shaped person—a functional expert. However, let's imagine a number of functional experts trying to work together on a new mobile app. An app developer, an SEO expert, an analytics expert, a content developer, and an art director have a kick-off meeting to decide on a strategy for the new mobile app.

The SEO expert insists that you should build the app around a keyword map to make sure that the site's structure mirrors an emphasis on keywords. The app developer insists that the mobile app be as easy to code as possible. The analytics expert says that the new design has to be based on what the app analytics show about usage of the current app. The content developer insists it's all about developing interesting, engaging navigable content. And finally, the art director is insisting that app composition and brand beauty is the number one goal. Which one of these I-shaped people is right? How do we manage all these different opinions and make decisions? No matter how good the I-shaped functional experts are at their individual functions, what they lack is not only an appreciation of their co-workers' areas of expertise, but also the training to actually find solutions at the intersection of their respective functional areas.

Let's now compare the I-shaped persons to those being T-shaped. A T-shaped person is typically multi-function aware, collaborative and seeking to learn more about how their function impacts others and the end product. T-shaped people are far more flexible, and more able to easily catch on to new trends, but are of course not as substantial in each adjacent discipline as in their primary skill. Contrary to the I-shaped person, the T-shaped specialist tend to get the general picture rather than immerse themselves in details, unless it's really needed.

In addition to I- and T-shaped concepts, there are descriptive variations that have emerged recently, the most common of which are (see Fig. 2):

- π-shaped—two legs of key skills connected with the dash of general knowledge
- M-shaped—three (or more) key, deep skills

Although the concept of going beyond T-shaped, such as π-shaped or even M-shaped is certainly an interesting one for some disciplines, it probably is not the way to go for testers. As we have learned over the years a certain degree of

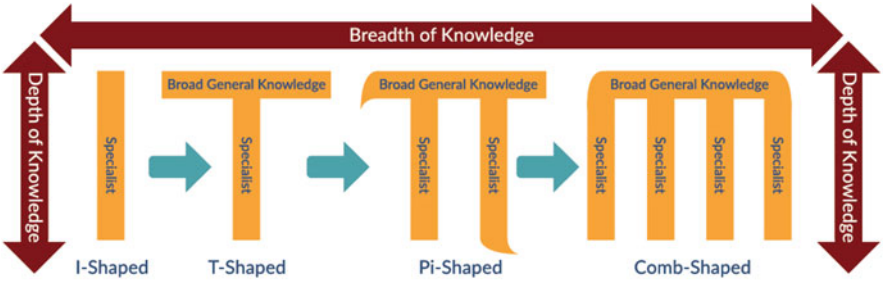


Fig. 2 Variations to the T-shaped concept

independence most often makes the tester more effective at finding defects due to differences between the author's and the tester's cognitive biases (critical distance). Having multiple specialties by being a π-shaped or even M-shaped tester, would typically make it much harder to keep the independent perspective. In these cases, you would as an expert be involved in tasks that you at the same time as a tester should evaluate. In Agile, preserving independence is already often more difficult when a tester is embedded in a team. At the same time, also being a π-shaped person possessing (and performing) another specialty beyond testing would probably make the required level of independence almost disappear altogether.

T-shaped people and the teams they work in can achieve results far better than teams that consist of only I-shaped people. But the development of T-shaped people is a serious, long-term undertaking and most often largely underestimated. It requires people with the right attitude and self-determination to start, but then it requires effort to continue to provide them with the training and resources they need and the type of safe collaborative environment that allows for T-shaped person and teams to perform at their best.

3 The T-Shaped Tester

To drive a career in software testing, what are the most valuable knowledge and skills to acquire. As already stated, one way to develop a testing career is to specialize by going deeper in a single specific non-functional or niche. These kinds of specialists are regarded in theory as being I-shaped, which means that their skills are seen as being very narrow but extremely deep. However, in a fast-paced world, this strategy has evident risks, such as if the area of specialization becomes outdated, unpopular or, as we will see later, when a specialized area changes to one that becomes common to all.

While in previous decades there was a demand for I-shaped testers, there is now a growing opportunity for T-shaped persons because those who have deep skill in one discipline and in addition general knowledge across disciplines will much easier be able to work in a changing environment. In the Agile world, the T-shaped tester is a team member whose key expertise is testing but who can also provide support in other activities, for example, those that lie in the fields of programming or business analysis (requirements engineering). So, in relation to T-shaped, we should look for the skills that can potentially boost the testers' profile. For a professional software tester, good options would be:

- *Testing*: have a deep and broad knowledge across the testing domain
- Other *development specialties*: business analysis, programming, technical writing, etc.
- *Domain knowledge*: Medicine, Insurance, Banking, IoT, etc.
- *Soft skills*: they have a positive impact on personal effectiveness, leadership and collaboration with others

Discussing the skill sets of T-shaped testers, we should also be aware of the proportions between 'horizontal' and 'vertical' aspects in the skill set. Depending on the environment, the need in each family of skills will differ. Those who have very deep and narrow expertise in the field can become over-skilled, as employers don't tend to pay for skills they don't need. Those who have broader skills can feel the lack of expertise in their key discipline at some point and will need to catch up on it in the short term if required. Hereafter the four knowledge and skill options identified for the T-shaped are elaborated upon with examples.

3.1 Deepening: Testing Knowledge and Skills

Today's tester needs to have a full toolbox with varying test methods and techniques. Working in a team, depending on the context and charter, the most appropriate methods and techniques shall be selected from the toolbox. Trying to define the toolbox for the tester, and thus the testing knowledge and skills required, the ISTQB product portfolio can be used as a reference model. Although there is much criticism on ISTQB in some testing communities, from a content point-of-view there is without doubt much interesting knowledge and material available across many areas of testing and documented in the various syllabus. ISTQB today is much, much more than the basic ISTQB foundation level syllabus.

If we take the ISTQB product portfolio as a starting point (see www.istqb.org for the latest version of the portfolio), many interesting topics and syllabi are available. Trying to define the required testing knowledge and skills, it is at least strange to see that ISTQB does not consider Agile testing to be a part of the core set of knowledge and skills. It is defined within the portfolio as a separate stream. Also interesting to see is that test automation and mobile application testing are considered as specialist areas within testing. Today, these are almost like standard requirements for a tester. The fact that these were originally defined as specialist areas by ISTQB perhaps shows how quickly the market changes. What is defined as a specialist area today could well be a common requirement for knowledge and skills tomorrow.

The picture in Fig. 3 is by no means intended to be complete or based on some extensive survey. It is intended to show on a high level what is expected from a tester in terms of testing knowledge and skills today and for sure tomorrow.

Having attended an ISTQB Foundation course (and having passed the exam) and then stating you are a professional tester ready for the future is almost like a joke. A 3-day ISTQB Foundation course is based on a well-founded syllabus, but it only teaches the basics and principles of testing and doesn't get the job done. One will also need to be trained in Agile testing, which I believe is core and not optional for any tester. The real meat is in attending more advanced hands-on, practical courses or workshops in which you will learn how to actually apply testing practices in various contexts. These advanced courses should include areas such as test automation and mobile application testing, and of course should be taught from an Agile perspective. Following the T-shaped concept and having a deep

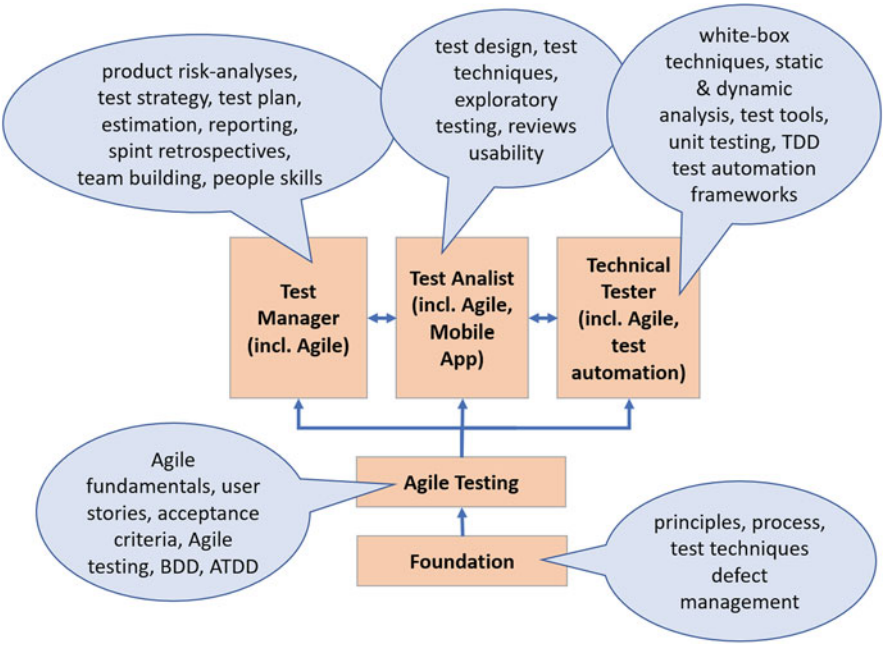


Fig. 3 Tester knowledge and skill set

knowledge in testing, we do not expect testers to choose between follow-up areas such as test manager, test analyst or technical test analyst. We expect testers to cover all three areas and become a true professional. As an example, there are fewer and fewer dedicated test managers or test leads. Many testers today are embedded in an Agile team, as such they perform testing tasks, but also coach and support business analysts doing functional user-story-based testing and developers doing automated unit testing. Being a tester in an Agile team also means you are involved in tasks that were originally in the exclusive domain of the test manager, e.g. product risk sessions, estimations, retrospectives, reporting, etc.

Note there are many other means of acquiring the same testing knowledge and skills; ISTQB is just one option and used here as an example. There are often great tutorials at testing conferences that discuss interesting topics and areas. There are also many highly practical courses available in the market, e.g. on test automation. Of course alternatives exist to doing formal training as well, e.g. mentoring, on-the-job learning, etc.

Not Just the Tester!

There will most likely be professional testers in the future, but even more so testing as an activity will remain to be extremely important and challenging. Not only the tester performs testing activity, also other team members, e.g. developers and business analysts, will perform testing activities especially as a result of Agile

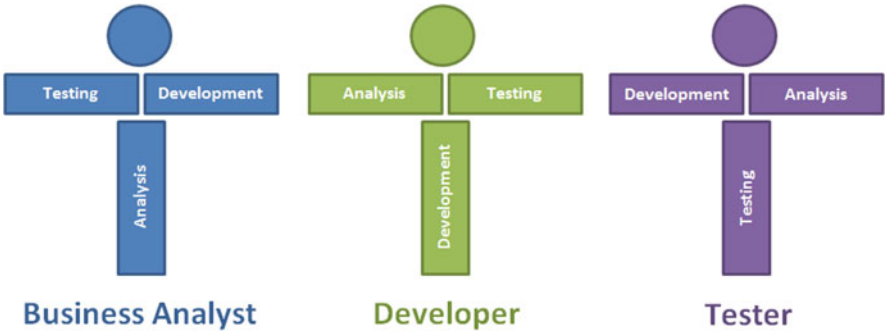


Fig. 4 T-shaped Agile team

transformation. Following the T-shaped concept, the tester is expected to have deep knowledge and skills in testing, but from other team members it is expected that they acquire testing knowledge and skills as part of their horizontal skill set bar (see Fig. 4). We cannot just direct other people to be involved in testing, they need to be trained and coached in testing to be able to perform this task. Remember what Glenford Myers stated in his book *The Art of Software Testing*: "Testing is an extremely complex and intellectually challenging task".

3.2 Broadening: IT Knowledge and Skills

Working in a cross-functional team closely with developers and business analysts implies that at least a tester needs to appreciate and understand what other team members are doing and preferably also be able to support them in their tasks. From a T-shaped tester, it is typically expected they can support a software developer with unit testing and a business analyst when they are defining acceptance criteria for user stories. It is required to understand the life cycle model that is being used and the technical environment in which development takes place. IT knowledge and skills for a tester should cover a wide and varying range of knowledge skills of which some essential examples are listed and described hereafter.

Requirements Engineering/Business Analysis
Testers are of course one of the main stakeholders. Test planning, product risk analysis, test cases are all based upon requirements. Tester are involved in requirements reviews, and need to understand what level of quality is reasonable to ask for. Often in Agile teams, testers support the identification and specification of requirements (user stories) and their acceptance criteria.

Programming
It isn't that testers need to work like a developer, but it is important to understand the inside of the application so that it becomes easy to comprehend its functioning

and risks areas, and create tests accordingly. Programming knowledge helps in identifying possible errors in the code and work closely with the developer on static analysis and unit testing, possibly using Test Driven Development. It is advisable to learn at least two programming languages, e.g. Python, Java or C++. Of course having programming skills also strongly support the ability to perform test automation tasks. With the increasing complexities and integrations in the application, relying on manual testing alone cannot get the job done.

Web and Mobile Technologies
Today most testers must also become familiar with the web and mobile technologies so that they can understand the application, its build and scalability and apply a suitable course of actions for its testing. It is highly important that testers keep an eye open on web and mobile technology advancements since it guides them in comprehending the coding architecture and technical challenges to deliver effective testing solutions.

Software Development Lifecycle
Testers need to learn and understand the software life cycle as it will help them understand the development tasks and plan testing cycles accordingly. Having an in-depth knowledge of software life cycle will also help anticipate challenges in the development process which can guide in taking the right measures beforehand. With Agile and DevOps methodologies being popular, testers of course need to learn and understand these as well, especially the impact this has on how testing is performed.

Project Management
Learning the skills of project management will support the tester in becoming a better test manager. Project management skills also prepare testers to be accountable and answerable for their work to concerned stakeholders and also undertake responsibility and management for specific test activities. This way, project management skills contribute to delivering quality results, improving the entire test process.

Configuration Management
The purpose of configuration management is to establish and maintain the integrity of the component or system, the testware and their relationships to one another. For the tester this an important process to ensure quality and as such it is essential to have some knowledge on configuration management. In addition also, testware needs to be managed via this process. All items of testware should be uniquely identified, version controlled and tracked for changes, related to each other and related to versions of the requirements so that traceability can be maintained.

3.3 Broadening: Domain Knowledge

In this context, domain knowledge is defined as knowledge about the environment in which the target system operates. For a tester it's important to understand the domain in order to be able to communicate with business stakeholders (product

owners), but also to make the right decisions while performing testing activities. Remember, exhaustive testing isn't possible, and testers are making trade-off decisions all the time, and they'd better be the right ones! What features are most important to test, which configurations occur most often, etc.? Hence for a tester possessing domain knowledge along with the other skills is a big plus to the industry. In the context of being a T-shaped tester, there are also benefits outside of testing. A tester with domain knowledge can probably partly relieve a business analyst, or assist other team members by bringing the necessary domain perspective.

Some examples where a tester will benefit from having domain knowledge are discussed next.

Understanding Risks and Better Test Cases
Unless you are aware of the domain, you can't identify and analyse the product risks, and write and execute test cases to effectively simulate the end user. It's not just about using at a particular activity, it is required throughout all testing activities.

Understanding Impact
When an issue arises, a tester understanding purpose of the functionality of the system will much better be able to analyse the impact of the issue. For example, when a defect is found in the payment process of an online pizza order application, a domain-based testers will have a clear idea about the process steps impacted needed for a successful transaction. This will assist the tester in doing better confirmation and regression testing when the defect is fixed.

More Important Defects
Domain knowledge testers are high in demand due to their ability to understand the application beyond just finding defects, e.g. during exploratory testing. They typically find more important defects.

Prioritize Defects
Since the tester understands the domain, the tester will have a clear idea of how to best prioritize the outstanding set of defect fixes.

More Effective Reviewing
A tester with domain knowledge can be more productive at the start of the project or iteration. Good knowledge of the functional flow of the business processes and business rules will help better understanding the requirements and as a consequence be able to perform reviews more effectively.

Being in an IT-dominated world, the value of a tester with domain knowledge is incredible as it is undoubtedly a critical success factor for testers. While testing any application, it is important to be able to think from an end user's perspective since they are the ones who are going to use the product. Domain knowledge usually must be learned from end users (as domain specialists/experts) and may, amongst others, include user profiles, workflows, business processes, business policies and configurations. Without going into detail on how to acquire business and domain knowledge, there is of course much more than just attending a training, also consider

apprenticing, observing users/customers actually using the application, visiting online forums and becoming part of communities.

Note, there seems to be a tendency to prefer technical testers in the Agile community, but as we have learned in this paragraph, there is a strong need and benefit to having testers with domain knowledge and sometimes even a domain background, e.g. end users that have become testers! Of course there never is a right answer in these kind of situations, but it is something that needs to be considered and balanced when assigning testers to a team and defining a required knowledge and skill set for the T-shaped tester.

3.4 Broadening: Soft Skills

Any software tester should also possess so-called soft skills (also known as people skills). Soft skills are important as they are used to approach work. Testers have an instinct and understanding for where and how software might fail, and how to find defects. A tester also should have the soft skills to influence and communicate in a manner that they become vital to the project. Testing requires a toolbox full of soft skills, including communication, time management, analytical skills, eagerness to learn and critical thinking, but also relatively standard people skills such as reading, reporting and presentation skills. Some of these skills allow a tester to be better in finding defects, but most of them relate to being better at communicating a, most often difficult, message. Having these soft skills is like a prerequisite to having the right attitude for being a T-shaped, open-minded tester within an Agile team, e.g. have empathy towards other disciplines, knowledge sharing and being a team player. Being a T-shaped tester and being part of a team means the soft skills that the tester possesses can also be used to support other team members at their tasks.

Hereafter, some of the important soft skills which a tester should possess in order to excel in his field are briefly described.

Communication
Since a tester has to deal with so many different team members, it is very important to have a proper communication channel with them. Whether it is a defect to report, explanation or clarification on an ambiguous requirement, a tester has to communicate with respective business analysts, developers and sometimes end users. When you have good communication skills, you eliminate ambiguity and misunderstanding while talking to the different team members. Also, most of the issues which arise due to a communication gap would be at par. For example, when you find a crucial defect, it is very important to explain it in a polite way so that the developer doesn't feel like you are blaming him for the root cause.

Time Management
Time management is the most important skill for a tester, even in Agile the last days on an iteration are usually packed with "stressful" last minute testing. When you know how to use your time properly and how to prioritize tasks according to the end

date, you will end up meeting the end dates more easily and encounter less work pressure.

Analytical and Detail Oriented
This is one of the skills which can help you find more and the most important defects. When you understand and are able to analyse requirements not only gaps in the requirements can be found and fixed but also you get to know the whole functionality, flow and the expected outcome of the product. Sometimes details could be missed that later result in a major issue which are much more difficult to rework. To mitigate such situations it is also very important to have a 'detail-oriented attitude'.

Eager to Learn
If you're eager to learn, and willing to take on activities outside your comfort zone needed to help the team deliver a good product, you have a bright future as a tester. It is crucial for the software tester to be aware of the latest tools and technologies and to keep learning without fail. When you have knowledge about the latest tools and technologies, you make better decisions.

Critical Thinking
Critical thinking is the kind of thinking that specifically looks for problems and mistakes. It is the ability to reason by carefully analysing something in order to determine its validity or accuracy. Critical thinking is about being an active learner rather than a passive recipient of information. It is possibly the most important type of thinking in the context of testing. As testers, we should always question ideas and assumptions rather than accept them at face value.

Knowledge Sharing
A tester should have an attitude of helping and coaching his Agile team members by sharing his knowledge. Not only will this avoid gaps and confusion, allow them to assist in and do better testing themselves but also motivate other team members to share their expertise and knowledge.

Team Player
Being part of an Agile team, you always need a team effort to achieve something. A tester motivates the team towards better testing and good levels of product quality. To achieve quality software, it is important for a software tester to be a good team player.

4 The Test Specialist

As already stated in the introduction paragraph, there is an alternative to the career path of becoming a T-shaped tester: become a test specialist. A specialist is defined as a person who concentrates primarily on a particular subject or activity; a person that is highly skilled in a specific and restricted field. Following the concept of

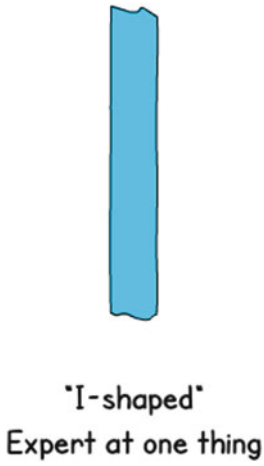


Fig. 5 Test specialist

"shaped" persons, a test specialist is an I-shaped person (see Fig. 5). A tester with a deep (vertical) expertise in one testing area and less knowledge and skills in other testing areas, let it be outside testing. Their expertise in that one testing area is much, much deeper than the expertise possessed by a T-shaped tester for the same testing area.

As products are becoming more and more complex, and are integrated in an almost open environment, many so-called non-functional testing issues have become extremely challenging. You cannot just do this on the side, this requires much specialized knowledge and skills, training and dedication. To some testers it may not be their piece of cake, or may even be too (technically) difficult to master. As a result, to still be able to test an IT-industry non-functional characteristics such as security, interoperability, performance and reliability, or other complex aspects like systems-of-systems, highly specialized testers will be needed. These specialists are typically full-time test professionals with in-depth knowledge and skills in one specific (non-functional) testing area only.

Also from a customer point of view that comes to us for a solution, the customer may sometimes come to the door with a problem that is to be addressed by a single specialist only. It doesn't require a team-based solution. It is with these kinds of problems or questions that the I-shaped specialist clearly has much added value.

So what are typical test specialists areas? Again, let's look at the ISTQB product portfolio as a reference framework to identify some testing topics that are considered to be areas where we would benefit from having test specialists. ISTQB clearly points us in the direction of non-functionals (see Fig. 6), e.g. security test specialist, usability test specialist or performance test specialist. These are valuable test specialists in the context as defined above. ISTQB also mentions amongst others automotive software tester, gambling industry tester and model-based tester. Personally I'm not sure these are test specialist areas that contain and require enough specialist knowledge and skills to be able to add value and survive as an I-shaped

Fig. 6 Examples non-functional test types

tester. Probably these areas are more specific to a certain domain and belong in the knowledge and skill set of a T-shaped tester in that domain.

Finally, it is also interesting to see that ISTQB also considers test automation and mobile application testing to be specialist areas within testing. Today, these are almost like standard requirements for a tester. The fact that these were originally defined as specialist areas by ISTQB perhaps shows how quickly the market changes. What is defined as a specialist area today could well be a common requirement for knowledge and skills tomorrow. This also points out the danger of being an I-shaped test specialist. Today, there is a huge demand for security test specialists and perhaps slightly less for performance test specialists. However, this may rapidly change or the specialism gradually moves to become a generalized knowledge and skill area that is required for T-shaped testers. Usability testing is probably somewhere in that transition.

To summarize, test specialists are needed and have much added value. It certainly is an alternative from being a T-shaped tester. Be careful which specialism to choose and keep an eye out for what is happening in the market today and the near future. Sometimes is it possible to hop from one test specialism to a new one when the current one becomes obsolete. At the same time, is doesn't hurt to also have some knowledge and skills that typically belong to the T-shaped tester as a fallback scenario. The latter will also be beneficial when working less stand-alone and being

part of a team. It will again also help to appreciate what others are doing and assist in looking at things from a different perspective.

5 Conclusions

With the current state of practice of the IT industry, we are far from achieving zero defects (if at all in the future). Software testing is and will remain an indispensable part of software development. The required knowledge and skills of the test professional of the (near) future are discussed. In detail, the T-shaped tester was presented and explained. In practice many talk about being a T-shaped tester, but I believe in reality we are far from being there. The need for I-shaped testers was also briefly presented. There are basically two options to choose from:

- Broaden your knowledge and skills and become a true test professional (T-shaped tester)
- Deepen your knowledge and skills in a specific testing area and become a test specialist (I-shaped)

For testers driving their career, it is extremely important to define individual direction of growth and development. Not forgetting the T-shaped concept, base your choice on your own strengths and passion, and take into account your work environment (life cycle, domain), trends in the industry and the current (and future) demands of the job market. In addition, reviewing your career plan on a regular basis will help you stay on top, and get the best value out of the work you do.

Acknowledgement An explicit thank-you is to Bob van de Burgt, who reviewed this book contribution in detail and provided valuable feedback.

Security: It's Everyone's Business!

Keith Yorkston

Abstract Security isn't only a bit of software that can be bought, installed and forgotten with the occasional upgrade thrown in. Security isn't only that set of password rules we are supposed to follow. Security isn't only that locked filing cabinet, or a guard and a scan card reader at the front door. It includes all those things, and many, many more. We all need to think about security differently. Every organisation has thousands of vulnerabilities—weaknesses that could be exploited by a malicious attacker. And, as a malicious attacker, I only need to find one vulnerability to exploit. It could be a helpful staff member holding the door open for a "fellow smoker", or a person in Finance who believed that last phone call asking them to process "that important invoice". It might be an open comms port on the production web server, or the unpatched server in the test environment. Or it could be the report listing last week's customer contacts that is mailed to the sales staff each Monday (including the sales staff who have left the organisation). I mention these because my colleagues and I have used all these techniques (and many more) to test organisations. We are security testers.

Keywords Software security · Software quality · Security testing · Security tester

1 Introduction

Your organisation has been hacked. Think for a minute—who might have instigated this attack? What type of person springs into your mind?

Did you have an image of a darkened room, with a faint green glow showing empty energy drink cans and a young, angry guy furiously pounding a keyboard? Did you imagine a vast room full of people in the strange uniforms of a totalitarian

K. Yorkston
Expleo Group, London, UK

S. Goericke (ed.), *The Future of Software Quality Assurance*,
https://doi.org/10.1007/978-3-030-29509-7_19

regime? Both could be true. Or did you think of that strange phone call that a colleague answered yesterday? Or the last invoice of the thirty that Finance processed in their last payment run? Or the guy who just took your secure waste "for disposal"? Or did you think of a workmate who always attaches a USB drive to machines at work and takes it home with them each afternoon? Or the office worker who printed an extra copy of the confidential report to pop into the outgoing mail?

Security isn't only a bit of software that can be bought, installed and forgotten with the occasional upgrade thrown in. Security isn't only that set of password rules we are supposed to follow. Security isn't only that locked filing cabinet, or a guard and a scan card reader at the front door. It includes all those things, and many, many more. We all need to think about security differently. Every organisation has thousands of vulnerabilities—weaknesses that could be exploited by a malicious attacker. And, as a malicious attacker, I only need to find one vulnerability to exploit. It could be a helpful staff member holding the door open for a "fellow smoker", or a person in Finance who believed that last phone call asking them to process "that important invoice". It might be an open comms port on the production web server, or the unpatched server in the test environment. Or it could be the report listing last week's customer contacts that is mailed to the sales staff each Monday (including the sales staff who have left the organisation). I mention these because my colleagues and I have used all these techniques (and many more) to test organisations. We are security testers.

But wait, you say. Don't testers sit at a desk in an office and write and run tests against software? Yes we do. But we also dress up as delivery drivers or people in the waste disposal industry, or wear suits after making fake company passes. What good is a fake badge? You say it won't open the security gates in reception?

You're right—on its own it won't. It would take about 10 min to create a fake ID card, as they all tend to have a photo, name and company logo on them (check your badge—am I right?) I have visited organisations and walked into reception purely to see the ID card design. Have a slightly confused look, map in hand, "Could you please tell me how to get to [any address nearby]?"

Or just wait for staff filing out at lunchtime. Then, into Microsoft Paint (yes, the big budget hacking tool), print out onto paper, and with some sticky-back plastic over an old card, I now have a freshly made organisation ID card. Of course, it won't pass close scrutiny, but when was the last time anyone checked an ID card? It gets a glance at best. Next trick, how to get through the gate?

Carry something. Literally, a big armful of paper/books/boxes/whatever. As you approach the gate (and the guard casually glances at your freshly made card) you ask, "Could you please open the barrier for me? I'm late for a meeting . . . "

And the organisation has been hacked.

2 Training Security Testers

This example is a security test. Security testing needs to consider what is known as the "iron triangle" of:

- Technology
- Processes
- People

The testing we do is not, and should never be, limited to technology alone. Many people today still think you can "buy security", or that security is limited to a technical solution. Hackers use computers, we will be hacked, therefore hackers attacking the organisation will use computers.

But consider the example above. The only technology involved was a programme first released in 1985. This hack relied on people vulnerability—the guard wanting to be helpful. It showed a weakness in the process—I didn't scan my card (the process) because the guard scanned their own card in trying to be helpful. How could this be stopped? Could the organisation hire meaner guards? Think on this for a second . . .

A number of things could be suggested. The process could be changed, but unless it is enforced, incidents like this could continue to happen. Awareness is key—if the guards know that someone might try this, they can be aware of the situation. We have all heard of the "mystery shopper"—what we need is a "mystery hacker". Or in other words, a security tester. The organisation must train the guards and staff to recognise social engineering—the science of skilfully manoeuvring people to take some desired action. Why would a guard open the gate for me? Because I presented a situation to them that appealed to their good manners. You know, those things your parents told you, "Wash your hands, say please, and *hold the door open for others* . . . " (I was also told to add "Clean up your room!"—thanks for proof-reading, Dad).

But this training should never stop at the staff in reception. Upper management also need to not only sponsor initiatives on security, they themselves need to participate in security training. They are a huge, visible target—it's easier to find details of an organisation's CEO than the name of a person in Accounts Payable. Look at the organisation's annual report, or your country's company register (Companies House in the UK). Then, using sites like Google or LinkedIn, look them up. Even going further, using a data mining tool such as Maltego can uncover a malicious user's treasure trove of publicly available information. Much has been written about spear phishing—targeting an individual for a specific phishing attack. It's vitally important for senior staff to understand the threats posed against the organisation and the possible vulnerabilities that could be targeted. But only not in general terms, the specific attacks focussed on just them.

Business email compromise (BEC) has, for a number of years, relied on a simple fact:

No one questions the Boss!

In 2018, the US FBI estimated global losses between October 2013 and May 2018 from *this attack alone* cost organisations US$12.5 billion.[1] This attack uses spear phishing. Find out details of the senior staff, then send an email to accountspayable@[insert organisation name]. Spoof it to come from that senior staff member, with a message like "An invoice is coming from Acme Corp, can you please process this payment quickly, as it is part of Project Merlin". It's especially useful if the attacker knows a project name within the organisation, but it's also surprising how many "Project Merlins" actually happen. Then, a call comes through to the general organisation phone number—"Can I please speak to Accounts Payable?"

The call is put through, and it's Tom Jefferson[2] from Acme Corp, asking about the invoice. He seems such a nice chap, and is very apologetic about the rush for payment. He's also very helpful, giving Acme Corp's bank details to the Accounts Payable staff. The call adds legitimacy to the spoofed email, and that $12 billion loss just got a bit bigger.

Once again, there are most probably processes that should be followed by staff. The senior staff member's email was spoofed. And, the account staff were convinced to be helpful to solve a problem by the email "from the Boss", and the call they received.

But, says you, don't we know the account the money was paid to? Surely if this information was passed to the Local Constabulary, they could stake out the bank branch, looking for the suspicious individual who, while wearing sunglasses and a false moustache, withdraws the money from the account? Perhaps, but another aspect of the attack is another vulnerable victim. A lonely person who struck up a conversation with a social media connection, and due to [an unexpected tax bill/a sick relative/a windfall from a recently departed relative] they require a bank account in the target's country. And, if the target opens this account on the basis that "Afta this is dun, I cn get my viza to meat my luv", the attacker now has an account that bears a striking resemblance to the Acme account number (so close, in fact, it's the same). Or the attacker might identify a like-minded person in that country who opens the account for a percentage of the money passing through it. And the consequence? Maybe a possibly innocent (or not so innocent) person could be charged with money laundering. And the hunt goes on for the money.

And the most targeted business sector for BEC? Which do you think? Sometimes the target can be unexpected. You must think like an attacker. With the prevalence and reliance on online services today, we can do almost anything online. Including finalising the purchase of property. The real estate sector is a lucrative area targeted by the attackers. Think of all those involved in purchasing a home—solicitors, surveyors, real estate agents, buyers and sellers. How many of these could be vulnerable? If you're buying a home and get a mail for the final purchase from the "real estate" giving the account details for the transfer. Or from the "solicitor"

[1] https://www.ic3.gov/media/2018/180712.aspx

[2] No relation to the third President of the United States.

stating there's a problem which will require a small fee to clear up. Or the real estate receives a mail from a "buyer" to list a property. Were I to turn to the "Dark Side", I would get a greater return attacking the local florist than I would attacking Amazon. The reason? Amazon spend a large amount of money hiring some of the world's experts in security, implementing top-line technology and robust processes to reduce vulnerabilities. The local florist/solicitor/real estate/surveyor cannot.

3 Reasons for (Cyber) Attacks

Why would people conduct these attacks? As mentioned previously, these attacks can obtain a large amount of money quite quickly. Years ago, motivation for the attackers was covered with MICE:

- *Money*—this is very much the motivation in the case of BEC. Money can be a multi-faceted motivator. I once was "called by Microsoft" to have the person on the call tell me about the "viruses" on my machine. After showing a little empathy ("You're job must be very difficult . . . ") and expressing how this call was a waste of time, they opened up to me, telling me they knew they were doing wrong, but it's a job to feed their family (or, were they trying to socially engineer me?). But they also felt safe, in that the chances of being caught were much less than getting fired for not "making enough successful calls". Some people can make a lot of money from malicious attacks,[3] while others do it to "pay the bills".
- *Ideology*—as before, ideology can encompass a vast array of views. This motivation could be a lone person who lost their business "to the bank", a small group supporting animal rights, or a large group with a particular secular belief. And, based on this motivation, the targets will be different. Then, there's the morally ambiguous groups of hackers, whose ideology can shift. And sometimes even this can be muddied—I enjoy when occasionally people attending demonstrations against capitalism/global corporate domination ,etc. wear the white Guy Fawkes mask from the movie *V for Vendetta*. You know, that trade-marked thing owned by Warner Brothers[4] . . .
- *Compromise*—you may have received the latest phishing mail floating around the world, a version of which I received is below:

```
Hello!
I'm a member of an international hacker
group.
As you could probably have guessed, your ac-
```

[3] "Black-hat sextortionists required: Competitive salary and dental plan"—listed on https://www.theregister.co.uk/2019/02/21/black_hats_sextortion_275k_salaries_helpers/

[4] "The irony of the Anonymous mask" listed on https://www.theguardian.com/technology/2011/aug/30/irony-of-anonymous-mask

```
count [email removed] was hacked, because I
sent message you from it.
Now I have access to you accounts!
For example, your password for [email re-
moved] is [password removed]
Within a period from July 7, 2018 to Septem-
ber 23, 2018, you were infected by the virus
we've created, through an adult website
you've visited.
So far, we have access to your messages, so-
cial media accounts, and messengers.
Moreover, we've gotten full damps of these
data.
We are aware of your little and big se-
crets...yeah, you do have them. We saw and
recorded your doings on porn websites. Your
tastes are so weird, you know..
But the key thing is that sometimes we rec-
orded you with your webcam, syncing the re-
cordings with what you watched!
I think you are not interested show this
video to your friends, relatives, and your
intimate one...
Transfer $700 to our Bitcoin wallet:
[Bitcoin wallet removed]
If you don't know about Bitcoin please input
in Google "buy BTC". It's really easy.
I guarantee that after that, we'll erase all
your "data" :D
A timer will start once you read this mes-
sage. You have 48 hours to pay the above-
mentioned amount.
```

I wonder if the author had a competitive salary and dental plan? Compromise allows the attackers to control the situation. I have something you don't want released (your photos from your phone/compromising browsing history/bank account details) and for some consideration (cash/a confidential report from your organisation) the problem *might* go away.

Ego—sometimes, the urge to be "the smartest person in the room" can take over. It could be childhood dreams (such as reading of the exploits of Kevin Mitnick or Kevin Poulsen, both security experts arrested for hacking) and imagining "If only . . . ", or it could be the need to, like many conspiracy theorists, be the only person "who knows the truth".

Some of the tools the attacker will use are fear, greed and even empathy. Chris Hadnagy and Michelle Fincher wrote the book *Phishing Dark Waters* [1] where they took an in-depth study of phishing. An attacker can use fear, such as the example above, or the alert from "your bank" saying your account has been hacked, and you should click the link below to "verify your password". They can use greed—the Spanish prisoner/Nigerian 419 scam—where the victim is asked to "please make a small payment/give me access to your account to release the

millions held offshore, for a substantial fee". And the most insidious is empathy—my grandmother/mother/child/cute puppy is dying, and I need money to save them (photo attached). All these attack the person—and our technology and processes must be in place to combat these.

4 What Security Testers Need to Understand

Let's be clear. We have looked at elements of criminal offenses. All the attacks mentioned above are crimes in the UK according to the Fraud Act and Computer Misuse Act amongst others. These crimes fall into two categories—cyber dependant (hacking an online account, where without IT, the crime wouldn't exist) and cyber-enabled (where the number of victims of a criminal act can be increased—an organisation I worked for used to receive snail-mail letters from "Nigerian Princes").

This is something that is absolutely vital for a security tester to understand. You might be running a legitimate test within your organisation, but unless you have documented permission from a person of relevant authority, you could be charged and even found guilty of an offence. Remember, a malicious user might be an employee of the organisation. If you do not have specific permission, how do we know that your test isn't an actual attempt at malicious harm? Something as benign as clearing your browser cache could be interpreted by authorities in the USA as destroying evidence. Any work conducted as a security tester must be covered by legal indemnity—the basis for which is supplied by the Open Web Application Security Project (OWASP).[5] If you have ever played Monopoly, this is the security tester's "Get Out of Jail Free" card. Literally. It provides evidence that the work security testers are undertaking IS AUTHORISED, and isn't cover for an internal malicious user.

As said, even if you are testing your own organisation's systems, you absolutely need this permission. Never let someone else (e.g. a hypothetical project manager) convince you this is not required. If this isn't in place, you are breaking the law, and could be subject to prosecution.

Another interesting threat that has grown since 2014 is a method called "account stuffing". Let's say a hypothetical user has multiple online accounts. The user cannot remember unique passwords for all the accounts, and uses the same password for a number of accounts. Now, let's say they have the same password for both the local florist AND their Amazon account. So, if I steal the local florist usernames and passwords (remember, their security policy might not be best practice), I could now access many more accounts with that credential set.

What's more disturbing is the marketplace for stolen credentials has been "automated". Stolen credentials are fed into an automated process checking these against sites like Amazon, PayPal or E-bay (amongst others). A successfully stolen and checked account can then be sold to interested parties for between $0.50 USD

[5] https://www.owasp.org/index.php/Authorization_form

and $3.50 USD, with the potential for the purchaser to make up to 20 times the cost price.[6] This is why we are continuously told not to replicate passwords over multiple sites. Account stuffing relies on a people-based weakness, we cannot remember long, complex, unique passwords, so we cheat! And inadvertently, create a vulnerability.

5 About Password Security

Passwords demonstrate the battle between security and usability. A long, complex password might be good for an organisation's security, but if security procedures become onerous in the view of users, they will find a way of subverting or avoiding them. The procedure thus becomes ineffective. Consider the following password rules (which might look familiar to many) and the subsequent passwords:

1. Must be a minimum 8 characters (12345678)
2. Must contain at least 1 upper and 1 lower case character (Qwertyui)
3. Must contain at least 1 number (Qwertyu1)
4. Must contain at least 1 special character (Qwerty1!)
5. must be changed every 30 days (Qwerty2")

These listed passwords would take at most minutes to break for tools like John the Ripper or Hashcat. Various lists exist of the "Top 25" passwords used—all of which vary slightly due to the data on which they draw, but contain many of the same passwords (and yes, "password" is in there!) But they point to a common theme—that is when we think we are being "random", we aren't. There is a reason why "qwertyuiop" isn't a good password—look at the top row of keys on a keyboard. Humans follow patterns, and those patterns can be predicted and replicated.

An interesting point is during the infamous Sony email hack relating to the release of the movie *The Interview.* The then CEO of Sony, Michael Lynton, had a password of *Sonyml3.*[7] Any prizes for guessing what his next password might have been?

We need to forget passwords. Any password of eight characters is broken very quickly, and just because the MINIMUM is eight, it doesn't mean EXACTLY eight. And even if it cannot be broken in seconds, the attacker may not mind. They would have all the time they need to crack the password, as even when an alert goes out from the attacked site, how many people actually change THAT password, let alone all the other accounts using that same one. It should be noted that the encrypted password isn't decrypted, but a known word is encrypted to see if the encrypted result matches any passwords in the stolen set. The longer the password, the exponentially more combinations could be used to create a password. And, it

[6]"The Economy of Credential Stuffing Attacks" listed on https://www.recordedfuture.com/credential-stuffing-attacks/

[7]https://twitter.com/kevinmitnick/status/545432732096946176?lang=en

doesn't need to be complex, only long. The password *dhr*$_*$*Qdfe* is much less secure than *dog* . . . , let alone a much longer *sausagedog* . . . !

6 Use Passphrases Instead of Passwords

The comedian John Oliver interviewed Edward Snowden[8] and the topic of passwords came up. We should forget about passwords, and think passphrases. Rules we can follow are:

1. Still use the mix of characters (upper/lower/numbers/special characters)
2. Use a combination of unrelated words
3. Use words from different languages—a mix is best
4. Do not rely on leetspeak/133t5p3@k alone (where letters are replaced by similar shaped numbers/special characters)
5. Do not rely on one rule alone!

As an example, let's base a passphrase on that favourite fermented curd—cheese. In using a combination of the rules above, my passphrase could be *Ch3ese&Kase&Farmaajo*. After all, who could forget cheese! At 20 characters, that would give the password crackers a run for their money.

Or, think song lyrics. Something like *4!We!Are!Young!And!Free.*[9] Even harder to crack, at 23 characters. Each character exponentially increases the number of combinations, so longer is better. Although, *MargretThatcherIs110%Sexy* still takes the prize for sheer creativeness.

What we need is time. If a breech is detected, we need time to ensure word gets out to those affected by the breech. So timely notification is key from the organisations who become the victims of attack. The longer the passphrase is, the longer it takes to crack.

We could go even further, and use a password manager. These tools will allow a secure container into which your credentials and passphrases can be stored. They are useful, in that they can allow secure passphrases to be auto-generated, stored, and most importantly made unique for every separate site or system accessed. Some also come with wallets to store payment information, and can work both in desktop and mobile environments. Both commercial and opensource tools are available.

The downsides of these tools can be:

- What password/phrase do you have to access this tool? If it's weak, it would reduce the usefulness of the tool.
- What security is built into this tool itself? Could the encryption it uses be an older, compromised version?

[8] https://www.youtube.com/watch?v=yzGzB-yYKcc—please watch this video—in 3 min you will know how simple passphrase security can be.

[9] The second line of the Australian national anthem.

7 About Usernames

But passphrases are only half the battle. What about usernames? How many people have a common username (usually an email address) across many different websites? It's interesting, in that often we are asked to input our email address for access to a catalogue or whitepaper. Or "Join for free" to receive great discounts. Or join with your Google or Facebook account. This can spread your information far and wide. And, if one of those sites you fed your details into is attacked, and your user credentials stolen, the impact could be much wider. Now, your email could go into that list of addresses targeted for attackers to use in phishing attacks. Some methods to avoid this include using a short-term mail service like 10minutemail.com[10] for those sites that mail a link to the download you're after, or having multiple mail accounts (Gmail/Hotmail/etc.) to use for various site logins.

8 Conclusions

Am I being paranoid? My kids abound in their father's alleged paranoia, extending to my son's custom-made tinfoil hats, or my wife asking why we need multiple broadband accounts. But, as Philip K Dick[11] once said, "Strange how paranoia can link up with reality now and then . . . ".

Once an attack has been made, and data lost, there is the aftermath. The embarrassment for those who fell for the attack, and the looks they now get from colleagues around the office. Another danger present is a phenomenon called "Monday's Expert". After an event, everyone sees the mistakes that were made when pointed out. Think about that sports programme where each week the panel look at the weekend's games. Of course that player was offside/onside/committing a foul/not committing a foul/over the line/short of the line. It's blatant when we are shown the multitude of slow motion high-definition camera views, complete with added computer graphics. How did the referee miss that? We can, from a security point-of-view, fall victim to "it could never happen to me", as we roll our eyes and say knowingly to colleagues beside the water cooler "How could they ever let that happen?"

But, play the event back at regular speed. Would you make the right/same/a different decision? We must appreciate that when faced with a decision, people always have the option of choosing the right/wrong/sub-optimal/a different outcome. They may not have enough information or knowledge of the background situation, and yet are asked to make that decision RIGHT NOW. That is where training can help.

[10] https://10minutemail.com/10MinuteMail/index.html

[11] American science fiction writer, whose books were the basis for such films as *Bladerunner*, *Minority Report*, *Total Recall* and *The Man in the High Castle*.

I'm not talking about turning every employee into a security expert—that will not be a practical (or cost-effective) solution.

The basic training that's required should allow the organisation's staff:

1. To summarise the need for security to protect technology/process/people
2. To relate the motivation of a malicious user to the organisation's assets
3. To recognise potential security vulnerabilities in the day-to-day tasks of their own job role
4. To follow security processes!

There should also be a small team of people within the organisation who do specialise in security. The training for this team would go much further—allowing this group to write, test/audit, and maintain the organisation's security to the required level. It's up to them to continuously test these procedures, and ensure the people using them not only understand the steps, but the reasons behind why the steps are necessary.

Earlier, I mentioned time. It takes time for an attacker look for vulnerabilities, and to exploit them once found. It is everyone's job in the organisation (and our job for our personal lives) to reduce the possible vulnerabilities. But they will always exist. There might be a determined attacker who, based on MICE, might want to attack your organisation, or even you personally. You cannot stop all attacks, but you can make the time and resources needed to expend in the attack to be too high a price for the attacker to pay. It's like a cryptic crossword—many people look at it and don't even attempt it. A smaller number start, and might even get part way through to completing it. But a few will be either determined enough to complete it (but it takes a long time) or both determined and clever enough to do it quickly. Although these people are to be feared, they are not invincible. But, luckily, they are few in number, and the methods of defeating them are growing. But so are the methods they can use to attack. Security is a subject that if you are standing still, you are moving backwards faster than you would realise. Your aim is to make the resources needed to expend in the attack greater than the attacker is willing to put on the table. We must do this through reducing vulnerabilities contained within our organisation's technology, our processes, and, most importantly, our people.

Finally, let's hope it's not a nation state that wants your stuff. This attacker has a potentially unlimited set of resources—if they want your stuff, they will get it. The only way to stay safe is to switch off all internet connected devices, destroy them, then go and live in a cave. Putting on tinfoil hat now . . .

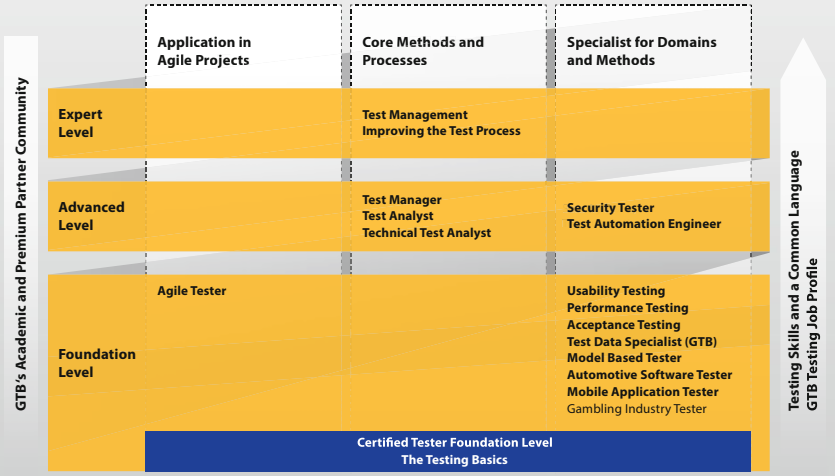
Reference

1. Hadnagy, C., Fincher, M.: Phishing Dark Waters. Wiley, Hoboken, NJ (2015)

GTB
German Testing Board

Software. Testing. Excellence.

ISTQB®/GTB Certified Tester Scheme



TESTING EXPERTISE FOR BEGINNERS AND PROFESSIONALS

As an association of committed subject matter experts and member of the ISTQB®, the GTB stands for the practice-relevant qualification of software and system testers.

GTB Premium Partners

CGI · EXCO · imbus · knowledge department · loyal team · Lysant · methodpark · philotech · QUALITYDOJO · sepp.med · software quality lab · sogeti · T-Systems · T-Systems

www.german-testing-board.info