TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

Tuukka Virtanen

LITERATURE REVIEW OF TEST AUTOMATION MODELS IN AGILE TESTING

Master of Science Thesis

Information and Knowledge Management

Examiner: Prof. Samuli Pekkola
Examiner and topic approved by the Faculty Council of the Faculty of Business and Built Environment on 26th of March 2018

ABSTRACT

TUUKKA VIRTANEN: Literature review of test automation models in Agile testing
Tampere University of Technology
Master of Science Thesis, 104 pages, 5 Appendix pages
May 2018
Master's Degree Programme in Information and Knowledge management
Major: Knowledge management
Examiner: Professor Samuli Pekkola

Keywords: agile testing, test automation, software testing, prescriptive modelling, agile practices, testing tools, literature review

Test automation is considered to be a crucial part of a modern Agile development team. Agile software testing methods and development practices, such as Test Driven Development (TDD) or Behavior Driven Development (BDD), continuously assure software quality during development time, from project start to finish. Agile software development methods require Agile testing practices for its implementation. Software quality is built-in and delivering functional and stable software continuously is a key business capability. Automated system and acceptance tests are considered as a routine part of the Continuous Integration (CI) and Continuous Delivery (CD) pipeline.

The objective of the research was to study, what test automation models, Agile practices and tools are found in Agile test automation literature and what kind of generic Agile test automation model can be synthesized from this literature. The objective was completed by conducting a systematic literature review of test automation models. The initial search included fifty scientific articles, from which ten models were selected for further analysis.

The selected articles and their models were modelled using prescriptive modelling. The tools and Agile practices mentioned in the articles were recorded and categorized. Each model was also categorized according to its domain of application. Using the collected data, a synthesized generic model for Agile test automation model was created.

Test automation models proved difficult to evaluate as the models were vastly different from each other in their description, depth of detail, utility, environment, scope and domain of application. A generic Agile test automation model would be characterized with agent, activity, artefact and event elements. It would have a functional information perspective and would be formally presented in text and graphic form. Continuous Integration was identified as the most popular Agile development method and Scrum as the most popular Agile management practice. Continuous Integration was also identified as the most popular tool category.

TIIVISTELMÄ

TUUKKA VIRTANEN: Kirjallisuuskatsaus testiautomaatiomalleista ketterässä ohjelmistotestauksessa
Tampereen teknillinen yliopisto
Diplomityö, 104 sivua, 5 liitesivua
Toukokuu 2018
Tietojohtamisen diplomi-insinöörin tutkinto-ohjelma
Pääaine: Tiedon ja osaamisen hallinta
Tarkastaja: professori Samuli Pekkola

Avainsanat: agile, testiautomaatio, ohjelmistotestaus, ketterät menetelmät, testaustyökalut, kirjallisuuskatsaus

Testiautomaatio on tärkeä osa ketterää ohjelmistokehitystä. Agile-metodologian mukaiset kehitysmenenetelmät, kuten Test Driven Development (TDD) tai Behavior Driven Development (BDD), varmistavat ohjelmiston toimivuuden koko kehitysajan alusta loppuun asti. Agile-testauksen neljä kvadranttia ja ketterän testauksen menetelmät varmistavat jatkuvasti laadun ketterässä ohjelmistokehityksessä. Automatisoidut integraatio- ja hyväksymistestit ovat rutiininomainen osa jatkuvan integraation kehitystä (Continuous Integration, CI), jossa muutokset lähdekoodiin laukaisevat automaatisoidut testit.

Tutkimuksen tarkoituksena oli tutkia, mitä ketteriä testiautomaatiomalleja, menetelmiä ja työkaluja löytyy alan tieteellisessä kirjallisuudessa ja syntetisoida sen pohjalta geneerinen testiautomaatiomalli. Tämä tavoite saavutettiin suorittamalla systemaattinen kirjallisuuskatsaus ketterään testiautomaatiokirjallisuuteen ja siinä mainittaviin testiautomaatiomalleihin. Haun perusteella luettiin viisikymmentä tieteellistä artikkelia, joista valittiin tarkempaan analyysiin kymmenen mallia, jotka sisälsivät myös mallia hyödyntävän casetapauksen.

Valitut testiautomaatiomallit mallinnettiin käyttäen kuvailevaa mallintamista (prescriptive modelling). Artikkeleista kirjattiin ylös maininnat testaustyökaluista ja ketteristä ohjelmistokehitysmenetelmistä. Testiautomaatiomallit myös luokiteltiin niiden käyttötarkoituksen (domain) mukaan. Hyödyntäen artikkelien dataa syntetisoitiin geneerinen malli testiautomaatiomallin kuvaamiselle.

Testiautomaatiomallien vertailu osoittautui vaikeaksi, koska mallit erosivat toisistaan paljon kuvauksessa ja sen tarkkuudessa, ympäristössä ja käyttötarkoituksessa. Geneerinen malli on kuvattu käyttäen agenttia, toimintaa, artefaktia ja tapahtumaa. Geneerisen mallin informaationäkökulma on funktionaalinen ja se on esitelty formaalisti käyttäen formaalia kieltä ja grafiikkaa. Continuous Integration oli suosituin ketterä kehitysmetodi ja Scrum johtamismetodi. Continuous Integration oli myös suosituin kategoria testaustyökaluille.

PREFACE

Writing this document feels like the end of a long journey, which it has been. But it also feels like the culmination of sixteen years of studying, beginning from Elementary School through Secondary School and Gymnasium up until Tampere University of Technology. After graduation, I'm throwing myself into the unknown. Once again. Like when I found testing. Testing is something that came to me by surprise. This document was created under a period of twelve months under which I have learned an enormous amount from testing – working as an analyst for Sogeti Finland Oy has been a place for professional growth as a tester and a human being.

I would like to thank Prof. Samuli Pekkola and my manager Sami Koivumäki at Sogeti Finland Oy for assisting in the conduction of this research and for their continued support and insights. I would also like to thank my twin brother Erkka, my mother Jaana and my father Kari, my partner Nelli Leinonen, all the people at Sogeti and Veikkaus who have contributed their knowledge about testing.

I would like to end this Preface with a brilliant quote by Timothy Leary that summarizes my thoughts about this learning journey of life and how one must constantly throw themselves into the unknown:

"Throughout human history, as our species has faced the frightening, terrorizing fact that we do not know who we are, or where we are going in this ocean of chaos, it has been the authorities, the political, the religious, the educational authorities who attempted to comfort us by giving us order, rules, regulations, informing, forming in our minds their view of reality. To think for yourself, you must question authority and learn how to put yourself in a state of vulnerable, open-mindedness; chaotic, confused, vulnerability to inform yourself."

Espoo, 18.03.2018

Tuukka Virtanen

TABLE OF CONTENTS

ABBREVEATIONS

ATDD	Acceptance Test Driven Development
BDD	Behavior Driven Development
CD	Continuous Delivery
CI	Continuous Integration
DDT	Data Driven testing
DevOps	Development and Operations
DTS	Defect Tracking System
JVM	Java Virtual Machine
LoC	Lines of Code
MBT	Model Based Testing
PRA	Product Risk Analysis
QA	Quality assurance
SbE	Specification by Example
SoC	Separation of Concerns
SUT	System under test
TDD	Test Driven Development
XP	Extreme Programming

1. INTRODUCTION

1.1 Research background

Many companies are adopting Agile testing methodologies in their software development and testing processes. Agile software testing methods, such as Test Driven Development (TDD), have introduced new ways of continuously assuring software quality during development time, from project start to finish. In Agile, the software quality is *built-in*. Delivering functional and stable, quality software continuously, is recognized as one of the key business capabilities in the software industry. Software quality is put in focus and automated system and acceptance tests are considered as a routine part of the Continuous Integration (CI) and Continuous Delivery (CD) pipeline. Test automation is an important part of the delivery pipeline, ensuring always working software and freeing testers from doing manual development and regression testing to do more exciting Exploratory Testing.

Development teams are now wondering, how to efficiently organize and automate their software testing efforts – what are the key features of test automation in an Agile environment? What is the role of test automation in Agile software testing? How does it differ from traditional, plan-driven testing? How could Agile teams use test automation to further develop their CI & CD processes?

Agile testing is a movement that focuses on *continuously* delivering the best possible software product. The term was first introduced by author Lisa Crispin, in her nominal work *Agile Testing* in 2009, where Crispin highlights the differences between Agile and waterfall testing. After the success of Agile testing, many development methods with a similar focus on testing gained popularity, such as Agile Acceptance Testing (AAT), Acceptance Test Driven Development (ATDD), Example Driven Development (EDD), Story Testing (ST), Behavior Driven Development (BDD) and Specification by Example (SbE). Extreme Programming (XP), Scrum and Kanban and other industry practices are also often bundled under the names of *Agile* and *Lean* (Adzic 2011). Detailed description of Agile testing and Agile testing activities is provided in Chapter 2.1.

In Agile testing*, test automation* is seen as the key driving force in generating benefits compared to traditional testing. Test automation is used to reduce manual testing and total testing time and resource waste. Automation is applied routinely to unit tests and components tests. Automated tools are applied for performance and load testing, security testing and usability testing. Both automated and manual activities are used together in functional tests, examples, prototypes and simulations. Automating redundant testing tasks leaves more time for manual exploratory and user acceptance testing. The purpose of test automation is not to detect new defects or to find inadequate business logic specifications. Test automation is practiced to provide *continuous* assurance of software functionality, which enables the developers to have confidence in the quality of their software. Detailed description of test automation practices and usage is provided in Chapter 2.5.

In Agile testing literature, multiple different models and frameworks for organizing and applying test automation exist. These models vary considerably in their description, depth of detail, utility, environment, scope and domain. The purpose of this research is to understand, how test automation is depicted in Agile testing literature by reviewing, categorizing and analyzing the differences between these models.

1.2 Research objective

The objective of the research is to discover, what test automation models, Agile practices and tools are found in Agile test automation literature and what kind of generic Agile test automation model can be synthesized from this literature. Ten scientific articles are selected for the literature review, in an effort to find out, how is test automation organized in Agile testing environment. The research aims to create and sample qualitative information about test automation practices and their evaluation in the domain of Agile testing and test automation. The research methodology is explained in detail in Chapter 4.1.

The selected test automation models will be modelled utilizing *prescriptive modelling* (Acuna et al. 2001). Prescriptive modelling is utilized to standardize model differences by characterizing models by their process elements and their relations. These process elements can be categorized in to three groups, including *model criteria*, *representation criteria* and *methodological criteria*. Detailed definitions for the criteria for different procedure elements are described in Chapter 4.1.6. In addition to criteria, models are categorized according to their domain or use case, e.g. open-source or safety-critical. This categorization is utilized to discover if test automation models have differences or similarities between groups.

After reviewing each article, the process model, or part of the process model, is depicted using Microsoft Visio in unified manner to help comparison. To gain insight of the tools and Agile practices that are used within the literature, the tools and Agile practices that are mentioned in the articles are recorded. Tools are categorized by their use case to understand what type of tools are most utilized in test automation. Agile practices are categorized to *development* and *management* methods to distinct between the technical and organizational managerial practices. The collected data and characterization model are used to synthesize a generic test automation model for Agile testing environment.

1.3 Research questions

From the research objective, the two main research questions were formed

- What test automation models, Agile practices and tools are found in Agile test automation literature?
- What kind of generic test automation model can be synthesized from Agile test automation literature?

To answer the first research question, a systematic literature review of Agile test automation literature is performed. After searching and reading fifty (50) scientific articles, a sample of ten (10) articles containing an Agile test automation model are selected for review and modelled using prescriptive modelling. Prescriptive modelling is used to prescribe models with standard definitions of model criteria, representation criteria and methodological criteria for effective comparison. The domain of application, practices and tools mentioned in each article are recorded and categorized. A table containing all models and their attributes is presented for effective comparison.

The second research question is answered by synthesizing a generic Agile test automation model description, using the data collected in the first question. The model will be synthesized from the most popular attributes mentioned in the literature. Discussion of model viability is performed.

To support the discussion of research results, following supporting questions were formed to aid discussion

- How to evaluate different Agile test automation models?
- What characteristics describe Agile test automation models?
- What domains are found in Agile testing literature?
- What Agile practices are found in Agile testing literature?
- What tools are found in Agile testing literature?
- How does the synthesized generic model compare with Agile testing literature?

The supporting questions provide context for discussing the research results, differences between models and comparing them to research literature. For the discussion, the forty (40) read articles excluded by the methodological inclusion criteria, will be used for the literary comparisons. The supporting questions and their answers are presented in Chapter 5.3.

1.4 Research scope

The research scope is limited to qualitative information about the state of test automation in Agile testing literature. Agile methods and practices are presented in level of detail needed for understanding their application. Full examination of Agile methods and practices and usage in other contexts are left out of scope. Technical implementation details of the researched Agile test automation models are left out of scope. Technical implementation or viability of the synthesized Agile test automation model is left out of scope.

1.5 Research limitations

The research is limited by time and resources of the researchers. This sets limitations to the research scope, accuracy and validity. The chosen search methodology and search strategy set limits to the possible results. The research is limited to discovering qualitative information about test automation aspects in Agile testing. The data collected about domain, Agile practices and tools is not statistically valid due to small sample size (ten articles). The data provides a qualitative characterization for the inspection of different Agile test automation models. The research is limited to synthesizing only a theoretical generic Agile test automation model that will not be tested in a real-life business case.

The research is limited by the number of articles available for search in used bibliographic databases. Utilizing only one search portal (Andor) limits the list of available articles. The human resources available for reading and reviewing the articles limits the number of read articles to fifty (50) and to ten (10) for reviewing. The research accuracy is limited by the accuracy of search terms, selection criteria and inclusion and exclusion criteria. Used search terms limit what kind of literature will be found. Relevant literature might be excluded. The research validity is limited by the validity of the prescriptive model characterizations and the conclusions drawn from them. Subjective differences between different characterizations and categorizations may differ.

1.6 Research structure

The structure of the research is as follows

- Chapter 1 gives an introduction into the research background, objective, scope and limitations.
- Chapter 2 contains theory about Agile testing and test automation.
- Chapter 3 lists Agile testing practices and their definitions.
- Chapter 4 contains research results.
- Chapter 5 contains discussion of research results.
- Chapter 6 contains research conclusions.

2. AGILE TESTING AND TEST AUTOMATION

2.1 Definition of Agile testing

Agile testing is defined by the words of the *Manifesto for Agile Software Development (2001)* and the notion of software quality being built-in – and in the case of software testing – tested-in. The four main statements of Agile say that we value

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan.

Since its original conception in 2001 to 2018, the Agile manifesto and its teachings have become *de facto* standard for organizing software development in the rapidly progressing field of software development. As the term's usage has expanded, the practical usefulness of the term itself has diluted in value. The word "Agile" is used so often in so many different occasions, that it has lost its original meaning – *Agile* is used as "catch-all" term that includes a collection of industry standards, practices and methodologies, applied in practice in various ways. In the same vein, *Agile testing* is used to refer to a collection industry standards, practices and methodologies used for software testing in Agile software development.

Agile testing encompasses all the different aspects of software testing related tasks in an Agile environment, including (but not limited to) manual testing, test automation, reporting found defects, documenting system behavior, but it can also be used to refer to the programming style used for the software development. Some of the names used to refer to practices similar or almost similar are Agile Acceptance Testing (AAT), Acceptance Test Driven Development (ATDD), Example Driven Development (EDD), Story Testing (ST), Behavior Driven Development (BDD) and Specification by Example (SBE). Terms for industry practices, such as, Extreme Programming (XP), Scrum and Kanban are also often bundled under the names of *Agile* and *Lean.* (Adzic 2011).

In her nominal work *Agile Testing* (2009), Lisa Crispin argues that Agile testing simply means to continuously focus on delivering the best possible product. Crispin divides Agile testing into four distinct *Agile testing quadrants* and highlights the differences between Agile testing and traditional, plan-driven waterfall testing. The most important differences are a result of the short iteration cycles of Agile development compared to traditional development. Testing has been traditionally done as the last part of the development process, when all the feature development work has ceased, acting as a final "gatekeeper" to releasing software into production.

2.2 Comparing Agile testing and Plan-driven testing

Agile product development follows short iteration cycles, usually from 1 to 4 weeks, which means that testing activities also have to happen within these cycles. Traditional plan-driven, waterfall-phased product development is defined by a sequence of chronological steps, beginning from gathering requirements, defining product specifications, developing product code and ending with testing activities and release into production. Differences between approaches are illustrated in Figure 1.

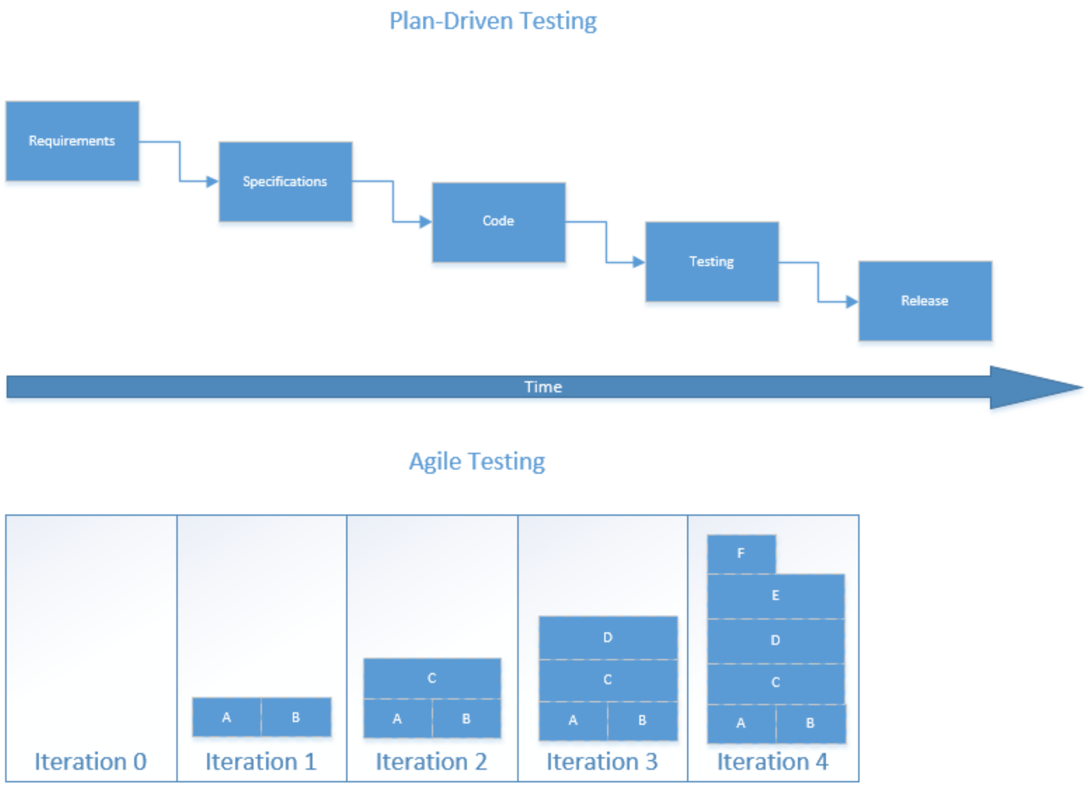

Figure 1. *Plan-driven testing vs. Agile testing (Crispin 2009, p. 13)*

Plan-driven development is based on the assumption that project requirements are fixed and do not change during the course of development. Comprehensive planning and test design can be executed upfront, as the formal exit criteria is known. Documentation is extensive and all functionality and features are documented in detail, which testers can utilize in finding information about the system under test (SUT) and in their test design. Test automation is built by tool specialists and only after the code is considered done. Regression tests are not automated during development time.

A certain amount of development time is allocated to testing and testing can begin once most of the programming has been completed. In real-life projects, product development can suffer from delays or unprecedented setbacks, which usually result in a need to lengthen the time allocated to programming phase. This extra time shortens the amount allocated to testing, which leads to less time for test execution and defect findings and ultimately, to lower software quality.

The Agile approach to software development and testing is iterative and incremental. Testers begin testing features as soon as the code is incremented. Test design and planning is done iteratively as the project progresses and documentation is usually scarce or non-existant. Each feature, use case or User story is is expanded, programmed and tested during the cycle. Features are not considered done unless testing has been executed.

Test automation is considered key to successful Agile projects. Developers or testers or any team members with test automation knowledge automate tests for each new feature and they serve as regression tests during the development. Keyword-driven tests separate the programming from the writing of the actual test scripts so that they can be developed without knowledge of programming. Zhongqian et al. (2013) argue that keyword-driven tests are easy to create without programming knowledge, they can be developed earlier and are easier to maintain when compared to traditional test automation.

Below is Table 1 by Hendrickson (2004) representing key differences between Plan-driven and Agile testing.

Table 1. *Plan-driven testing versus Agile testing (Hendrickson 2004)*

	Plan-driven testing	*Agile testing*
Change	Manage & control it	Change is inevitable – embrace and expect it
Planning/test design	Comprehensive upfront plan/test design	Plan/design as you go
Documentation	Can be heavy	Minimized – Only as much as necessary
Handoffs	Formal entry/exit criteria	Team Collaboration
Test Automation	System level built by tool specialists, created after code is 'done'	All levels, built by anyone, an integral part of the project

2.3 Agile Testing Quadrants

Software quality can be measured in many dimensions; as the ability to perform required functionality or as the ability to satisfy customer needs and they both require a different approach software testing. Because of this vast amount testing possibilities, the question is, where should time and resources be allocated? The question may seem arbitrary, but in fact it is a quite complex and philosophical question about the nature of our software product – when is software considered to be done, i.e. no more software testing is required? What is the D.O.D, definition of done? Contemporary wisdom tells us that in the Agile world, software is never considered "done". The software is in a constant state of development and software testing is only finished when the software reaches the end of its life cycle and is abandoned and replaced by a new piece of software.

Crispin (2009) introduces *Agile Testing Quadrants* as a theoretical framework to approach Agile software testing in multiple different dimensions. It provides a theoretical framework to assess which dimensions of software are tested and in which ways. It is based on the Agile testing matrix presented by Marick (2003). The Agile Testing Quadrant framework categorizes Agile testing tasks into four quadrants

- Automated (Q1)
- Automated & manual (Q2)
- Manual (Q3)
- Tools (Q4)

that face four different dimensions of software quality

- Business-facing
- Team-facing
- Technology-facing
- Product-facing

The Agile Testing Quadrants are illustrated in Figure 2.

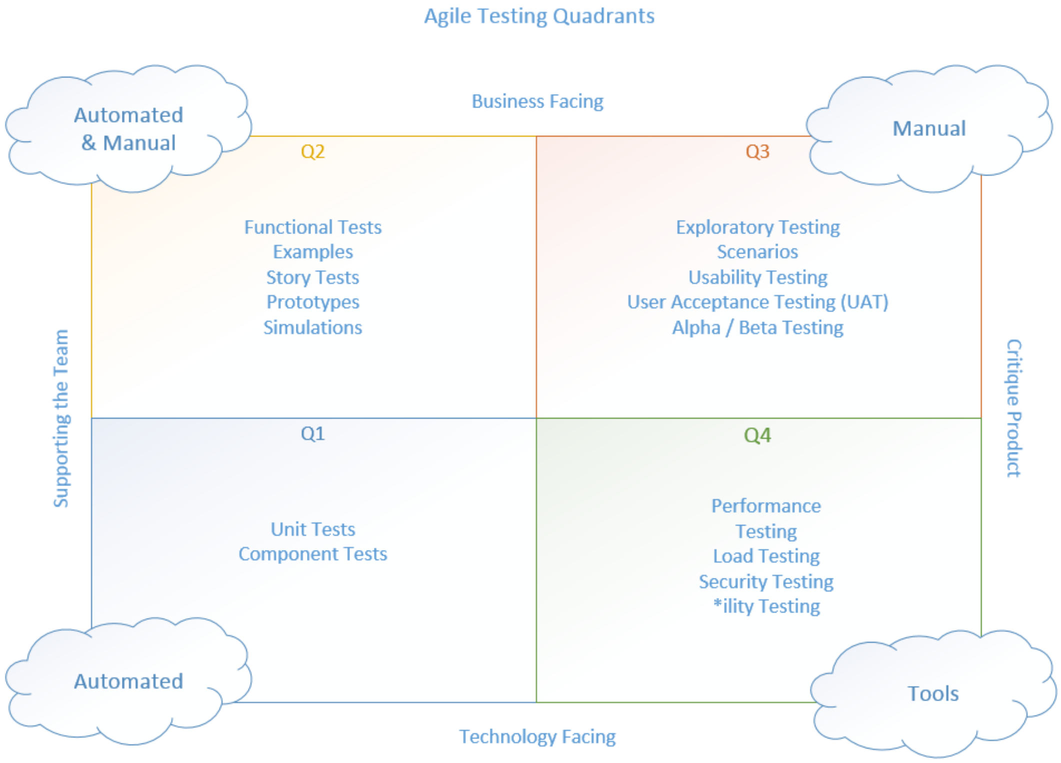

***Figure 2.** Agile Testing Qudrants (Crispin 2009, p. 98)*

2.3.1 Q1: Automated testing

Quadrant 1 represents Test Driven Development (TDD). Programmers are to adopt any of the Test Driven Development methodologies, such as Behavior Driven Development (BDD) or Specification by Example. The key is to automate unit and component tests as they are being developed. The automated unit and component tests ensure constant software functionality and serve as automated regression tests when new features are added. The unit tests are written in the same programming language as the development language and technical expertise is required to understand them. The internal quality of the code is defined by the programmers, not by external customers, as code units usually contain proprietary information and design choices that are to remain internal. The process of writing unit tests also forces programmers to question their architecture and design choices and to design the architecture to be easy to test and automate.

2.3.2 Q2: Automated & manual testing

Quadrant 2 represents tests, such as functional tests, examples, story tests, prototypes and simulations that support the development team on a level higher than unit tests. These tests are external and facing business and customers and help to define the external quality level and features required. These tests are written on a functional level that can be understood by business analysists and they can also write them.

2.3.3 Q3: Manual testing

Quadrant 3 represents Exploratory Testing, scenarios, usability and user acceptance tests. Alpha and beta testing is also in this quadrant. These tests usually require both automation and manual testing and Exploratory Testing is central to them. Test automation might be used to automate the creation of the data sets used in these tests, but manual Exploratory Testing is used to explore program paths and to find new defects. This task requires a certain level of intuition and knowledge about the system under test and is heavily dependant on the skills and experience of the tester. Hard to define, subjective quality metrics, such as usability, fall under this category.

2.3.4 Q4: Testing tools

Quadrant 4 represents technology related tests. Characteristics such as performance, robustness and security are in this category. These tests are highly specific to the used tools and technologies and their design choices. They ensure that the technical aspects, that can not be measured with straightforward functional requirements but that are affecting the user experience, are tested.

2.4 Agile testing team and quality assurance organization

Agile product development team works as one and they encourage “whole team” approach (Crispin 2009). All project participants should view themselves as one team working together to achieve a common goal. Traditional, functional teams have clearly defined roles for each team member, such as, programmers, business analysts and testers, who each work their special domains with little overlap. Both team structures are illustrated in Figure 3 below.

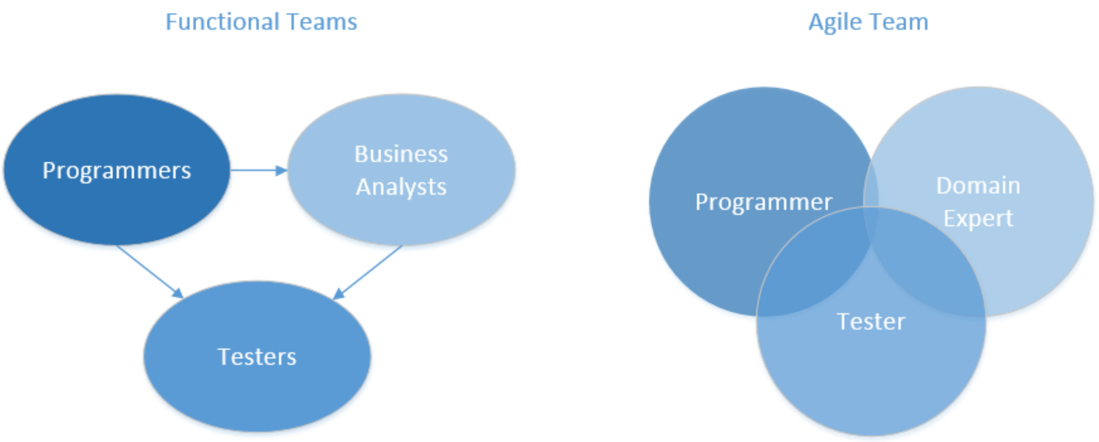

Figure 3. *Agile team structure vs. Functional team structure (Crispin 2009, p. 64)*

In functional teams, testers gain test specifications and requirements from programmers and business analysts. Business analysts work with programmers in defining test specifications to test the appropriate business logic and functionality. Testers work to fulfill testing requirements but do not actively participate in their design or refinement.

2.4.1 Agile testing teams

In Agile teams, testers and all other team members are expected to work closely together and team roles are less strictly defined. Testers are encouraged to share their findings and domain knowledge with programmers and external team members. Agile teams stress the importance of face-to-face communication and consider it critical to the success of a project. (Crispin 2009, p. 59). Translating and effectively communicating business requirements is a requirement. Testers work in a domain that overlaps with programmers and business analysts and work to link and translate business requirements to programmers and verify them.

In an Agile testing team, the tester is fully integrated into the development team and the role expands from manual testing to test automation. Skills of script writing and programming are required to write tests that minimize unnecessary regression tests and manual testing (Ottosen 2016). Test automation frees time for the tester to be used more productively, such as doing Exploratory Testing.

The Agile tester is always trying to improve their skills in test automation and software development. The Agile tester must have basic understanding of computers science and programming. They must also have a good understanding of web programming standards and most commonly used tools and operating systems, like command lines tool ins UNIX envinronment. Common degrees for testers include computer science background or corresponding computer engineering background.

2.4.2 Quality assurance organization

Depending on the organization, Quality assurance (QA) can be organized as a part of the Agile development team or as an independent functional team. The independent QA team provides testing and *Quality assurance as a service* (QAAAS) to the other parts of the organization. Reasons for having an independent QA team include stressing the importance of an independent check and audit role, having an unbiased view of the quality of the product and the want to separate testers from developers to avoid likeness. One common mistake is to confuse “independent” with “separate”. If production budget and processes are kept in discrete functional areas, a division between developers and testers is inevitable. This can lead to competition between groups and cause friction within the organization. Crispin (2006, p. 60) openly advocates for having the QA team as *a community of testers* working on different Agile product development teams. The QA community should provide a learning organization for testers to help them with their career development and share knowledge with each other. (Crispin 2006)

The Agile “whole team” approach has forced many organizations the send their testers to work within the Agile project teams. Arguments for the Agile approach include being closer to the development of the product, building better team spirit, sharing responsibility of quality (Crispin 2006, p. 63). Testers should be full-fledged members of the development team. Testing is not seen as the “necessary evil”, but as a critical part of the product development process. Each sprint defines the timeline that starts with planning and ends with testing and quality assurance. Teams should try to avoid the common “small waterfall” mistake, where developers spend a week writing code and the testers spend the next week testing it. Agile team members are cross-functional and each team member usually has multiple areas of domain expertise. Agile team members are capable of fluently transitioning between QA and developer roles. (Crispin 2006).

Table 2 by Pettichord (2000) lists a number of characteristics for testers and developers.

Table 2. *Tester and developer characteristics (Pettichord 2000)*

Testers	Developers
Get up to speed quickly	Thorough understanding
Domain knowledge	Knowledge of product internals
Ignorance is important	Expertise is important
Model user behavior	Model system design
Focus on what can go wrong	Focus on how it can work
Focus on severity of problem	Focus on interest in problem
Empirical	Theoretical
What’s observed	How it’s designed
Sceptics	Believers
Tolerate tedium	Automate tedium
Comfortable with conflict	Avoid conflict
Report problems	Understand problems

2.5 Definition of test automation

Test automation is typically defined as “the use of a separate software from the testable application to control and execute test cases against defined specifications”. In general, the term “test automation” could be used to refer to all software interactions that can be atomized into unitary steps, sequentially linked together and imperatively executed. The broader definition includes automating parts of software delivery and testing pipeline, such as writing build scripts for Continuous Integration servers, updating test results automatically to test management software or generating large amounts of random input data for test execution. Building and maintaining a test automation framework falls under the category of test automation tool development.

Test automation is a distinct area of modern software development that resides between the domains of quality assurance (QA) and Continuous Integration (CI). Test automation has a crucial part in ensuring software quality during its development and maintenance. Specifically, in Agile development environments, where time is considered a luxury, test automation is used as a part of the software development process, such as TTD (Test Driven Development), to ensure continuous testing during development. Test automation can be seen to cover five different areas of work, categorized as regression test automation, development test automation, work flow automation, process automation and verification and validation.

This Chapter is structured as follows: First, the typical usage and benefits of test automation are presented in Chapter 2.5.1, Second, the typical test automation categories are listed in Chapter 2.5.2 – 2.5.5. The last Chapter 2.5.6 presents a list of commonly used test automation tools.

2.5.1 Typical use

Test automation is typically used to automate the repeated testing of time consuming, complex tasks. Test automation provides key advantages over manual testing, such as, the length of the testing phase is significantly reduced, number of defects found before going into production increases and manual regression testing is no longer required (Adzic 2011, p. 39). Test automation is less error prone than manual testing and it guarantees that test steps are always reproduced exactly the same way, while also providing test results traceability in the form of test execution logs. Prerequisite manual testing tasks, such as, creating test data, test initialization and teardown can be automated to save more time for the actual testing. Test automation is not well suited for finding defects in program logic or usability or other user perceived metrics. Finding new program pathways, user experience or usability defects are most effectively assessed with manual Exploratory Testing.

2.5.2 Development test automation

Development test automation refers to automated tests that ensure development phase software quality and provide feedback on the state of the program. Developers are responsible for developing unit tests for their components and for maintaining and updating them as the program evolves. Test Driven Development, TDD, uses development tests as a starting point for software development.

2.5.3 Regression test automation

Regression test automation refers to automated tests that ensure software quality between software updates (Kandil et al. 2016). Regression test automation is a main component in Agile testing and development. A test automation suite is built by a test automation developer for the system and continually assures working software. Regression tests, such as unit tests or acceptance tests, are continuously run on CI-server and provide continuous feedback.

2.5.4 Workflow automation

Workflow automation refers to the automation of testing workflows in Agile testing context. Workflow should be considered broadly to refer to “the execution path of a task”. In a test automation team, this could mean writing a bash script to automate the zipping and unzipping of test result screenshots or to automating the test data creation for acceptance tests.

2.5.5 Process automation

Process automation refers to the automation of a process for testing actitivies in the context Agile testing. Process automation consists of achieving process-to-process interoperability. In a test automation team, this could mean automating the process of collecting customer feedback for test data usage. Automated processes can range from trivial parsing problem to complex business logic descriptions.

2.5.6 Verification and validation (V&V)

Verification and validation (V&V) are the desired outcomes of using test automation. IEEE standard (1984) defines software verification as “the process of determining whether or not the products of a given phase of software development cycle fulfill the requirements established in the previous cycle” and software validation as “the process of evaluating the software at the end of software development cycle”. Test automation can be used to automate tests to verify that the required sprint features are functional or to validate standard checks to confirm production level quality.

2.5.7 Typical test automation tools

Typical test automation tools can be divided into general programming languages and automation frameworks and to “Capture and replay” tools. General programming languages are used to writing test cases in the same language as the SUT and to execute them imperatively. Automation frameworks, such as, Robot Framework and Cucumber, follow a keyword-driven approach (also described as Keyword Driven Development, KDD), which allow test case specifications to be written in natural language form.

When using the keyword-driven approach, the keyword implementations are done by test automation specialists, but natural language allows the test case implementation to be written by a business analyst or a tester. Common test automation frameworks are listed in Table 3. “Capture and replay” tools register tester’s interactions with the SUT and save them as scripts, which can then be automatically re-executed (Polo et al. 2013). Examples of commonly used software are TestComplete, Selenium and Appium Studio. Common “Capture and replay” tools are listed in Table 4. Tools have maturity and expertise level as defined by Polo et al. (2013) where 1=low, 2=medium, 3=high level of maturity and 1=beginner, 2=advanced and 3=expert level of required expertise.

Table 3. *Common test automation frameworks (Polo et al. 2013)*

Language	*Framework*	*Description*	*License*	*Maturity level**	*Expertise level**
Java	JUnit	The most famous XUnit framework for Java	Open-source	3	1
Java	JTest	A commercial tool that includes automated test generation and execution	Commercial	3	3
Java	JMock	An extension of the JUnit framework to create mock objects	Free	1	2
JavaScript	DOH	Runs tests in browser or independently	Open-source	3	2
JavaScript	QUnit	Tests any generic JavaScript code and	Free	2	3

		is very useful for regression testing			
JavaScript	JSTest.net	Enables JavaScript unit tests to be run directly in othe XUnit frameworks	Free	3	3
C/C++	C++ Test	A commercial framework that includes unit test generation and code coverage reporting	Commercial	3	3
C/C++	Cantata++	A commercial framework designed for testing embedded systems	Commercial	2	3
C/C++	Opmock	A stubbing and mocking framework for C and C++ based on code-generation headers	GPL	1	2
C/C++	Google C++ Testing Framework	A framework designed by Google to test C++ systems	Free	2	2
.NET	NUnit	A framework integrated in Visual Studio to create and run unit tests	Free	1	1
.NET	DbUnit.Net	An XUnit framework for testing databases	Open-source	2	3
.NET	MbUnit	A model-based XUnit framework	Free	2	2
.NET	QuickUnit.NET	Design tests without code and very helpful for TDD	Commercial	3	3

PHP	PHPUnit	An XUnit framework that reports results in XML and HTML, including coverage information	Open-source	1	3
PHP	Apache-Test	A PHP impelementation of Test::More	Open-source	3	3
PHP	Enhance PHP	An XUnit framework that includes mock and stub features	Commercial	2	3
Internet	HTMLUnit	An extension of JUnit that allows testing of HTML code	Open-source	1	3
Internet	Selenium	A record and replay framework that works with most Web browsers	Open-source	3	2

Table 4. *"Capture and replay" tools (Polo et al. 2013)*

Tool	*Technology*	*Description*	*License*	*Maturity level**	*Expertise level***
TestComplete	Multilanguage / multitechnology tool	Runs on Windows	Commercial	3	3
Abbot	Java	Designed for Java Interface	Open-source	1	2
Jacareto	Java	Doesn't generate scripts, can edit tests via GUI	Open-source	1	3

Selenium	Web	Generates scripts that testers can modify	Open-source	3	2
TPT's *Automated GUI Recorder*	Java	Generates scripts that testers can modify	Open-source	3	3
IBM *Rational Robot*	Multitechnology	Designed for e-commerce, enterprise resource planning and client/server applications	Commercial	3	3
PesterCat	Web	Generates scripts in XML	Commercial	2	2

2.6 Test automation levels

The International Software Testing Qualifications Board (ISTQB 2017) defines test levels as "groups of testing activities that are organized and managed together". Test levels are abstractions used to manage time and resources between different software development phases. Test level definitions vary between organizations, but typically test levels refer to three distinct software testing levels

- unit testing / component testing
- system testing / API testing
- acceptance testing / UI testing.

These levels are often referenced together with the software development V-Model and its corresponding levels. The test levels can be presented as a test level pyramid, in which the number of test cases is larger near the bottom and fewer near the top. In the case of test automation, the term *Test Automation Pyramid* is used to refer to test automation levels (Cohn 2009). Test Automation Pyramid is illustrated in Figure 4.

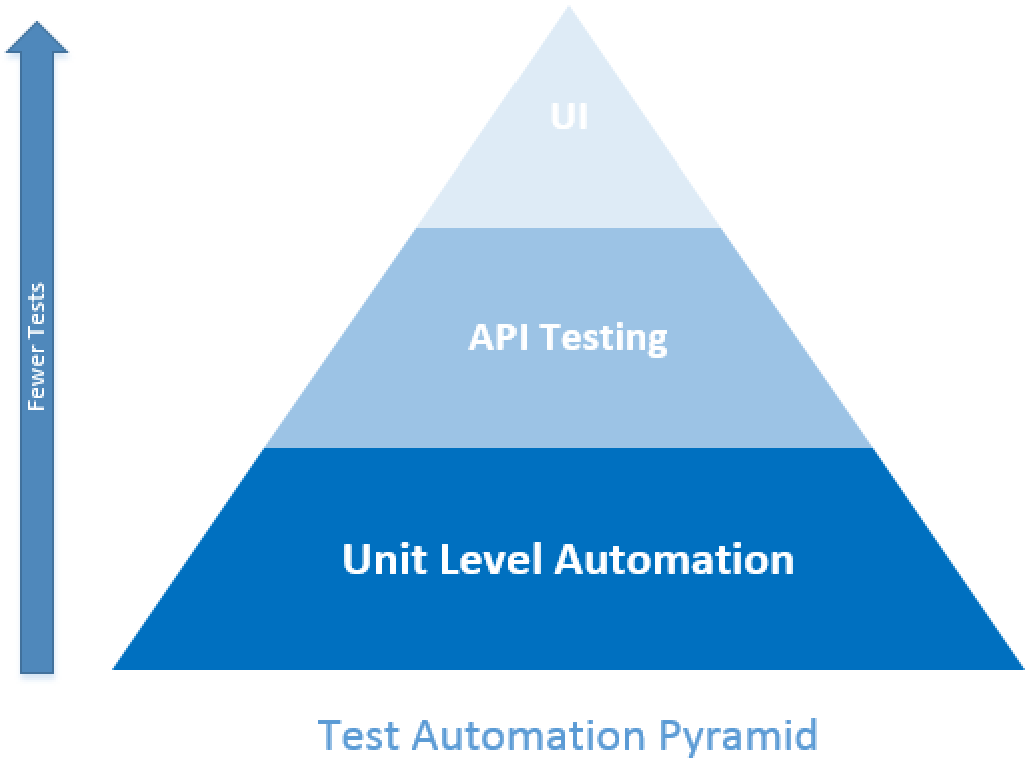

Figure 4. *Test Automation Pyramid (Cohn 2009)*

2.6.1 Unit testing

Unit testing, also referred to as module testing, development testing or component testing, refers to the testing of individual software components (ISTQB 2017). Unit testing is typically executed by the developer of the unit with the aim of demonstrating that the unit meets the requirements defined in technical specifications (TMap Next 2006, p. 82). Unit tests are designed to assure that the smallest parts of the program, such as routines, loops, and objects, behave as intended. Unit tests executing these parts are automated to display continuous quality of software units and robustness of the software.

In contrast to system and acceptance testing, unit testing cannot be organized as a separate task from development. The developer is an integral part of the unit test process. The developer has to have intimate knowledge of the SUT and the domain and to be familiar with common defects found in similar systems. Unit test approaches include input space partitioning, boundary values, error guessing, all combinations/pairwise/n-wise testing, test coverage criterions and mutation testing (Polo et al. 2013).

An example of a development test requirement is that all statements of the code should be evaluated at least once. This means that all program paths should be travelled to achieve *code coverage* of 100%. Other common requirements include condition coverage (CC), decision coverage (DC), condition / decision (C/D) coverage, modified condition / decision coverage or multiple condition coverage (MCC) (TMap Next 2006). The modified condition / decision coverage (MC/DC) is the required level of testing coverage for safety critical software. In avionics, the standard DO-178B (1992) requires that every possible outcome of condition is the determinant of the outcome of the decision, at least once (Tmap Next 2006). List of testing coverages and their explanations according to TMap Next (2006) are presented in Table 5.

Table 5. *Unit test coverage types (TMap Next 2006)*

Unit Test Coverage	*Explanation*
Condition coverage (CC)	The possible outcomes of (“true” or “false”) for each condition are tested at least once.
Decision coverage (DC)	The possible outcomes of the decision are tested at least once.
Condition / decision coverage (C/D)	The possible outcomes of each condition and of the decision are tested at least once. This implies both “condition coverage” and “decision coverage”.
Modified condition / decision coverage (MC/DC)	Every possible outcome of a condition is the determinant of the outcome of the decision, at least once.
Multiple condition coverage (MCC)	All the possible combinations of outcomes of conditions in a decision (therefore the complete decision table) are tested at least once. This implies “modified condition / decision coverage”

TMap Next (2006, p. 333) lists common arguments for and against development tests. Developers are under time pressure and their main focus is delivering product features and functionality. As such, there is incentive to cut time for testing, in favor of feature development. Human factors, such as taking pride in their development work, can cause developers to feel resentment towards testing or “doubting” of their development capability. It can also be argued, that the subsequent system and acceptance tests will also test the code unit functionality, though indirectly.

Arguments for the use of development tests are frequent. Their use decreases the amount of rework required after delivery, as subsequent levels are of higher quality. Planning is also more certain, as the volume of uncertain rework declines and the lead total development time shortens. Early rework is always more cost-effective than fixing defects later in product development. Developers gain faster feedback on their mistakes and development tests help them to better plan and design their software architecture and gives them confidence in their code quality and integrity. Tools for unit tests include debuggers, code-analysis and review tools and specific unit test frameworks.

2.6.2 System testing

System testing, also referred to as API testing, refers to the testing of system components and their interoperability in a simulated production environment (TMap Next 2006, p. 82). The purpose is to demonstrate that the system subsystems communicate between each other correctly and that the system as a whole meets the specified functional, non-functional and technical design requirements.

System test automation has the aim to assure full system operability continuously. Regression tests are written to assure that the correct system level functionality is executed and business logic functions as required by the system specifications. Short smoke test suites are used to quickly ensure critical software functionality on most common program pathways. Automated regression tests are key to achieving maintainable and robust, quality software. It makes further development easier and gives the developer a feedback loop to continuously monitor and improve their software. Many types of automated system tests exist: automated checksum-checks, automated job-pipelines, automated build scripts, automated text- and symbol-processing, automated verification and validation.

A list of system test automation strategies is presented in Table 6.

***Table 6.** List of system testing strategies*

Test Strategy	*Aim*
Regression testing	Continuous assurance of software quality by continuous execution of tests
Smoke testing	Quick confirmation of critical functionality by executing easy, fortunate program pathways
Stress testing	Assess how the system performs under exaggerated levels of stress
Performance testing	Assess how the system performs when under heavy load or traffic
Recovery testing	Assess how the system performs in recovery / black-out situations

There are multiple testing strategies to system testing and system test automation. Common testing strategies include regression testing, smoke testing, stress Testing, performance testing and recovery testing. The testing strategy can, for example, be chosen according to the Product Risk Analysis (PRA), which prioritizes most business-critical parts of the software before others. The chosen testing strategy is then converted into the test plan and then into test specifications and then to keywords in test automation scripts.

2.6.3 Acceptance testing

Acceptance testing, also referred to as user interface (UI) testing or user acceptance testing (UAT), refers to testing carried out by system users in an optimally simulated production environment to demonstrate that the developed system meets the requirements of the users (TMap Next 2006, p. 82). Acceptance tests include usability and other -ility testing, such as visual testing. Visual testing can be automated using optical image recognition software, such as Sikuli Script, fmbt-library or Applitools.

Acceptance test automation is usually based on keyword-driven testing. Common frameworks for automating keyword-driven testing are Robot Framework, Cucumber or FitNesse (Gärtner 2013). Keywords are implemented in general purpose programming languages such as Java, JavaScript or Python. A list of common keyword-driven test automations frameworks is Table 7.

Table 7. *A list of common acceptance test automation frameworks*

Framework	*Descriptions* *by their creators*	*Website*
Robot Framework	"A generic test automation framework for acceptance testing and Acceptance Test Driven Development (ATDD)"	http://robotframework.org/
Cucumber	"Open-source tool for executable specifications"	https://cucumber.io/
FitNesse	"The fully integrated standalone wiki and acceptance testing framework"	http://fitnesse.org/

The business logic is written using natural language like syntax, using previously programmed keywords. Business analysts and testers can understand and communicate business logic into test case specifications without the attention of the developer. Other acceptance test automation approaches include the page-object model, Model Based Testing, user stories and Given-When-Then-syntax.

A common syntax for writing automated test case specifications is Given-When-Then, meaning to describe a test case in three sections as illustrated in Table 8.

Table 8. *Given-When-Then (Fowler 2013)*

GIVEN	State of the world before test actions. Can be described as "pre-conditions".
WHEN	Behavior to be specified.
THEN	Changes expected due to behavior.

A common example for acceptance test automation is to automate the HTML user interface interactions and to simulate use cases. The developer could write a python script that operates a web browser, such as Google Chrome, using WebDriver- and Selenium -libraries. Robot Framework could be used as an example of a free and open-source, general-purpose test automation framework. It can be used to write keywords that can then be executed sequentially. These keywords would be made to match certain HTML elements, such as id or name, and then execute the defined keyword, such as click element. These keywords are sequentially placed according to desired programming logic.

Common arguments against acceptance test automation include their short life cycle. User interface is the part of the software that is exposed to the most frequent change. This means that after each small change to the user interface or business logic, the scripted user interface test fails, leading to maintenance update work. Therefore UI-tests cause consideradable workload to keep up to date.

2.7 Test automation in Agile testing

Test automation is a prerequisite for an Agile development team. With short delivery cycles of weeks or even days, extensive manual testing work piles up in one iteration and spills over to the next, rendering it unsustainable for fast moving Agile projects. (Adzic 2011, p. 39). Agile test automation is focused on automating tests for new product features. Test automation is executed without detailed business requirements. Test documentation is minimized (or presented by a specification by example) and active communication between the developers and testers is preferred. Most of the testing is automated and run daily on CI-server. The measurement of quality is continuous and the process flows without human intervention.

Automating functional tests is the starting point for test automation adoption. Some experts argue that everything that can be automated, should be automated. Several benefits can be acquired through automation such as:

- The length of the testing phase is significantly reduced
- Number of defects found before going into production increases
- Manual regression testing is no longer required
  (Adzic 2011, p. 39)

One of the key product development questions is, what capabilities should the product exhibit, in what timeframe, and with which and how many resources. The purpose of planning is described as “to arrive iteratively at an optimized answer to the ultimate new product development question of what should be developed” (Cohn 2006, p. 11). Cohn (2006) reminds us to consider that nearly two-thirds of projects significantly overrun their cost estimates, sixty-four percent of features included in products are rarely or never used and that the average project exceeds its schedule by 100%.

Regression test automation scripts are written to ensure that the software maintains full functionality. It ensures that no future update breaks any part of the software unknowingly. That is why regression tests are most valuable when run frequently. Their quality of providing information about software quality is inversely correlated with their update date.

Development test automation is usually mentioned with acronyms, such as Specification by Example (SbE), Test Driven (TDD) and Behavior Driven Development (BDD). They describe a way of working that starts the development process from the wanted test results. Test cases are written first and then the functionality is programmed to get that result. The final software product will have a fully functional regression test suite at the end of development, to ensure that the whole software is working according to specifications. A list of test automation development approaches is in Table 9.

Table 9. *Agile test automation development approaches*

Approach	*Description*
Test Driven Development (TDD)	Specify wanted test results first and then program implementation that matches the test specification
Behavior Driven Development development (BDD)	Specify wanted software behavior first and then program implementation that produces desired software behavior
Specification by Example (SbE)	Specify wanted examples first and then program implementation that expands on the examples
Acceptance Test Driven Development (ATDD)	Each part of program must pass an acceptance test before being merged into master branch

Test automation in an Agile team also means automating work flows between different systems, eventually automating and operating a Continuous Integration pipeline. In practice, this could mean writing bash scripts to collect test job results and merge them together into one xml-file and then uploading these test case results into TestRail through their API. This also includes writing a Slack-bot to post notifications into the team chat or securing the cloud platform pipeline.

3. AGILE TESTING PRACTICES

3.1 Scrum

Scrum is an Agile framework for software development management. The idea of Scrum is to break development work down into defined blocks that can be completed within a fixed time frame, a *sprint* or a *cycle*, usually from 1 – 8 weeks. The development work is iterative, each cycle adding a new feature or functionality to the program. Completed features are evaluated at the end of each sprint and a list of next week's development tasks is assigned. Agile scrum teams collaborate daily and share information between team members.

The Scrum ideology focuses on change. Customer requirements change frequently, and customer's do not always know exactly what they want. Change requires Scrum teams to be iterative and incremental, to build modularly. Continuous improvement is one of the key values built into Scrum.

In Scrum team, three roles are defined: the development team, the Scrum Master and the Product Owner. The Product Owner is responsible for the success of the product that the team will build. The Product Owner sets the priorities for the features that the team will be implementing and works together with other stakeholders to derive them. (Gärtner 2013, p. 7). The Scrum Master role is to keep consensus between team members and help them to stay focused on the set tasks. The development team's role is to accomplish their tasks within the Scrum cycle. The roles and their definitions are listed in Table 10.

Table 10. *Scrum team roles*

Role	*Tasks*
Product Owner	Sets product requirements and responsible for product success
Scrum Master	Coaches the development team to succeed and responsible for the process
Development team	Complete development tasks within sprint

3.1.1 The Product Owner

The Product Owner is responsible for managing the product backlog and ensuring the value of the work team performs (Schwaber 2009). The Product Owner is visible to everyone and communicates to many stakeholders. The Product Owner leads the development effort to create a product that generates the desired benefits (Pichler 2010, p. 2).

3.1.2 The Scrum Master

The Scrum Master is often described as the “coach” of the development team that helps the development team to succeed. While the Product Owner is responsible for the developed *product*, the Scrum Master is responsible for the development *process* (Cohn 2017). The Scrum Master tries to find ways to render the development process more efficient, i.e. find ways to lower development barriers, facilitate meetings with stakeholders, reduce resource waste and lower risk.

3.1.3 The development team

The development team in Scrum does not feature traditional software development roles, such as programmers, designers, business analysts or testers, but more expansive, cross-functional developer roles. The team members value Agile principles and collaboration and harbor a “whole team” approach. Their goal is to complete development tasks defined by the Product Owner, using process defined by the Scrum Master, within the given sprint. A typical Scrum team consists of five to nine people (Cohn 2017). Scale in Scrum is achieved by increasing the number of Scrum teams, not the team member size.

3.2 Kanban

In traditional context, “Kanban” refers to scheduling systems for manufacturing processes that follow *lean* and *just-time-time* (JIT) principles. The Kanban system focuses on the manufacturing process and its flow, and to minimize the amount of work in progress. It was originally developed by the Japanese automobile manufacturer Toyota in the 1940’s. The literal meaning of the word “Kanban” in Japanese is “signboard”, “visual signal” or “card”, referring to the method of using physical cards to signal work steps and their phases (Leankit 2017). The four main principles of the Kanban system can be listed as

- visualize work flow
- limit the amount of work in progress
- focus on the flow
- continuous improvement.

Today, in the context of Agile knowledge work, “Kanban board” refers to a work visualizing tool for demonstrating process phases and task completion. In practice, Agile teams use a physical board with notes or a shared web-based solution as their Kanban board tool. An example of modern Agile team’s Kanban board is illustrated in Figure 5.

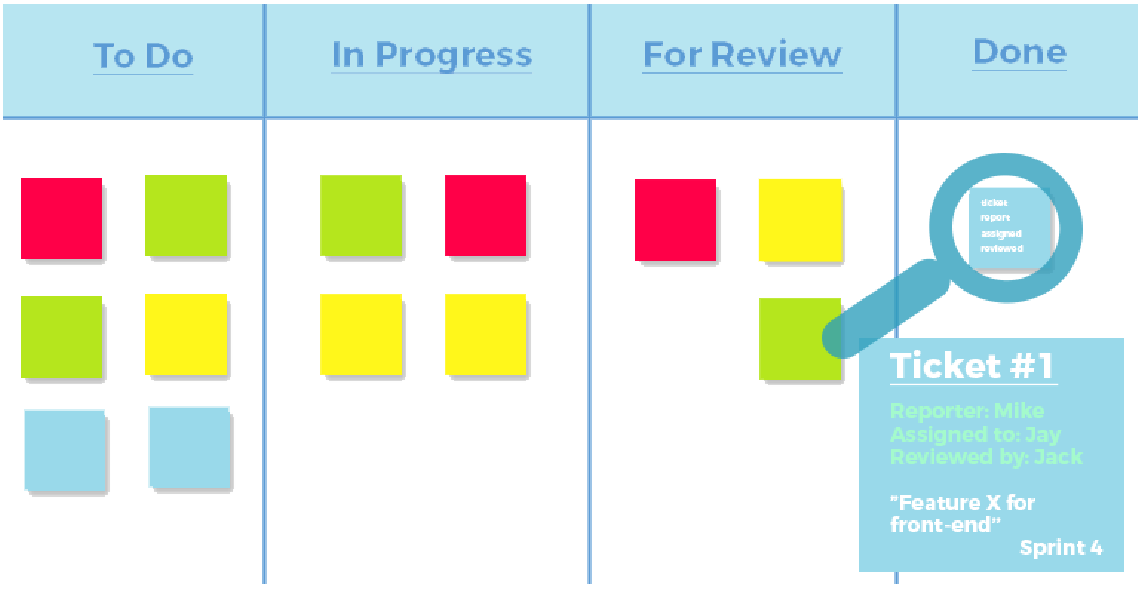

Figure 5. *An example of a modern Kanban board visualization*

The progress of the tasks is visualized through the incremental movement of the tasks on the board from left to right, from start to finish. Tasks are represented by small cards, which can be differentiated by color to mark different types of tasks. The x-axis represents the completion of tasks in one sprint. The board is divided into phases, which commonly include four phases: backlog, in progress, review, and done.

All the sprint tasks start in the "backlog" and once they are assigned to developers, they will be moved to the second phase, "in progress". After the developer has completed the assigned task, it is moved to the "review" phase. In this phase, the other team members review the completed work. If the completed task quality is deemed unsatisfactory, it can be moved back to the "in progress" phase for next sprint rework. If it passes the peer-review, it is moved to the rightmost side of the board, to the "done" or "completed" phase.

The purpose of Agile Kanban boards is the same as its traditional counterparts, to maximize process flow. The Kanban board works well in unison with Agile practices, such as Scrum (described in Chapter 3.1). It limits the amount of work in progress, as each developer is assigned only one or two features per sprint to complete, which forces the Product Owner to prioritize product features. It visualizes, how far features are from completion and what is their current status of development.

Other important Agile value, continuous improvement, is also built-in to the Kanban board system. The review phase ensures that all completed features are of required quality. Peer-review by other team members forces the team members to work cross-domain and to take interest in each other work and areas of expertise. Found defects or problems are shared by the team members and the whole team approach encourages shared responsibility of quality for the whole team.

Common tools for Kanban board visualization include the use of whiteboard and notes and software solutions. Modern issue tracking systems (ITS) include the option to visualize tasks or issues with a Kanban board. List of common tools in Table 11 below.

Table 11. *Common Kanban visualization tools*

Tool	*Website*
Trello	https://www.atlassian.com/software/trello
Jira	https://www.atlassian.com/software/jira
Kanban Flow	https://kanbanflow.com/
Kanboard	https://kanboard.net/

3.3 Defect Tracking System (DTS)

Defect Tracking System (DTS), or a defect management tool (DMT), is utilized to register, handle and manage found defects in a systematic manner (TMap Next 2006, p. 432). The Defect Tracking System contains a database that stores information about defects, such as detection date, author, priority and status. In general, it can be classified as a type of issue tracking system (ITS). The Defect Tracking System follows the found defect from its detection to its resolution.

Depending on the length and scope of the sprint, the number of found defects can be voluminous and the number increases with every sprint. This large amount of found defect data has to be efficiently stored, shared and updated among Agile team members. Modern Agile teams use web-based Defect Tracking Systems to report, document and share found defects. Modern tools commonly allow importing or exporting data between different systems, such as automatically updating test run results from a CI server to the Defect Tracking System.

Modern tools integrate many different features to their software, such as task management, document collaboration, code management or test run and test case documentation. A common DTS tool Jira, developed by Atlassian, integrates issue tracking with test case and test run management with their other testing tool, TestRail. Test cases and test execution steps are specified in TestRail and organized in multiple different test runs. The test run results will be show the percentage of failed test cases and link directly to their failed execution steps.

List of common Defect Tracking Systems is in Table 12.

Table 12. *List of common Defect Tracking Systems*

Tool	*Website*
Application Lifecycle Management (ALM)	https://software.microfocus.com/en-us/software/application-lifecycle-management
IBM Rational Quality Manager	http://www-03.ibm.com/software/products/fi/ratiqualmana
TestRail	http://www.gurock.com/testrail/
Jira	https://www.atlassian.com/software/jira
Mantis Bug Tracker	https://www.mantisbt.org/
Axosoft	https://www.axosoft.com/
FogBugz	http://www.fogcreek.com/fogbugz

3.4 Extreme Programming (XP)

Extreme Programming includes several features and practices. The methodology was first documented by Kent Beck in 1999 in his book "Extreme Programming Explained". In the same year, the rules of XP were posted by Don Wells at extremeprogramming.org, where they can still be viewed in 2018.

Typically, Extreme Programming is characterized as technically oriented, containing frequent releases and short development cycles in test driven environment. In practice, this means the extensive use of Pair Programming, code review, unit testing, few features, flat management and fitness for changes in customer requirements. Coding style was preferred simple and easily understandable – the acronym KISS, Keep It Simple Stupid – is frequently deployed in Extreme Programming. Phases of an Extreme Programming Project are illustrated in Figure 6.

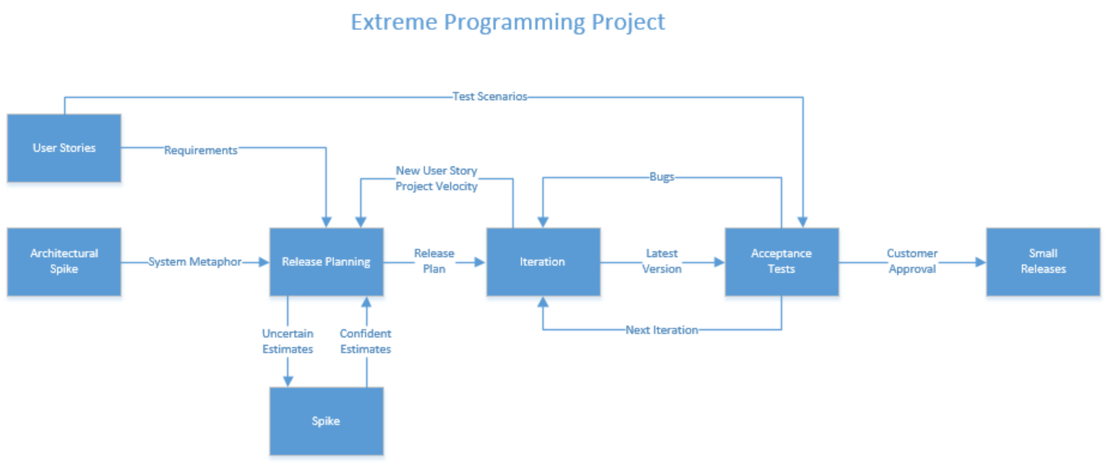

Figure 6. *Extreme Programming Project (Wells 2000)*

The Extreme Programming Project transitions from release planning to iteration to acceptance tests and small releases. User stories provide requirements for the release planning and they are later utilized as test scenarios for acceptance tests. System metaphor refers to a simplification of the system that can be easily described to gain an understanding of the system under test. Spike solutions, meaning writing simple programs to explore the program pathway space, are utilized in architectural and release planning. After each iteration, new user stories are taken under work and project velocity – the speed of development task completion – is measured. (Wells 2000)

3.4.1 Rules

According to Wells, there are 29 rules to Extreme Programming, that are categorized in five areas: planning, managing, coding, designing and testing. The rules and their implications are listed in Table 13 below.

Table 13. *Extreme Programming rules (Wells 2000)*

Category	*Planning*	*Managing*	*Coding*	*Designing*	*Testing*
Rules	User stories are utilized.	Dedicated, open work space	The customer is always available	Keep it simple stupid, KISS	All code must have unit testing
	Release schedule follows release planning	Sustainable work pace	Code agrees to standards	System metaphor	All code must pass unit tests before release

Frequent, small releases	Daily stand-up meeting	Code unit test first	Class, Responsibility and Collaboration (CRC) cards	A test is created for a detected bug
Work is divided into iterations	Assess development velocity	All production code is pair programmed	Create Spike solutions	Acceptance tests are run frequently
Iteration planning on every iteration	Move team members between roles	Only one pair integrates code at the time	No function added early	
	Adapt XP to changes and needs	Integrate often		
		Dedicated integration computer		
		Collective ownership		

3.4.2 Pair Programming

Pair programming is typical element of Extreme Programming. Two developers work side by side, co-create and co-design program code and architecture. Two developers have roles that are clearly defined. The first developer operates the keyboard and writes the program code, while the second developer evaluates and checks the code written. The second developer tries to think ahead of other developer's writing speed and to identify possible upcoming challenges.

TMap Next (2006, p. 342) lists several advantages to Pair Programming, such as

- many typing errors are caught while typing
- number of defects in the final product is lower
- technical design is better
- lower number of lines of code (LoC)

- problem solving is faster
- team members enjoy work more
- more team members gain understanding of the SUT
- team members learn considerably more about the SUT

3.5 Specification by Example (SbE)

There are two popular models that follow the process of Specification by Example, as described by Gojko Adzic in his nominal work “Specification by Example” (2011). These models are *the acceptance testing-centric model* and *the system behavior-centric model.* Acceptance Test Driven Development focuses on the automation of tests as part of the Specification by Example process. The system behavior centric model, often referred to as Behavior Driven Development, focuses on the process of specifying scenarios of the system behavior.

The costs of implementing Specification by Example can usually be justified to the management on the basis of avoiding late acceptance testing (SbE, p. 46). Specification by Example shortens the acceptance test phase, allowing the software to go into production two months earlier compared to traditional testing. Specifications are key to validating tests, and as such, in need of tight scrutiny. There are number of qualities to impose on specifications, such as

- Specifications should be focused and self-explanatory
- Examples should be precise and testable
- Specifications should be about business functionality, not software design
- Specifications should be in domain language
- Avoid writing specifications that are tightly coupled with code
- Don’t create flow like descriptions
- Scripts are not specifications, (SbE, Chapter 8).

3.6 Behavior Driven Development (BDD)

Behavior Driven Development (BDD) starts with specifying the wanted software behavior and then programming the implementation that produces the desired software behavior. BDD is often considered to be an extension to Test Driven Development. BDD provides a way to achieve modularity in the software development process.

3.7 Acceptance Test Driven development (ATDD)

Acceptance Test Driven Development (ATDD) aims for collaboration of business customers, developers and testers in producing testable product requirements and to build high quality software in a more rapid way (Gärtner 2013). The key point of ATDD is that each part of the program must pass an acceptance test before being merged into the master branch. Gärtner (2013) argues, that ATDD can be used in conjunction with Test Driven Development, using acceptance tests as bases for feature development.

3.8 Exploratory Testing (ET)

As most tests are automated, testers are free to use Exploratory Testing (ET) to find yet unknown defects or behavior in the system. According to Crispin (2009, p. 194), Exploratory Testing combines learning, test design and test execution into one test approach. Testers apply heuristics, sophisticated guesses and prior knowledge to implicate common problem areas. As the system under test is more familiar and well-understood, Exploratory Testing results improve.

Test automation can assist Exploratory Testing. Test automation tools should be used for automating test setup, generating test data and executing repetitive tasks (Crispin 2009, p. 201). Exploratory Testing can then immediately continue from the interesting point in the program, which saves the team a lot of time and redundant work. Many of the most difficult to detect defects or anomalies are hidden far down the program's path of execution, such as memory leaks and crashes.

3.8.1 Characteristics

Hagar argues in *Agile Testing* (2009, p. 199) that Exploratory Testing should be based on factors including risk analysis, a model of software behavior, past experience and developer opinion. Hagar lists five key characteristics useful in Exploratory Testing: test design, careful observation, critical thinking, diverse ideas and rich resources. Characteristics and their descriptions are listed in detail in Table 14.

Table 14. *Exploratory Testing (ET) characteristics and their descriptions (Agile Testing 2009, p. 199)*

Characteristic	*Description*
Test design	A good test designer understands many test methods. Thinking of multiple ways to approach the test, is one of the most important aspects of Exploratory Testing –

	the ability to quickly change and to adapt testing methods.
Careful observation	Exploratory testers are pedantic observers. They can identify subtle changes or unusual patterns in the program.
Critical thinking	Testers ability to quickly assess and to evaluate program behavior. A good tester is able to quickly direct the program execution to desired problem areas, when they are noticed.
Diverse ideas	Testing experience and subject matter expertise produce more accurate exploratory guesses. Shared expertise helps to create joint understanding of explored system.
Rich resources	Exploratory tester has a large set of tools, techniques, test data, colleagues and information sources to draw inspiration and support from.

3.8.2 Levels

Gregory & Crispin (2015, p. 188) mention Exploratory Testing as an integral part of Agile testing practices. Exploratory Testing is introduced in the Agile testing Quadrant 3 (Chapter 2.3.3.). It explores the workflow to evaluate if the anticipated business value has been delivered. Gregory & Crispin (2015) identify four product levels of Exploratory Testing:

1. Task level
2. Story level
3. Feature level
4. Product release level

Task level exploring happens during programming. Testers explore, how the program acts with different input parameters and try to find ways to cause errors. Examples of task level exploring include testing API inputs and responses or checking boundary values for exceptions.

Story level exploring is executed after a user story passes the expected results and automated tests. Story level testing should focus on development risks, boundary conditions and exceptions, detailed functionality issues and possible program states. Examples include asking for more details after suspecting an unspecified program pathway.

Feature level exploring can begin afters all the user stories are finished and the feature is considered complete. Exploration should focus on interactions with other parts of program and other systems. Features should be explored by multiple team members to foster the use of different test approaches. Examples include asking multiple opinions about usability of the expected feature.

Product release level exploring happens during integrated product delivery or release candidate delivery. Explorative testing should focus on high-risk system workflows, system and environment dependencies and system performance. Examples include testing web-based game on multiple different mobile phone devices and platforms.

3.9 Daily stand-up

Daily stand-up is an Agile practice where the team helds a daily meeting to discuss their work, usually standing up. This provides a moment for each team member to share what they have accomplished, what they are going to accomplish and where they need help accomplishing something. Daily meetings help to keep the team focused and they also enourage knowledge-sharing between team members.

The daily stand-up is usually kept brief. They typically last under 15 minutes, but they can be much longer if the situation demands it. Each team member is given a turn to present their accomplishments and what they are going to do next. Any challenges or troubles that might have come up, should be shared among the team. Through team collaboration, the obstacle should be less difficult to overcome.

The key point is sharing, what every team member is planning to accomplish during the day. Agile teams focus on three time horizons: day, iteration, release (Cohn 2006, p. 29). The daily stand-up focuses on the shortest horizon, a single day. Number of daily tasks should be an evaluation that can realistically be accomplished. The team also discusses the iteration schedule and what how the daily tasks help to accomplish the next iteration requirements. The release schedule is also kept in mind and assessed, what kind of large units are yet to be delivered or architectural changes to be adapted to.

3.10 User stories

User story is a high-level user or business requirement, commonly used in Agile software development. Typically, it consists of one or more sentences in the everyday or business language capturing what functionality a user needs. This also includes non-functional criteria and acceptance criteria. (Ottosen 2016, p. 10)

User stories combine strengths of both verbal and written communication. They provide (light) documentation but their purpose is to encourage discussion about product features and functionality. They get the customer or user side engaged with the product development process and foster mutual understanding of the product.

User stories are independent of each other. They can be modified, extended or swapped without affecting stories or other specifications. That also makes them a great tool for Agile, iterative development. User stories are often updated or created, when a new use case or functionality is introduced. (Ottosen 2016, p. 10)

Table 15. *User story following Role-Action-Result syntax (Ottosen 2016)*

1. As a <role>
2. I need <action>
3. so that <result>

User stories typically follow a certain syntax, such as Role-Action-Result or Given-When-Then presented in Table 15. This gives the user stories a definite length and scope, centered around a definite feature. The user story is co-designed by the developer and the user and the solution is also a mutual agreement. Describing the user actions in detail is preferred. User stories also have a defined acceptance criteria. Only after acceptance criteria is met, can user story be discarded.

Story requirements should include all necessary information needed for test execution. *SMART* requirements state that that requirements should be Specific, Measurable, Acceptable, Relevant and Time-specific. Requirements should be written in high-level language. (Ottosen 2016, p. 10)

Story champion is a term used by teams to describe a particular developer who stays with the story until it’s completed. The story champion will act as a point of contact for that user story and will answer all the issues regarding it. This ensures the efficient transfer of knowledge while switching pairs of developers working on the user story. This role is sometimes referred to as “story sponsor” in other corporations. (Adzic 2011, p. 55)

3.11 Virtual test environment

Virtual machines have been used for test environment versioning and management since the 1990s. Modern solution is to use containers, such as Docker, Kubernetes, Ansible or Puppet, to easily track and maintain test environments. Using a virtual test environment brings more control into the test environment management and over the software requirements.

Containers are light virtual environments running the needed software environment. Multiple containers can be easily run on the same server, as the live containers share the same kernel. The containers are spun up from images, compiled packages of required software for the test environment. The image installs appropriate software and libraries according to its yaml-file. This allows the test environment to always download the approariate software requirements, patches and libraries of apprioriate versions to ensure working condition.

Usually virtual test environments are initialized by the CI-server. The CI-server cleans the old working space, starts the build process, runs the tests in desired virtual test environment and saves results to the new working space. Test automation engineers write scripts needed to automate the test environment creation process.

List of common tools for virtual environments and containerization in Table 16.

Table 16. *List of common tools for virtual test environment*

Tool	*Website*
Docker	https://www.docker.com/
Kubernetes	https://kubernetes.io/
Ansible	https://www.ansible.com/
Puppet	https://puppet.com

3.12 Continuous Integration (CI)

Continuous Integration (CI) means to continuously integrate source code changes into a new software build (Gärtner 2013, p. 8). After a committed change to the working branch, the test automation server starts a virtual test environment, builds the working branch executable and runs all the unit and acceptance tests and displays information about the results (Gärtner 2013, p. 9). Continuous Integration is considered crucial to Agile development and Agile testing as it provides continuous assurance of the software state and quality and deployablity into production.

Currently used popular commercially licensed CI tools include Bamboo, Go, TeamCity and open-source ones include Jenkins, Hudson CI and CruiseControl (Swartout 2014, p. 88). The test automation developer automates the CI software pipeline using bash-scripts, os-operations or other tools. This often requires building software wrappers or to frameworks to achieve software-to-software interoperability.

Continuous Integration is prerequisite for Continuous Delivery and as such, a prerequisite for an Agile testing process. Swartout (2014, p. 44) defines Continuous Integration as “a method of delivering fully working and tested software in small incremental chunks to the production platform” and that it does not mean delivering large portions of the code infrequently. DevOps, meaning *development* and *operations*, is a term that is used often in conjuction with Continuous Integration, as CI is used as a tool for DevOps. DevOps should be understood as a broader term for collaboration between developers and business operations, such as automating a business function to i.e. aligning operations with business, to achieve a common goal.

List of common tools for Continuous Integration tools in Table 17.

Table 17. *List of common tools for Continuous Integration*

Tool	*Website*
Jenkins	https://jenkins-ci.org/
Hudson CI	http://hudson-ci.org/
Cruise Control	http://cruisecontrol.sourceforge.net/
Bamboo	https://fi.atlassian.com/software/bamboo
GoCD	https://www.gocd.org/
Team City	https://www.jetbrains.com/teamcity/
Travis CI	https://travis-ci.org/

4. RESEARCH RESULTS

4.1 Research strategy

4.1.1 Systematic literature review

A systematic literature review process aims to “identify, critically evaluate and integrate all the findings of relevant, high-quality studies addressing the research question” (Siddaway 2014). A systematic review answers “a defined research question by collecting and summarizing empirical evidence that fits pre-specified eligibility criteria” (University of Edinburgh 2017). Systematic reviews are characterized as being objective, systematic, transparent and replicable (Siddaway 2014).

A systematic literature review compiles published research on a topic. A thorough, systematic search of research literature is required to ensure that the most relevant studies are used to reduce bias in the review process (CRD 2009, p. 16). The style of the review can be argumentative, integrative, historical, methodological, systematic or theoretical (NCBI 2016).

This systematic literature review aims to identify, what test automation models, Agile practices and tools are utilized in researched literature and which characteristics are the most preferred. The identified practices and characteristics are then classified into different categories. Lastly, the preferred practices are formulated into a synthesized generic test automation model.

4.1.2 Fink’s systematic literature review model

Systematic literature review models are used to provide guidelines for research and help to limit the amount of information to be searched. Literature review models define a process path for research that can be confirmed and replicated. This research utilizes a systematic literature review model proposed by Fink (2014). Fink’s literature review model is based on seven tasks, which are listed as:

1. Select research questions
2. Select bibliographic or article databases
3. Choose search terms
4. Apply practical screening criteria
5. Apply methodological screening criteria
6. Do the review
7. Synthesize results

This research was conducted in the following steps. First, the research questions were defined (Chapter 1.3). Second, the bibliographics databases for research where selected (Chapter 4.1.3). Third, the search terms were formulated (Chapter 4.1.4). Fourth, the practical screening criteria was applied (Chapter 4.1.5). Fifth, the methodological screening criteria was applied (Chapter 4.1.5). Sixth, the review of the selected articles was executed (Chapter 4.2). Seventh, the synthesized results and model are presented (Chapter 5).

4.1.3 Bibliographic databases

In this research, bibliographic databases are the primary source of research literature. The term "bibliographic database" has traditionally been defined as abstracting and indexing services for the scholarly literature (Trawick et al. 2003). More recent definitions refer to bibliographic databases as any large collection of indexed text documents, such as, web-based subscription academic journal services (Kusserow et al. 2014, p. 1). Books about the subject matter were used as background research and to gain insight into the subject matter before conducting the research.

The research was performed using search engine Andor. Andor is the name of the in-house search engine developed at the Tampere University of Technology Library (TUT 2017). Andor functions as a web-portal that amasses links to multiple different bibliographic databases. The databases utilized through Andor were Scopus, ScienceDirect, SpringerLink and IEEE Xplore. The utilized bibliographic databases and their descriptions are listed in Table 18.

The following books about the subject matter, that were recommended by my colleagues, were used as secondary reference and as background research into the subject matter:

1. Agile Estimating & Planning (2006), Mike Cohn, Pearson Education, 330 p.
2. Agile Testing: a Practical guide for testers and Agile teams (2009), Lisa Crispin & Janet Gregory, Addison-Wesley, 533 p.
3. More Agile Testing: Learning journeys for whole teams (2015), Janet Gregory & Lisa Crispin, Addison-Wesley, 486 p.
4. ATTD By Example (2013), Markus Gärtner, Addison-Wesley, 211 p.
5. Agile Product Management with SCRUM (2010), Roman Pichler, Addison-Wesley, 133 p.
6. Specification by Example (2011), Gojko Adzic, Manning Publications, 296 p.
7. TMap Next: for result-driven testing (2006), Sogeti Nederland B. V., 752 p.

Table 18. *List of utilized biobliographic databases and their descriptions*

Name	*Description*
Scopus	According to publisher Elsevier (2017), Scopus is “the largest abstract and citation database of peer-reviewed literature: scientific journals, books and conference proceedings” and contains over 22 000 peer-reviewed journals and over 69 million records.
ScienceDirect	According to publisher Elsevier (2017), ScienceDirect is “the world's leading source for scientific, technical, and medical research” and currently contains over 14,2 million items.
SpringerLink	According to the publisher Springer Publishing (2017), SpringerLink “provides researchers with access to millions of scientific documents from journals, books, series, protocols and reference works” and currently contains in total over 11 million scientific documents with over 6,2 million scientific articles.
IEEE Xplore	IEEE Xplore is the research database of IEEE, which publishes “the leading journals, transactions, letters, and magazines in electrical engineering, computing, biotechnology, telecommunications, power and energy, and dozens of other technologies” and currently contains over 4,4 million items (IEEE 2017).

4.1.4 Search terms

Selecting purposeful search terms are critical for research validity. Search term refers to the words or phrases utilized in the search query (Chandler & Munday 2016). The bibliographic database search engine locates the relevant content according to the recognized keywords in search terms. The selected search engine Andor supports Boolean operators, such as “AND“ and “OR” to condition the searched keywords (TUT 2017).

The selected search terms were formulated from the research questions (Chapter 1.3). Boolean operator “AND” was used to join singular search terms into purposeful search terms. After carefully examining the research questions, the following search terms were constructed:

```
“test automation”
“software test automation”
“agile testing”
“agile testing” AND “test automation”
“agile” AND “test automation”
“agile” AND “testing” AND “framework”
```

The selected search terms were input into the Andor search engine. The Andor seach engine was restricted to search for “scientific articles and journals” and to rate search results according to “relevance”. The search was conducted in two parts, in 15.09.2017 and 4.11.2017. The number of search results for each search are displayed in Table 19.

Table 19. *Number of Andor search results*

Search term	***Number of Andor results***
“test automation”	5586
“software test automation”	358
“agile testing”	311
“agile testing” AND “test automation”	8
“agile” AND “test automation”	2915
“agile” AND “testing” AND “framework”	8367

4.1.5 Practical and methodological inclusion and exclusion criteria

The initial searches yielded a large quantity of search results, over 15 000 articles and journals. A screening process was applied to reduce the number of irrelevant articles and to find the most meaningful articles to the research questions. A practical and methodological screening were applied search results.

Following Fink's (2014) systematic literature review model, the screening process includes two phases: practical and methodological inclusion and exclusion criteria. Practical inclusion and exclusion criteria refers to attributes that can be screened practically and are essential to be included. Practical criteria covers a wider range of articles in the research topic and is used to identify relevant areas of interest. Methodological inclusion and exclusion criteria follow the guidelines of the chosen research method to uncover relevant research results. Methodological inclusion and exclusion criteria are applied to improve search quality and to reduce the number of unsuitable results.

For this search, the following practical inclusion and exclusion criteria were applied:

Inclusion criteria:

- Language: *English*
- Publication date: *Released after the year 2000*
- Publication type: *Scientific articles or journals*
- Abstract: *Mentions of test automation, Agile testing or framework*
- Relevance: *Within the first ten search results*
- Availability: *Full-text available*

Exclusion criteria:

- Language: *Not in English*
- Availability: *Full-text non-available*

This resulted in narrowing the search results into fifty relevant articles. A full list of the articles is available in Appendix A. After reading and analyzing the articles, the following methodological inclusion and exclusion criteria were applied:

Inclusion criteria:

- The article contains a proposal for applying test automation in Agile testing environment and answers research questions
- A case study using the proposal was performed

Exclusion criteria:

- The article does not propose or present a test automation model or framework
- The article does not follow scientific guidelines or research standards
- Only one article per author(s)

The methodological screening was focused in finding the most relevant articles to the research questions that presented a model for test automation in Agile testing environment with a case study performed using the model. The screened articles were also to contain mentions of test automation tools, QA organizational structures, used programming languages, frameworks and examples of their application.

Articles containing test automation proposals but with no clear mention of Agile methodologies or organizational or managerial practices were dismissed as out of scope. Conference papers and bibliographical indexes were also excluded. Applying this criterion resulted in narrowing the list of fifty articles into ten relevant articles. The articles and their proposed models are reviewed in Chapters 4.2.1 – 4.2.10.

4.1.6 Prescriptive modelling

The selected ten articles and their proposals were reviewed, modelled and characterized. The proposed process frameworks were modelled using Microsoft Visio in a unified manner to help comparison with various different presentations and characterized and presented using *prescriptive modelling* proposed by Acuna et al. (2001). As the proposed models and their descriptions by their authors varied immensily in detail, depth and complexity, converting the model attributes into the characterization matrix and comparing them proved difficult.

Prescriptive models define the recommended or required means of executing the software development process and answer the question “how should software be developed?”. Prescriptive models can be divided in to two categories: manual and automated prescriptive models. Manual presctipive models can be standards and methodologies centered on management, development, evaluation and software life cycle and organisational life cycle support processes, while automated prescriptive models refer to activities related to assistance, support and management of computer-assisted software production techniques. Prescriptive models belonging to manual modelling category include traditional structured methodologies, organizational design methodologies and software life cycle development standards. (Acuna et al. 2001)

In this review, the selected ten proposals were prescribed within a characterization matrix proposed by Acuna et al. (2001). The models are characterized in three different areas: **model criteria**, **representation criteria** and **methodological criteria**. The model criteria includes process elements represented by the model, such as agent, activity, artefact, role and event and process environments addressed by model, such as organizational, creative ability, social interaction, environment flexibility and scientific/technological environment. Creative ability, social interaction and environment flexibility are considered to be part of the organizational culture. Model criteria and their definitions are presented in Table 20.

***Table 20.** List of model criteria descriptions*

	Model criteria	*Definition*
Process elements represented by the model	Actor	Entity executing the process
	Role	Describes set of actor responsibilities
	Activity	Produces externally visible changes in the process
	Artefact	(Sub) product or raw material produced by process activity
	Event	A noteworthy occurrence happening in a specific moment
Process environments represented by the model	Organizational	Reflects model dealings with organisational issues, such as organizational culture, behavior, the design and evolution of the organization.
	Creative ability	The ability to develop new organizational and software process models
	Social interaction	Relations between different reasoning structures within the organization
	Environment flexibility	Organization's position on socio-cultural, scientific/technological environment and generic software process models

	Scientific/technological	Model points to tools, infrastructure and software used for software production

The representation criteria includes information perspectives, such as functional, behavioral, organizational and informative and notation characteristics from the viewpoint of information quality. Representation criteria and their definitions are presented in Table 21.

Table 21. *List of representation criteria descriptions*

	Representation criteria	*Definition*
Information perspectives	Functional	Represents process element implementation and their information entity flows
	Behavioral	Represents sequentially the conditions under which process elements are implemented
	Organizational	Represents where and by whom in the organization the process elements are implemented
	Informative	Represents information entities output or manipulation in the process, including structure and relationships
Notation characteristics	Informal	Information represented informally
	Formal	Information represented formally
	Automated	Information generated by automated systems
	Text	Information notated in text format
	Graphic	Information notated in graphic format

The methodological criteria describes the modelling procedure (non-developed or developed), procedure coverage (partial or suffient) and procedure definition (undefined, semi-defined or defined). Methodological criteria and their definitions in the context of this research are presented in Table 22.

Table 22. *List of methodological criteria descriptions*

	Methodological criteria	*Definition*
Modelling procedure	Non-developed	Does not show defined causal relationships between process elements
	Developed	Showing defined causal relationships between process elements
Procedure coverage	Partial	Covers a technical or organizational part of test automation process in Agile environment
	Sufficient	Covers both technical and organizational parts of the test automation process in Agile environment
Procedure defintion	Undefined	Procedures are not defined
	Semi-defined	Some procedures are defined
	Defined	All procedures are defined

In addition to prescriptive modelling, common domains of application, common Agile practices and common tools were identified and registered, if mentioned by name by the authors. This was done in effort to capture, what Agile practices and tools are used in the literature and which are the most popular. Agile practices and tools are listed at the end of each model review.

4.2 Data review

4.2.1 Lean Canvas Model (2017)

Nidagundi & Novickis (2017) propose "Lean Canvas Model" for Scrum software testing. They argue that Scrum software development framework delivers software in incremental and iterative way. They define Lean Canvas as "a white board with several blocks with title names and is used mainly for the evaluation of business ideas".

Lean Canvas life cycle starts with an idea and a phase of collecting ideas for product requirements. The life cycle model and metrics are used to measure progress and to determine when the project is considered done. The life cycle model applied to Scrum software testing can be written as a repeating loop of the following actions:

1) Ideas=Product backlog, sprint backlog,
2) Build=Development,
3) Product=Testing,
4) Measure=Sprint planning meetings,
5) Data=Test data, Burndown charts,
6) Lean=Retrospective meetings, sprint review.

Lean Canvas is based on three principles of 1. Creating a document of your plan, 2. Identification of waste process or parts of your plan, 3. Repetitive test cycles for your plan. The Lean Canvas process for software testing is presented in Figure 7. The model operates in customer-facing domain.

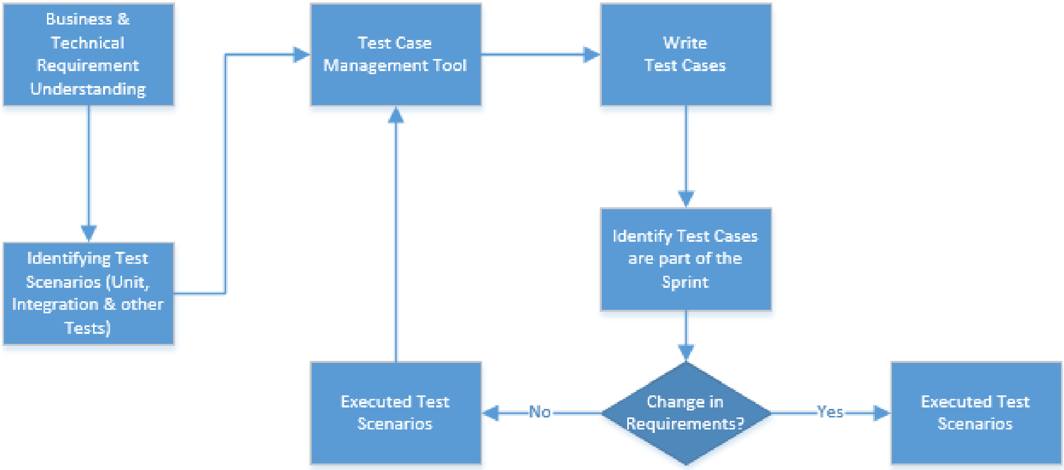
Business & Technical Requirement Understanding
Identifying Test Scenarios (Unit, Integration & other Tests)
Test Case Management Tool
Write Test Cases
Identify Test Cases are part of the Sprint
Change in Requirements?
No
Yes
Executed Test Scenarios
Executed Test Scenarios

***Figure** 7. Lean Canvas process for software testing (Nidagundi & Novickis 2017)*

The Lean Canvas Model starts with understanding business and technical requirements of the product. The second step is to identify test scenarios in unit and integration testing and through other tests. These test scenarios are managed by a test case management tool. With each sprint, test cases that are part of the sprint are identified. If product requirements have changed, different test cases are run.

In the context of prescriptive modelling and model criteria, process elements, such as the actor implementing the model and the role and responsibilities are not defined. The events (change in requirements) triggering activities (executing test scenarios) and producing artefacts (executed test scenarios) are defined. The model criteria does not present process environment elements, such as organisational, creative ability, social interaction, environment flexibility or scientific/biological environment. The representation criteria and information perspective can be defined as behavioral (presenting a sequential conditions for process element implementation). From the viewpoint of information quality, the information is informally presented in text and graphic form. In the methodological criteria, the procedure can be characterized as developed (showing defined causal relationships between procedure elements), covering a part (technical environment undefined) of the test automation process with undefined procedures (presented procedures are not defined on base-level, only containing top-level descriptions of procedure actions). Review of the model is presented in Table 23 and characterization matrix in Table 24.

Table 23. *Lean Canvas Model review*

Name	**Lean Canvas Model**
Authors	Nidagundi, P. & Novickis, L.
Published	2017
Domain	Customer-facing
Testing tools	N/A
Agile practices	Scrum, Test case management tool

Table 24. *Lean Canvas Model characterization matrix*

	Model Criteria										**Representation Criteria**									**Methodological Criteria**						
	Process elements represented by the model					Process environments represented by the model					Information perspectives				Notation characteristics					Modelling procedure		Procedure coverage		Procedure definition		
							cultural								from the viewpoint of information quality			from the viewpoint of formal notation								
	agent	activity	artefact	role	event	organizational	creative	social inter-	environ-	scientific/technological	fuctional	behavioral	ogranisational	informative	informal	formal	automated	text	graphic	non-develop	developed	partial	sufficient	undefined	semi-defined	defined
Lean Canvas Model		X	X		X							X			X			X	X		X	X		X		

4.2.2 Quality Experience (2016)

Prechelt et al. (2016) propose a "Quality Experience" model for delivering successful Agile projects without dedicated testers. Their research centered around the question: How does successful Agile development work without separate testers? Prechelt et al. performed three case studies based on Grounded Theory evaluation for interviews and direct observations. All three case study Agile development teams developed web-based portals for customer use.

Prechelt et al. (2016) studied three Agile teams, of which only one team had a dedicated tester role, while the two others shared the testing responsibilities between team members. The idea of doing Agile development without dedicated testers is not new; Extreme Programming (XP) has a dedicated role of tester, while Scrum clearly states that teams should be cross-functional with everyone having the same title of developer. Prechelt et al. (2016) consider Kanban to be agnostic of the separate-tester role issue.

In their research, Prechelt et al. (2016) define a term "Quality Experience" to denote a mode of quality assurance and deployment. Here, quality is a holistic attribute, which includes aspects from business value creation to operational tasks. According to Prechelt et al. (2016), a Quality Experience team

1. feels fully responsible for their quality of software
2. receives feedback about its quality
3. quickly, directly and realistically
4. while rapidly repairing found defects.

The Quality Experience process appoints six key areas of interest: conscious empowerment decision, the role of responsibility, the role of feedback, rapid repair of defects and motivation effects, that have the desired consequence of frequent deployment. The process model is based on the architectural precondition of modular software architecture. The software architecture must be sufficiently able to decouple from the work of one team from another, to enable work delegation and concurrent development.

The Quality Experience process starts with a concious decision to empower the development team to take control over the deployment and monitoring of the product. This requires the team to be cross-functional and capable and architecture to be modular. In this context, empowerment means assigning quality responsibilities to the developer role, which leads to increased feeling of responsibility to quality, both socially and psychologically. Automated tests provide constant feedback on the quality and the feedback is direct, realistic and quick. The defects are rapidly repaired as soon as their detected. This repeated, rapid cycle of development leads to developers having higher motivation and co-defining requirements with the team, as each developer feels shared responsibility for quality.

The Quality Experience process model is presented below in Figure 8. The model operates in customer-facing domain. The model begins with the feeling of empowerment to deploy, leading to feeling responsible and higher motivation and more frequent deployments. The blue lines mean engineering, red ones social and green ones psychological driving forces.

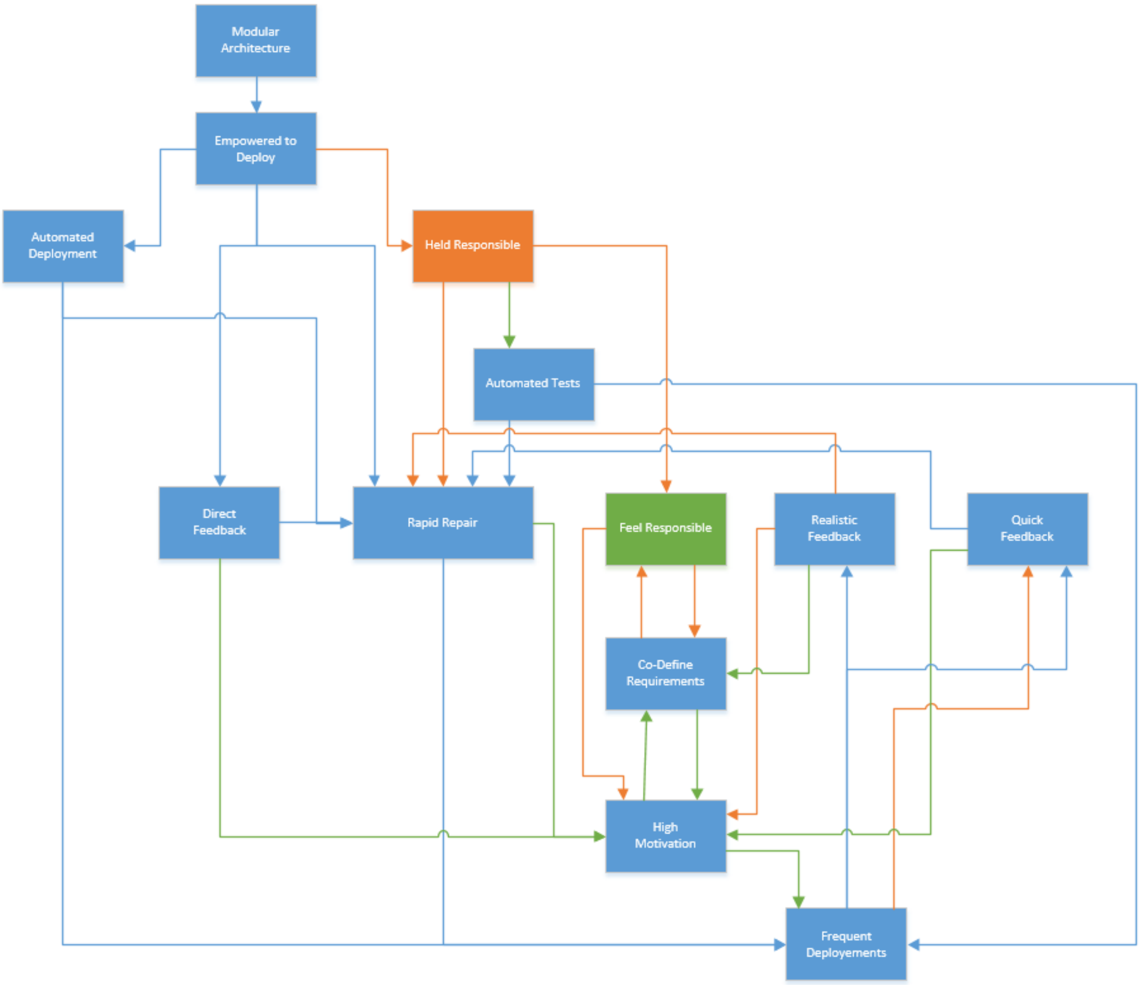

Figure 8. *Quality Experience process description (Prechelt et al. 2016)*

In the context of prescriptive modelling and model criteria, process elements, such as actor, activity, artefact, role and event are clearly defined. The developer (actor) is empowered to deploy (activity) when held responsible (role) and produces automated tests (artefact) which lead to quick feedback and rapid repair (event). The model criteria does not present process environment elements, such as organisational, creative ability, social interaction, environment flexibility or scientific/biological environment. The representation criteria and information perspective can be defined as functional (presenting information flows of process elements) and behavioral (presenting a sequential conditions for process element implementation). From the viewpoint of information quality, the information is formally presented in text and graphic form. In the methodological criteria, the procedure can be characterized as developed (showing defined causal relationships between procedure elements), covering a part (does not include technical environment) of the test automation process with defined procedures (each presented procedure is defined). Review of the model is presented in Table 25 and characterization matrix in Table 26.

Table 25. *Quality Experience review*

Name	**Quality Experience**
Authors	Prechelt, L., Schmeisky, H. & Zieris, F.
Published	2016
Domain	Customer-facing
Testing tools	N/A
Agile practices	Kanban, Scrum, XP, CI

Table 26. *Quality Experience characterization matrix*

	Model Criteria										**Representation Criteria**									**Methodological Criteria**						
	Process elements represented by the model					Process environments represented by the model					Information perspectives				Notation characteristics					Modelling procedure		Procedure coverage		Procedure definition		
							cultural								from the viewpoint of information quality			from the viewpoint of formal notation								
	agent	activity	artefact	role	event	organizational	creative	social inter-	environ-	scientific/technological	fuctional	behavioral	ograinsational	informative	informal	formal	automated	text	graphic	non-develop	developed	partial	sufficient	undefined	semi-defined	defined
Quality Experience	x	x	x	x	x						x	x				x		x	x		x	x				x

4.2.3 Agile Testing for Railway Safety-Critical Software (2016)

Li et al. (2016) present in their article “Towards Agile Testing for Railway Safety-Critical Software” a proposal for an Agile testing framework designed for safety-critical railway software. Their study focused on Chinese Train Control System (CTCS). During their study, they designed a test framework that includes a build automation tool that manages source code, unit tests, integration tests, resources and other tools and supports Continuous Integration and delivery. The model operates in safety-critical domain.

Li et al. (2016) identified three key challenges: lack of Continuous Integration (CI) and Continuous Delivery (CD), unit test generated manually and lack of integration testing. They proposed utilizing Continuous Integration and deployment, generating unit test coverage logic automatically and generating test paths from Model Based Testing (MBT).

When testing safety critical software, rigorous testing and stronger test coverage criteria in unit testing is required. Modified condition decision coverage (MCDC) should be used for unit testing, the same standard applied by the aviation industry. According to Li et al. safety critical software requirements usually change due to changes in detailed system design or faults detected by testing, not because of changes in requirements. They noted, that safety critical software is not generally more complex or have more lines of code (LOC). Safety critical software had more often clauses with four or more predicates than non-safety critical software.

The Agile test framework is depicted in Figure 9. The automation build tool is at the center of the framework. The automation build tool, such as Jenkins or Travis CI, manages source code, resources and dependencies and executes unit and integration tests. For

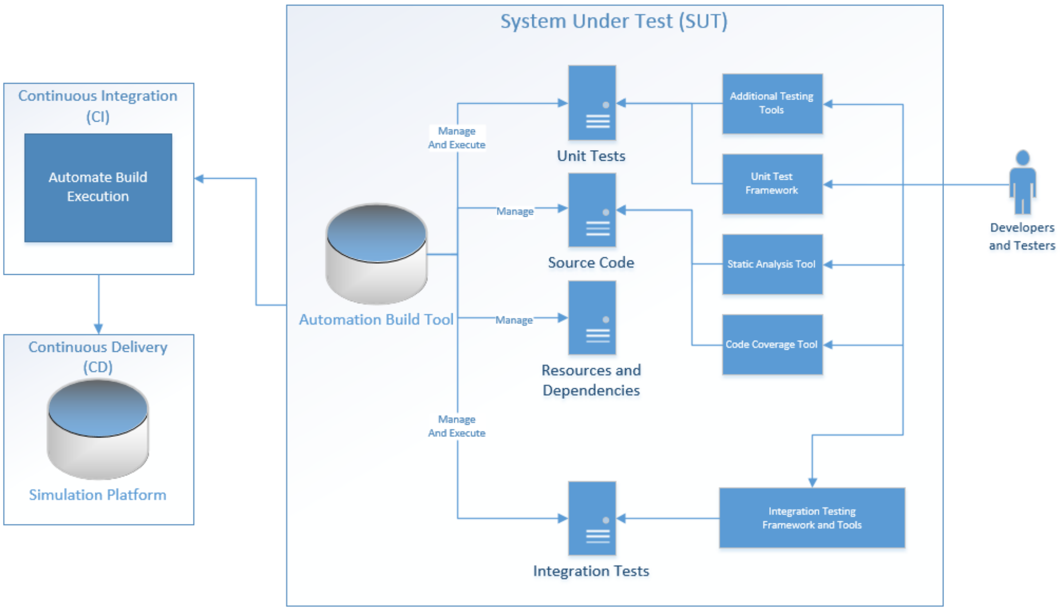

__Figure 9.__ Agile Testing for Railway Safety Critical Software (Li et al. 2016)

used example tools, Apache Maven was used for a unit testing framework, JaCoCO for a coverage measurement tool and CheckStyle for a static analysis tool.

After a new commit is made, source code is pulled from a repository, such as GitHub, and automatically build and executed. As deploying railway software automatically directly to production would not be safe or realistic, automatic builds and integration tests are run in a simulated platform. This simulated platform includes simulated train stations, railroad rails, blocks, etc. The simulated platform has the advantage that multiple different kind of station types can be simulated and more comprehensive test scenarios can be created. There is a constant continuous feedback loop between the Continuous Integration of source code to build and Continuous Delivery to the testing platform, from where the code could be deployed directly into production.

Li et al. (2016) utilized Model Based Testing (MBT) to generate test paths from the state machine UML descriptions. These test paths define the abstract, logical level tests of the SUT. To reduce the massive number of test cases generated from multiple different train track state possibilities, combinatorial testing was used. Combinatorial testing reduces the number of test cases significantly, while increasing the over all risk only slightly. The key insight comes from the observation that most defects are caused by interactions involving only a small number of parameters.

In the context of prescriptive modelling and model criteria, process elements, such as actor, activity, artefact and event are clearly defined. The developers and testers (actors) execute testing actitivites, such as create unit tests (activity) that automated build tool executes (artefact) when triggered by a new commit (event). The actor role responsibilities are not defined. The model criteria does not present process environment elements, such as organisational, creative ability, social interaction, environment flexibility or scientific/biological environment. The representation criteria and information perspective can be defined as functional (presenting information flows of process elements). From the viewpoint of information quality, the information is formally presented in text and graphic form. In the methodological criteria, the procedure can be characterized as developed (showing defined causal relationships between procedure elements), covering a part (does not include organizational environment) of the test automation process with defined procedures (presented procedures are defined in text). Review of the model is presented in Table 27 and characterization matrix in Table 28.

__Table 27.__ Agile Testing for Railway Safety-Critical Software review

Name	**Agile Testing for Railway Safety-Critical Software**
Authors	Li, N., Guo, J.a, Lei, J.b, Li, Y., Rao, C.a, Cao, Y.
Published	2016
Domain	Safety-critical
Testing tools	Jenkins, Travis CI, Apache Maven, JaCoCo, Checkstyle
Agile practices	CI, CD, MBT

__Table 28.__ Agile Testing for Railway Safety-Critical Software characterization matrix

	Model Criteria										**Representation Criteria**									**Methodological Criteria**						
	Process elements represented by the model					Process environments represented by the model					Information perspectives				Notation characteristics					Modelling procedure		Procedure coverage		Procedure definition		
							cultural								from the viewpoint of information quality			from the viewpoint of formal notation								
	agent	activity	artefact	role	event	organizational	creative	social inter-	environ-	scientific/technological	fuctional	behavioral	ogranisational	informative	informal	formal	automated	text	graphic	non-develop	developed	partial	sufficient	undefined	semi-defined	defined
Agile testing for railway	X	X	X		X						X					X		X	X		X	X				X

4.2.4 Development Method for Acceptance Test Process (2016)

Shim et al. (2016) propose a model “Acceptance Test Process” for acceptance test automation. They base their model on the Agile Testing Quadrants (described in more detail in Chapter 2.3). The process is modelled in Figure 10. The model operates in acceptance testing domain.

The primary purpose of acceptance tests is to determine, whether the product is ready for publication or not, not finding defects. Writers classify different test automation tools by the phase of usage. They also modelled automation design process and the practice of Separation of Concerns (SoC), where design decisions are separated into different layers. The top level layers include business and developer layers. Shim et al. (2016) list key quality attributes for acceptance test automation: readability, maintainability, traceability and accessibility. Readability concerns the automation specification and how easy it is to understand by developers and business staff and other stakeholders. Maintainability concerns, what is the effort required to maintain the system in working condition. Traceability concerns requirements and tests. Accessibility concerns how deep is the relationship between stakeholders and the collaboration between the developers.

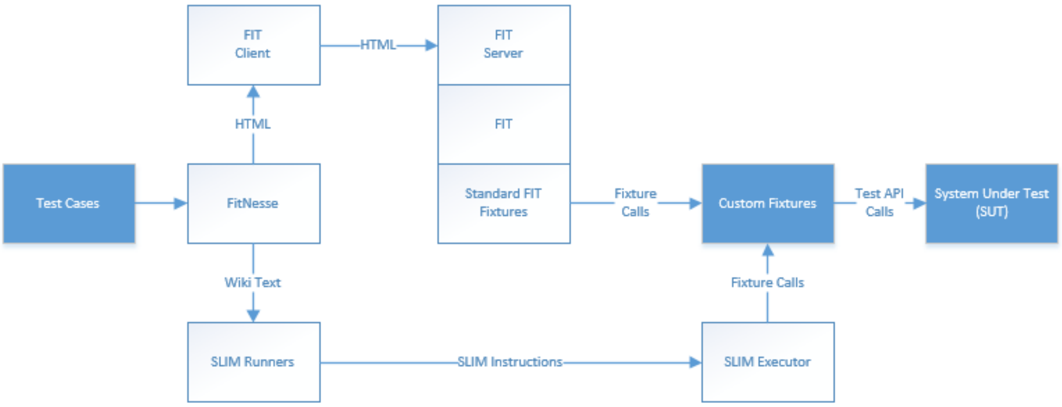

Figure 10. *Model for Acceptance Test Automation (Shim et al. 2016)*

The Acceptance Test Automation process is described in following steps:

1. Establishment of strategies
2. Derivation of requirements
3. Preparation of requirements specification and test cases
4. Construction of test automation

Writers applied test automation using FitNesse for the architecture design. FitNesse is a wiki-based testing tool that allows writing configuring test variables through web forms.

In the context of prescriptive modelling and model criteria, process elements, such as actor, activity, artefact, role and event are not defined. The model criteria does not present process environment elements, such as organisational, creative ability, social interaction, environment flexibility or scientific/biological environment. The representation criteria and information perspective can be defined as informative (presenting information entities output or manipulated by the process). From the viewpoint of information quality, the information is formally presented in text and graphic form. In the methodological criteria, the procedure can be characterized as developed (showing defined causal relationships between procedure elements), covering a part (does not include organizational environment) of the test automation process with undefined procedures (presented procedures are not defined). Review of the model is presented in Table 29 and characterization matrix in Table 30.

__Table 29.__ Acceptance Test Automation review

Name	**Acceptance Test Automation**
Authors	Shim, J.-A., Kwon, H.-J., Jung, H.-J., Hwang, M.-S.
Published	2016
Domain	Acceptance testing
Testing tools	FitNesse
Agile practices	SoC, SbE, ATDD

__Table 30.__ Acceptance Test Automation characterization matrix

	Model Criteria										**Representation Criteria**									**Methodological Criteria**						
	Process elements represented by the model					Process environments represented by the model					Information perspectives				Notation characteristics					Modelling procedure		Procedure coverage		Procedure definition		
						cultural									from the viewpoint of information quality			from the viewpoint of formal notation								
	agent	activity	artefact	role	event	organizational	creative	social inter-	environ-	scientific/technological	fuctional	behavioral	ogranisational	informative	informal	formal	automated	text	graphic	non-develop	developed	partial	sufficient	undefined	semi-defined	defined
Acceptance test automation					x									x		x		x	x		x	x		x		

4.2.5 Dynamic Regression, Code Bisector and Code Quality (2015)

Sivanandan (2015) proposes a model for enhancing Agile methodologies using Dynamic Regression, Code Bisector and Code Quality with Continuous Integration (CI). Sivanandan (2015) presents Dynamic Regression as a solution to running unit and functional automation for only the code where changes happened. Code Bisector is a tool for finding the broken piece of code faster. Sivanandan's (2015) last question centers around how to define quality of program code and which seven axes affect Code Quality.

Sivanandan (2015) claims that using the above mentioned three processes, a business group was able to increase their Return On Investment (ROI) by 70-80%. This translates to reduced time to finding defects, better code quality and delivering on time. Sivanandan (2015) claims that using these techniques, predictive software quality can be attained. All three models operate in customer-facing domain.

Dynamic Regression method begins by mapping source code against test suites. The second step is to run code coverage for particular test suites and identifying related classes which have been called during the execution and mapping them accordingly. Sivanandan (2015) describes this step as very time consuming, as all the (possibly thousands) of test cases have to be mapped accordingly. A "Smart Engine" is run every six hours to fech a list newly added files to the source control system. If any of source code files is touched or modified, the Smart Engine mapper parses the mapper database and pulls the corresponding test suite. The list of corresponding functional test suites are then automatically executed on the CI-server.

Code Bisector is defined as "a method for finding a code change that results in a specific behavior change". It is utilized to help developers find the defects more quickly and minimizing the manual trace down effort. Code Bisector is integrated into the CI to intimate delta change breakages to the development team. These breaking changes are then displayed on the quality dashboard with responsible developer.

Sivanandan (2015) also argues that Code Quality is built on seven axes, which are: sticking to coding standards and best practices, frequent use code comments in the source code (especially in public APIs), duplicate lines of code, code complexity among components, zero or low coverage by units tests (especially in complex parts of the program), unattending potential bugs, using complex design and architecture.

The process of using the three methods, Dynamic Regression, Code Bisector and Code Quality is represented in Figure 11. The process description uses tools such as Jenkins as the CI-server and Robot Framework as the acceptance test framework. The architecture of the test suite mapping database is left out.

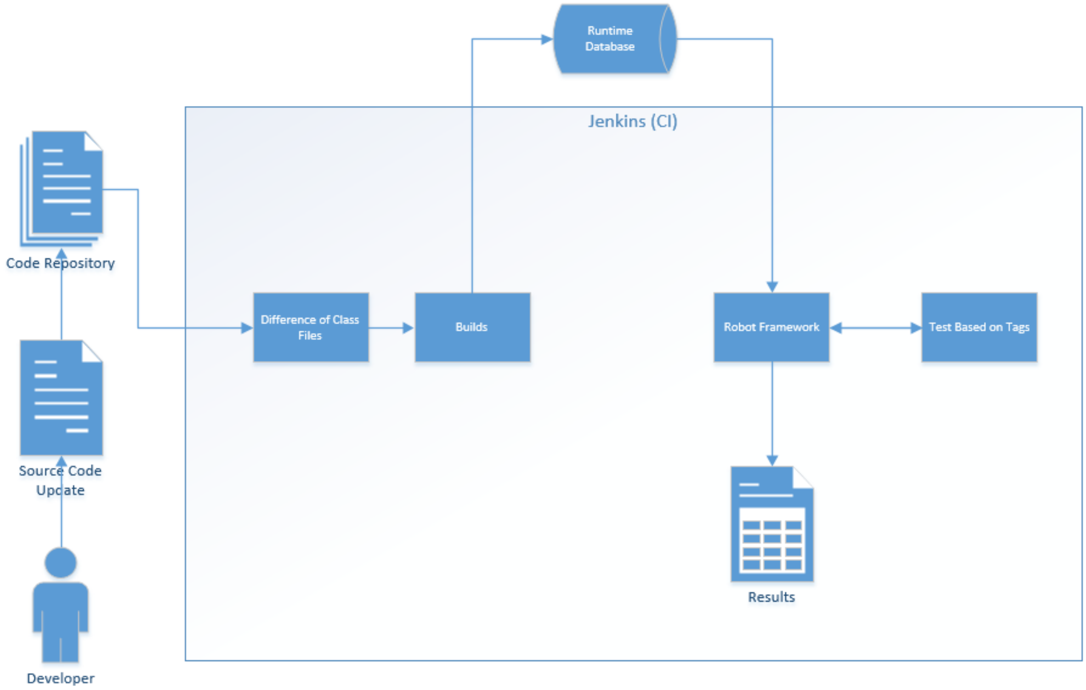

Figure 11. *Dynamic Regression, Code Bisector and Code Quality (Sivanandan 2015)*

In the context of prescriptive modelling and model criteria, process elements, such as actor, activity, artefact, role and are not defined. Process is modelled as sequential events. The model criteria does not present process environment elements, such as organisational, creative ability, social interaction, environment flexibility or scientific/biological environment. The representation criteria and information perspective can be defined as informative (presenting information entities output or manipulated by the process). From the viewpoint of information quality, the information is informally presented in graphic form. In the methodological criteria, the procedure can be characterized as developed (showing defined causal relationships between procedure elements), covering a part (does not include organizational environment) of the test automation process with semi-defined procedures (some presented procedures are defined). Review of the model is presented in Table 31 and characterization matrix in Table 32.

***Table 31.** N-Tiered Test Automation System review*

Name	**Dynamic Regression, Code Bisector and Code Quality**
Authors	Sivanandan, S.
Published	2015
Domain	Customer-facing
Testing tools	Jenkins, Cruise Control, Robot Framework, Perforce, Git
Agile practices	Dynamic Regression, Code Bisector, Code Quality, CI

***Table 32.** Dynamic Regression, Code Bisector, Code Quality characterization matrix*

	Model Criteria										**Representation Criteria**									**Methodological Criteria**						
	Process elements represented by the model					Process environments represented by the model					Information perspectives				Notation characteristics					Modelling procedure		Procedure coverage		Procedure definition		
							cultural								from the viewpoint of information quality			from the viewpoint of formal notation								
	agent	activity	artefact	role	event	organizational	creative	social inter-	environ-	scientific/technological	fuctional	behavioral	ogranisational	informative	informal	formal	automated	text	graphic	non-develop	developed	partial	sufficient	undefined	semi-defined	defined
DR, CB & CQ					X									X	X				X		X	X			X	

4.2.6 N-tiered Test Automation System for Agile (2014)

Day (2014) proposes a multi-tiered test automation architecture for Agile software systems that increases both test coverage and depth. According to Day (2014), test automation is a major characteristic of a mature Agile development team. Compared to traditional stable systems, constantly evolving Agile systems face the challenge of maintenance overhead that negates the return of investment brought by test automation. The N-tiered test automation architecture tries to solve this challenge by separating the project into distinct tiers (application layers) that can operate under instable systems. The architecture model operates in customer-facing domain.

Each distinct layer has different interfaces to transfer data between different clients, such as web application receiving data from one system and sending data to another system. The number of layers depends on the complexity of the project. Day (2014) argues that there is no set number of layers that a system must have, but typically it has at least two: presentation and business. The presentation layer includes the graphical user interface (GUI) and the business layer includes parts of the system handling the business logic. Day (2014) also provides examples of other typical layers, including a data tier to test data integrity and web services tier to test API responses between systems.

Day (2014) showcases a case study of building an enterprise application using Java technologies during thirteen two-week sprints. The case study project was designed to manage and maintain a record of user specific entities and to perform domain specific analysis. The test automation architecture is depicted in Figure 12. The architecture is separated into two layers: front-end and back-end. The front-end test automation architecture handles basic GUI and user interaction validations with smoke style test suites. The displayed GUI data is retrieved from a data store that seeds the test data to drive testing. The back-end tes automation architecture validates the business rules, data integrity and web service functionality.

Day (2014) observed that change occurred maintenance was decreased, as clearly defined abstraction layers allowed for quick refactoring of the code. Concentrating on each layer specific context allowed the test automation to increase the coverage and depth of test scenarios. When compared to the prior release, the new case study test automation architecture saved 720 man hours per sprint on testing and 2,5 million dollars in support costs.

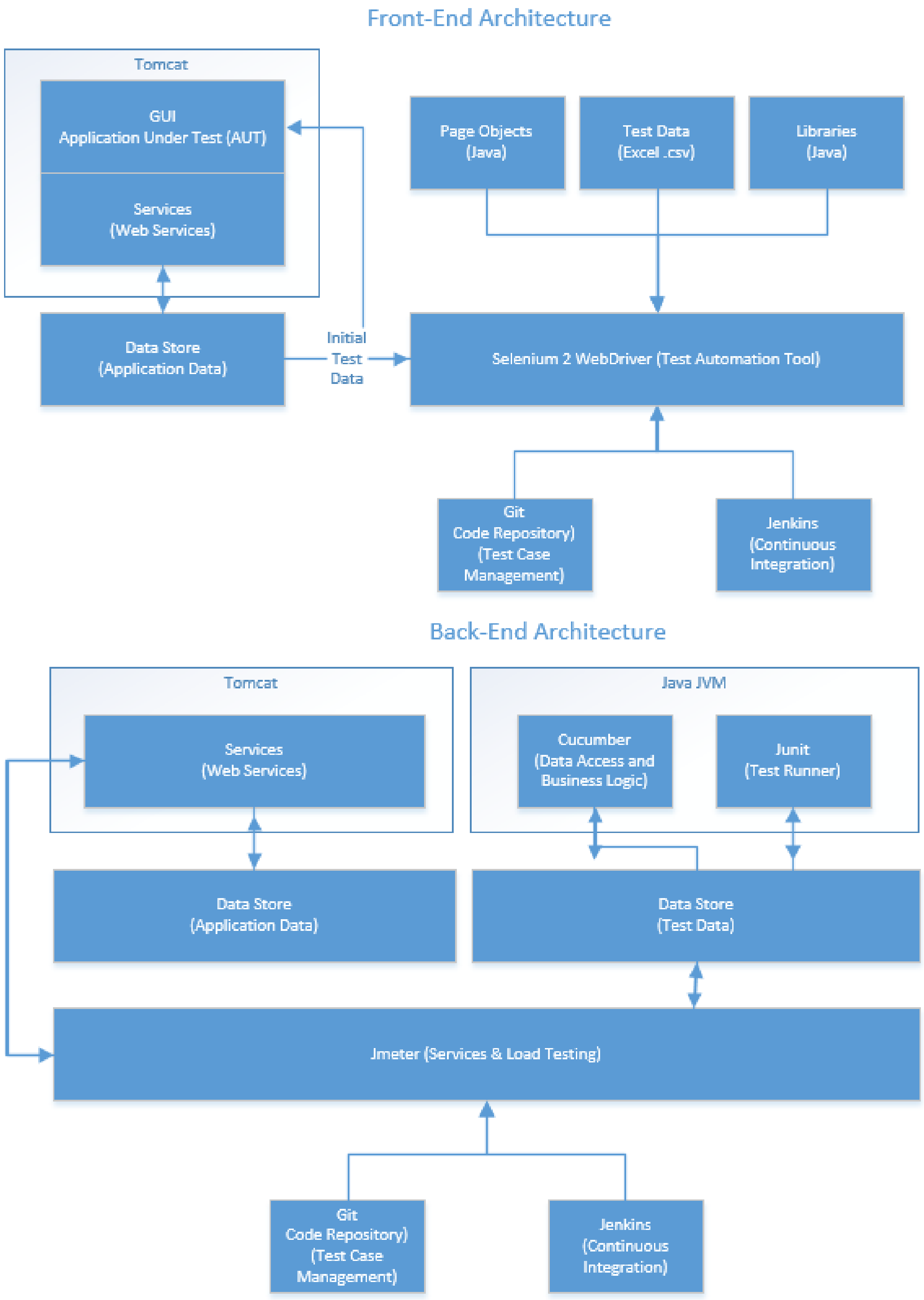

Figure 12. *N-tiered Test Automation System (Day 2014)*

In the context of prescriptive modelling and model criteria, process elements, such as actor, activity, artefact, role or event are not defined. The model criteria does not present process environment elements, such as organisational, creative ability, social interaction, environment flexibility or scientific/biological environment. The representation criteria and information perspective can be defined as informative (presenting information entities output or manipulated by the process). From the viewpoint of information quality, the information is formally presented in graphic form. In the methodological criteria, the procedure can be characterized as developed (showing defined causal relationships between procedure elements), covering a part (does not include organizational environment) of the test automation process with defined procedures (presented procedures are not defined). Review of the model is presented in Table 33 and characterization matrix in Table 34.

Table 33. *N-Tiered Test Automation System review*

Name	**N-tiered Test Automation System**
Authors	Day, P.
Published	2014
Domain	Customer-facing
Testing tools	Selenium2 WebDriver, Excel, Git, Jenkins, JMeter, Cucumber, JUnit
Agile practices	CI, Test case management tool

Table 34. *N-Tiered Test Automation System characterization matrix*

	Model Criteria										**Representation Criteria**									**Methodological Criteria**						
	Process elements represented by the model					Process environments represented by the model					Information perspectives				Notation characteristics					Modelling procedure		Procedure coverage		Procedure definition		
							cultural								from the viewpoint of information quality			from the viewpoint of formal notation								
	agent	activity	artefact	role	event	organizational	creative	social inter-	environ-	scientific/technological	fuctional	behavioral	ogranisational	informative	informal	formal	automated	text	graphic	non-develop	developed	partial	sufficient	undefined	semi-defined	defined
N-tiered test automation														X		X			X		X	X		X		

4.2.7 Test Automation Practices in Agile Development (2012)

Collins & Ferreira (2012) studied software industry test automation practices in Agile development in their industry report from 2012. They studied two software projects for one and a half years in Brasil. Their report identifies two software engineering practices, test automation and Agile methods, that have played a key role in producing low-cost, highly complex, maintainable software with high user satisfaction in time to market. The proposed model operates in customer-facing domain.

Collins & Ferreira (2012) define test automation as means to automate software testing activities. This refers to the development and execution of automated scripts, verifying testing requirements and the use of automated testing tools. The Agile software development process is characterized by the ability to rapidly accommodate changes in requirements and to prioritize the development of critical functionality.

Collins & Ferreira's (2012) proposal is based on incorporating Scrum methods with Agile testing. Their key factors for successful Agile testing include looking at the big picture, colloborating with the customer, building a foundation for Agile core practices, providing and obtaining feedback, automating regression testing and adopting the Agile testing mindset and the whole team approach. Different Agile testing quadrants offer different domains for automation and capturing different characteristics requires different tests. Automating all test actitivies during the development phase can take as much as fifty percent of total development time. Similarly, test automation success factors were identified as programmers' attitude, tester's having a low learning curve for testing tools, initial investment, constantly changing code leading to maintenance hell, legacy code unfit for automation, old habits of doing manual testing.

In the model, developers code, create and execute unit tests, while testers review unit tests and automate acceptance tests. Testers also do Exploratory Testing, performance testing and security testing and other *ility testing. In this case, both roles commit to the same Subversion version control system. The automation build tool used is Hudson CI that handles the fetching of the newest version of source repository, code compilation, executing unit tests, packaging, running Selenium acceptance tests and running JMeter stress tests. The tests are executed in three different environments: development, test or deployed into production. The process model is depicted in Figure 13.

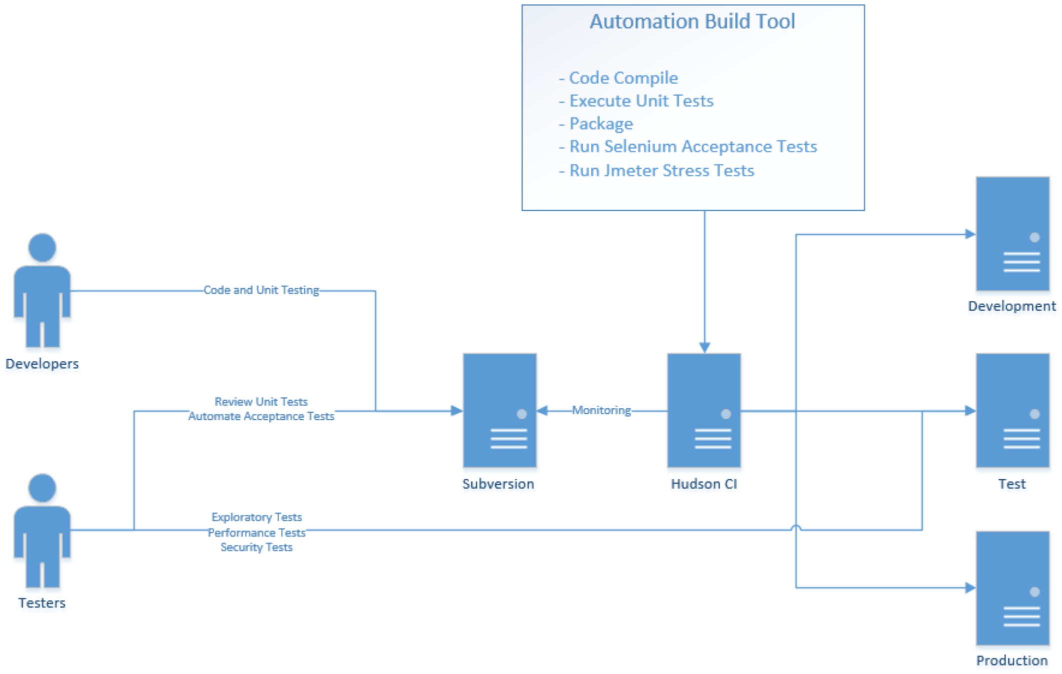

Figure 13. *Test Automation Practices in Agile Development (Collins & Ferreira 2012)*

Article mentions the team using tools, such as, TestLink, Mantis Bug Tracker, Subversion, and Jmeter. They found the that the acceptance tests kept breaking because the GUI interface was not stable enough. TestLink was used for managing test plans, writing test cases and reporting test executions. Mantis Bug Tracker was used by testers for Defect Tracking System (DTS). Subversion was used to manage and share code and documentation between team members. JMeter was used for performance and stress testing.

In the context of prescriptive modelling and model criteria, process elements, such as actor, activity, artefact, role and event are clearly defined. The developer and tester (actor) have distinct responsibilities (roles) and tasks (actitivities) producing different products (artefacts). Events triggering actions are not defined. The model criteria present process environment elements in text, such as organisational (adopting Agile mindset), creative ability (programmer’s attitude), social interaction (whole team approach), environment flexibility (initial investment) or scientific/biological (low learning curve for test tools) environment. The representation criteria and information perspective can be defined as behavioral (presenting sequential conditions for process element implementation). From the viewpoint of information quality, the information is informally presented in text and graphic form. In the methodological criteria, the procedure can be characterized as developed (showing defined causal relationships between procedure elements), covering a sufficient part (includes both organizational and technical environment) of the test automation process with semi-defined procedures (some presented procedures are defined). Review of the model is presented in Table 35 and characterization matrix in Table 36.

***Table 35**. Agile Testing Process review*

Name	**Agile Testing Process**
Authors	Collins, E. & Ferreira, V.
Published	2012
Domain	Customer-facing
Testing tools	Subversion, TestLink, Mantis Bug Tracker, Hudson CI, JMeter
Agile practices	CI, DTS

***Table 36.** Agile Testing Process characterization matrix*

	Model Criteria										**Representation Criteria**									**Methodological Criteria**						
	Process elements represented by the model					Process environments represented by the model					Information perspectives				Notation characteristics					Modelling procedure		Procedure coverage		Procedure definition		
						cultural									from the viewpoint of information quality			from the viewpoint of formal notation								
	agent	activity	artefact	role	event	organizational	creative	social inter-	environ-	scientific/technological	fuctional	behavioral	ogranisational	informative	informal	formal	automated	text	graphic	non-develop	developed	partial	sufficient	undefined	semi-defined	defined
Agile Testing Process	X	X	X	X		X	X	X	X	X		X			X			X	X		X		X		X	

4.2.8 Test Driven Development (2011)

Parsons et al. (2011) argue in “Test Driven Development: Advancing Knowledge by Conjecture and Confirmation”, how Test Driven Development (TDD) is critical Agile software development. They collaborated with an Agile team and observed the team’s adoption of the practice. Based on their findings, they propose an analytical model for TDD in Agile software development, which is depicted in Fig. 14.

Parsons et al. (2011) recount that TDD was first utilized by NASA in the 1960’s, but in the context of structured methods, it was infrequently used. TDD became later popular during the 1990’s and gained wider acceptance, particularly withing the Extreme Programming (XP) community. At present, TDD is considered to be a regular part of the Agile principles and seen in many different organisations.

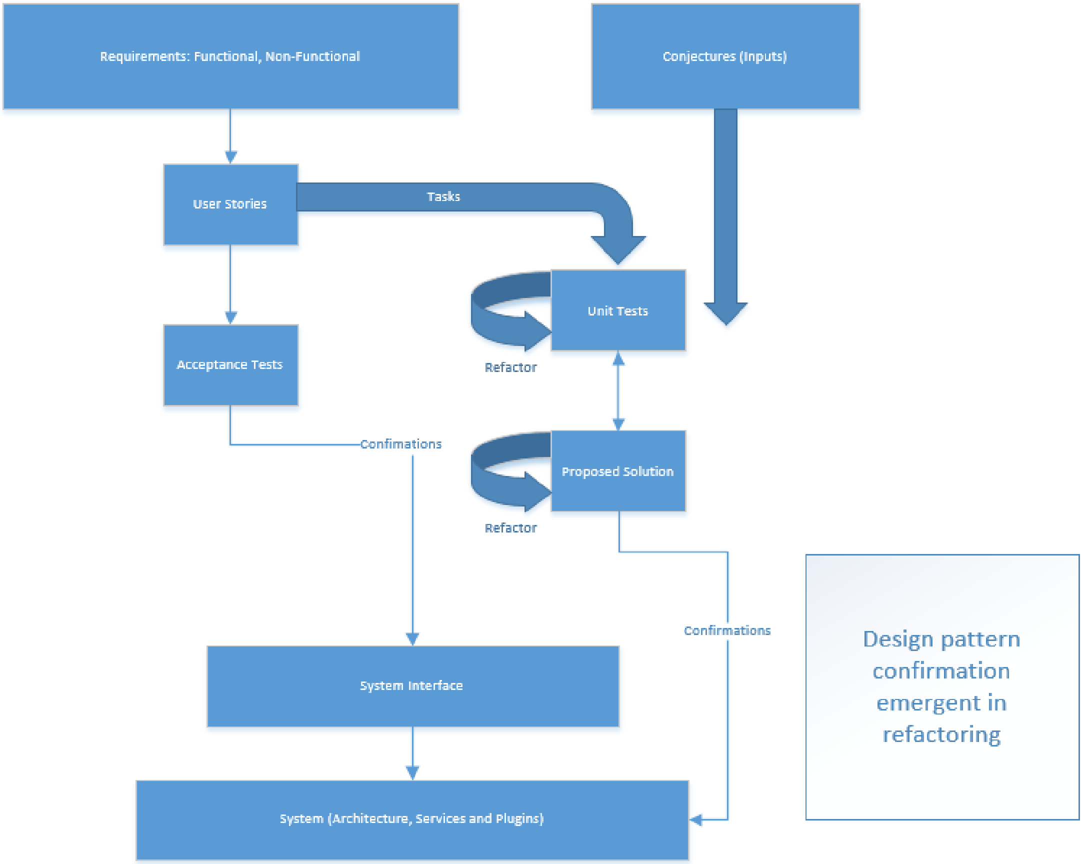

***Figure 14.** Test Driven Development (Parsons et al. 2011)*

Test Driven Development (TDD) is defined by writing tests before code. This means that granular tests provide continuous feedback of the state of the software and the tests serve as a valuable collection of unit tests for regression testing. They argue, that the main benefit of using TDD is improvement in product quality. This provides greater predictability into the development and helps to estimate the total project cost.

Test Driven Development requires the support of the customers and domain experts in order to be utilized successfully. User stories are used as informal requirements and serve as a starting point for future development and future conversations between the developers and other stakeholders. User stories are formed into multiple smaller tasks that a developer can develop unit tests for. The user stories also form the basis of the acceptance tests.

Parsons et al. (2011) write how the philosophical approach to testing differs in TDD compared to traditional testing. Their perspective is to reinterpretate Popper's theory on conjecture and falsification as advancement of knowledge. Traditionally, the conjecture has focused on the programming problems and post hoc tests are used for *falsification* of those problems. Test Driven Development focuses instead on the *positive confirmation* of software. This shift in perspective redirects the team's interest (conjecture) to write tests that confirm working program functionality and features. TDD operates in customer-facing domain.

The team used a Four Stage Model of Agile Development and present the Popper's model in the form of: 1. Intial Problem, 2. Trial Solution, 3. Error Elimination, 4. Resulting solution (with new problems). This model is presented in Figure 14, explaining how the final product desing emerges from using trial solutions and error elimination. Unit tests are refactored until all the automated unit tests are passed and acceptance tests are refactored until a proposed solution is found. Acceptance tests confirm the functionality of system Graphical User Interface (GUI) and unit tests confirm the functionality of the underlying system.

In the context of prescriptive modelling and model criteria, process elements, such as actor and role are not defined. Activities leading to artefacts are defined but events triggering them are not clearly defined. The model criteria does not present process environment elements, such as environment flexibility or scientific/biological environment. Organizational element, the role of Agile tester is mentioned in text as a crucial part of the model. Social interaction between developers is required for Pair Programming to be useful. The creative ability is presented with conjecture and emergent design patterns. The representation criteria and information perspective can be defined as functional (presenting information flows of process elements). From the viewpoint of information quality, the information is informally presented in text and graphic form. In the methodological criteria, the procedure can be characterized as developed (showing defined causal relationships between procedure elements), covering a sufficient part (includes both organizational and technical environment) of the test automation process with defined semi-procedures (some presented procedures are defined). Review of the model is presented in Table 37 and characterization matrix in Table 38.

Table 37. *Test Driven Development review*

Name	**Test Driven Development**
Authors	Parsons, D., Lal, R. & and Lange, M.
Published	2011
Domain	Customer-facing
Testing tools	N/A
Agile practices	User stories, Pair Programming, XP

Table 38. *Test Driven Development characterization matrix*

	Model Criteria										**Representation Criteria**									**Methodological Criteria**						
	Process elements represented by the model					Process environments represented by the model					Information perspectives				Notation characteristics					Modelling procedure		Procedure coverage		Procedure definition		
							cultural								from the viewpoint of information quality			from the viewpoint of formal notation								
	agent	activity	artefact	role	event	organizational	creative	social inter-	environ-	scientific/technological	fuctional	behavioral	ogranisational	informative	informal	formal	automated	text	graphic	non-develop	developed	partial	sufficient	undefined	semi-defined	defined
TDD		X	X			X	X	X			X				X			X	X		X		X		X	

4.2.9 Agile Test Framework (2011)

Jungyub et al. (2011) propose Agile Test Framework for business-to-business interoperability. Agile Test Framework consists of test case design and test execution model. The framework defines a test case in two levels: abstract and executable. The abstract level corresponds with the human interaction requirements and the executable level with the machine readable requirements. The framework operates in customer-facing domain.

Agile Test Framework depicted in Figure 15, is based on five Design Decisions: Two-level test case design, Pluggable test components and infrastructure designs, Event-driven test execution design, Modular test case design and Event-centric test case design. The framework has two key concepts: systematic test case design and test infrastructure. Jungyub et al. (2011) propose different strategies for increasing system interoperability, such as XML-based test case design, self-describing test case design and business process-based test representation. Test intrastructure is introduced as a concept that is "a permanent, invariable functional module that allows for re-configurability of test beds". Test infrastructure is designed to be modular and re-configurable; different reusable, pluggable modules can be configured to different test scenarios.

As the importance of infrastructure has expanded, the role of Test Infracture Provider has been broken into three new roles: Test Service Provider, Test Bed builder and Test Framework (TF) provider. The Test Framework provider designs the Standard Interface Definition (WSDL). The Test Service provider designs the pluggable and re-usable test components. The Test Bed builder searches for the relevant pluggable test components and generates a test harness script and assembles the test bed spefic to the desired test. Standard developers define specifications into requirements according to standards and test users define usage specifications according to usage requirements.

In the context of prescriptive modelling and model criteria, process elements, such as actor, activity, artefact, role and event are clearly defined. The actors, such as standard developer, test user and test bed builder, have distinct roles and responsibilities and events triggering them. They produce different artefacts, such as the test framework and test cases. The model criteria does not present process environment elements, such as creative ability and social interaction. Organizational elements, such as different organizational roles are addressed. Environmental flexibility is addressed as part of the modular test infrastructure. Scientific environment is presented with attention to test framework specifications and documentation. The representation criteria and information perspective can be defined as functional (presenting information flows of process elements) and behavioral (presenting a sequential conditions for process element implementation). From the viewpoint of information quality, the information is formally presented in text and graphic form. In the methodological criteria, the procedure can be characterized as developed (showing defined causal relationships between procedure elements), covering a sufficient part (includes both organizational and technical environment) of the test automation process with defined procedures (each presented procedure is defined in text). Review of the model is presented in Table 39 and characterization matrix in Table 40.

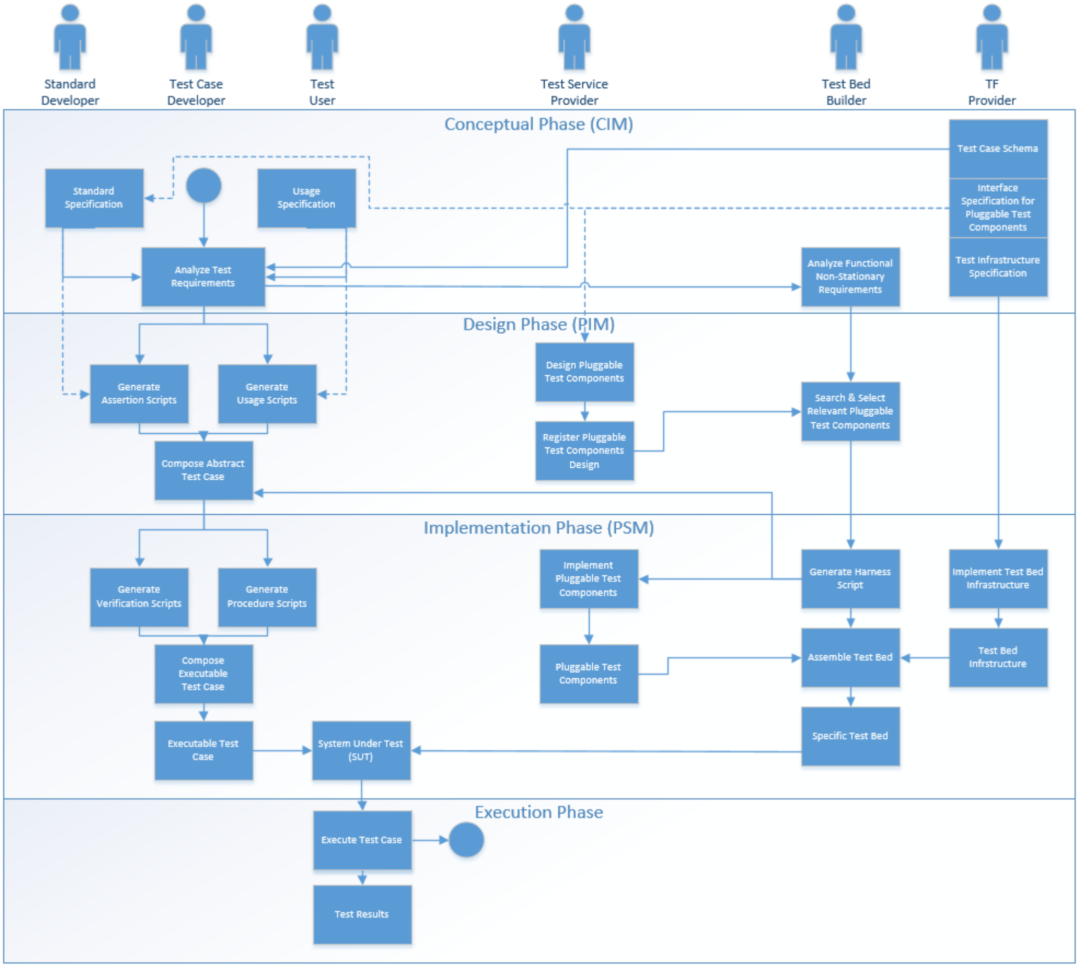
Standard Developer
Test Case Developer
Test User
Test Service Provider
Test Bed Builder
TF Provider
Conceptual Phase (CIM)
Test Case Schema
Standard Specification
Usage Specification
Interface Specification for Pluggable Test Components
Analyze Test Requirements
Analyze Functional Non-Stationary Requirements
Test Infrastructure Specification
Design Phase (PIM)
Design Pluggable Test Components
Generate Assertion Scripts
Generate Usage Scripts
Search & Select Relevant Pluggable Test Components
Register Pluggable Test Components Design
Compose Abstract Test Case
Implementation Phase (PSM)
Implement Pluggable Test Components
Generate Harness Script
Implement Test Bed Infrastructure
Generate Verification Scripts
Generate Procedure Scripts
Compose Executable Test Case
Pluggable Test Components
Assemble Test Bed
Test Bed Infrstructure
Executable Test Case
System Under Test (SUT)
Specific Test Bed
Execution Phase
Execute Test Case
Test Results

Figure 15. *Agile Test Framework (Jungyub et al. 2011)*

Table 39. *Agile Test Framework review*

Name	**Agile Test Framework**
Authors	Jungyub. W., Nenad, I. & Hyunbo, C.
Published	2011
Domain	Customer-facing
Testing tools	N/A
Agile practices	N/A

Table 40. *Agile Test Framework characterization matrix*

	Model Criteria										**Representation Criteria**									**Methodological Criteria**						
	Process elements represented by the model					Process environments represented by the model					Information perspectives				Notation characteristics					Modelling procedure		Procedure coverage		Procedure definition		
							cultural								from the viewpoint of information quality			from the viewpoint of formal notation								
	agent	activity	artefact	role	event	organizational	creative	social inter-	environ-	scientific/technological	fuctional	behavioral	ogranisational	informative	informal	formal	automated	text	graphic	non-develop	developed	partial	sufficient	undefined	semi-defined	defined
Agile Test Framework	X	X	X	X	X	X			X	X	X	X				X		X	X		X		X			X

4.2.10 Agile Method for Open-Source Safety-Critical Software (2011)

Gary et al. (2011) propose an Agile Method for Open-Source Safety-Critical Software. There has been a common misconception that Agile development is inherently incompatible with safety-critical software development. In safety-critical software applications, occurrence of an error could lead to loss of life. Gary et al. (2011) argue that Agile methods have matured enough to be clearly understood, defined and executed process models. They believe that Agile methods have contributions to safety-critical software development in the areas of implementation quality and process management. Using Scrum process management or Extreme Programming (XP) can improve the management of traditional safety critical activities. The proposed model operates in safety-critical and open-source domain.

Gary et al. (2011) use the image-guided surgical toolkit (IGSTK) as an example of their argument. The IGSTK is an open-source toolkit for surgery featuring image import, image display, registering, segmenting, tracking support and scene graph manipulation. It is developed by devoted volunteers and their part-time work. The architecture of IGSTK is heavily layered by design. All the system variables are heavily typed. It has an interface that accepts and returns event responses to the internal hardware. The hardware's the internal state machine determines, whether the requested action can be performed, depending on the state of the component.

They list six key areas to consider in safety-critical environment, which include 1. hazard analysis, 2. safety requirements, 3. designing for safety, 4. testing, 5. certification and standards and 6. resources. Hazard analysis is defined as the identification and analysis of hazards in terms of their severity and urgency. Safety requirements are specified in formal notion and allow formal analysis. Designing for safety is understood as thinking system-wide safety when modeling the system components. Extensive testing should be employed to verify the functionality of the software in an appropriate environment. The system should be assessed compared to industry standards and certified. The IGSTK focused on areas 2, 3, 4 and 5 using Agile methods. They argue that these methods have nothing that prevents their application to Agile environment. The methods are considered platform-agnostic and the most important measurement of Agile is "working software". A "right amount of ceremony" is required to have structure in the process, but not restricting individual team members' roles.

In Figure 16, the process for both the requirements and code are depicted. Requirements start as posts to a internal Wiki and put in review. The requirement is defined and then discussed in a group. If the requirement fails the review, it is aborted and moved to the log area of the Wiki under 'unaccepted'. If the requirement is requested to be modified it is moved to 'in pending', waiting for revision. If the requirement passes the group review, it is accepted and entered into the system. After that the required implementation is developed and implemented. The implementation is placed under code review and inspected. After verification, the requirement is moved into a Word document.

The coding process starts with the selection of the requirements feature list. Unit tests are developed and tested locally. After that the code is checked into a Virtual Sandbox and run nightly builds and validated. After validation, the code is moved into the Main branch. Extensive system testing and code reviews are performed. The validated release is then packaged and released into production.

In the context of prescriptive modelling and model criteria, process elements, such as actor (developer), activity (requirements gathering and implementation), artefact (requirement, documentation, code), role (developer responsibility to safety) and event (triggering of evaluation and implementation) are clearly defined. The model criteria does not present process environment elements, such as scientific/biological environment. Organizational elements is addressed by only accepting highly trained applicants to the development process. Social interaction is addressed by promoting constant communication. Creative ability leverages the collective creative power of the open-source community. Evolving process is mentioned as part of environment flexibility.

The representation criteria and information perspective can be defined as behavioral (presenting a sequential conditions for process element implementation). From the viewpoint of information quality, the information is formally presented in text and graphic form. In the methodological criteria, the procedure can be characterized as developed (showing defined causal relationships between procedure elements), covering a sufficient part (includes both organizational and technical environment) of the test automation process with defined procedures (presented procedures are defined).

Review of the model is presented in Table 41 and characterization matrix in Table 42.

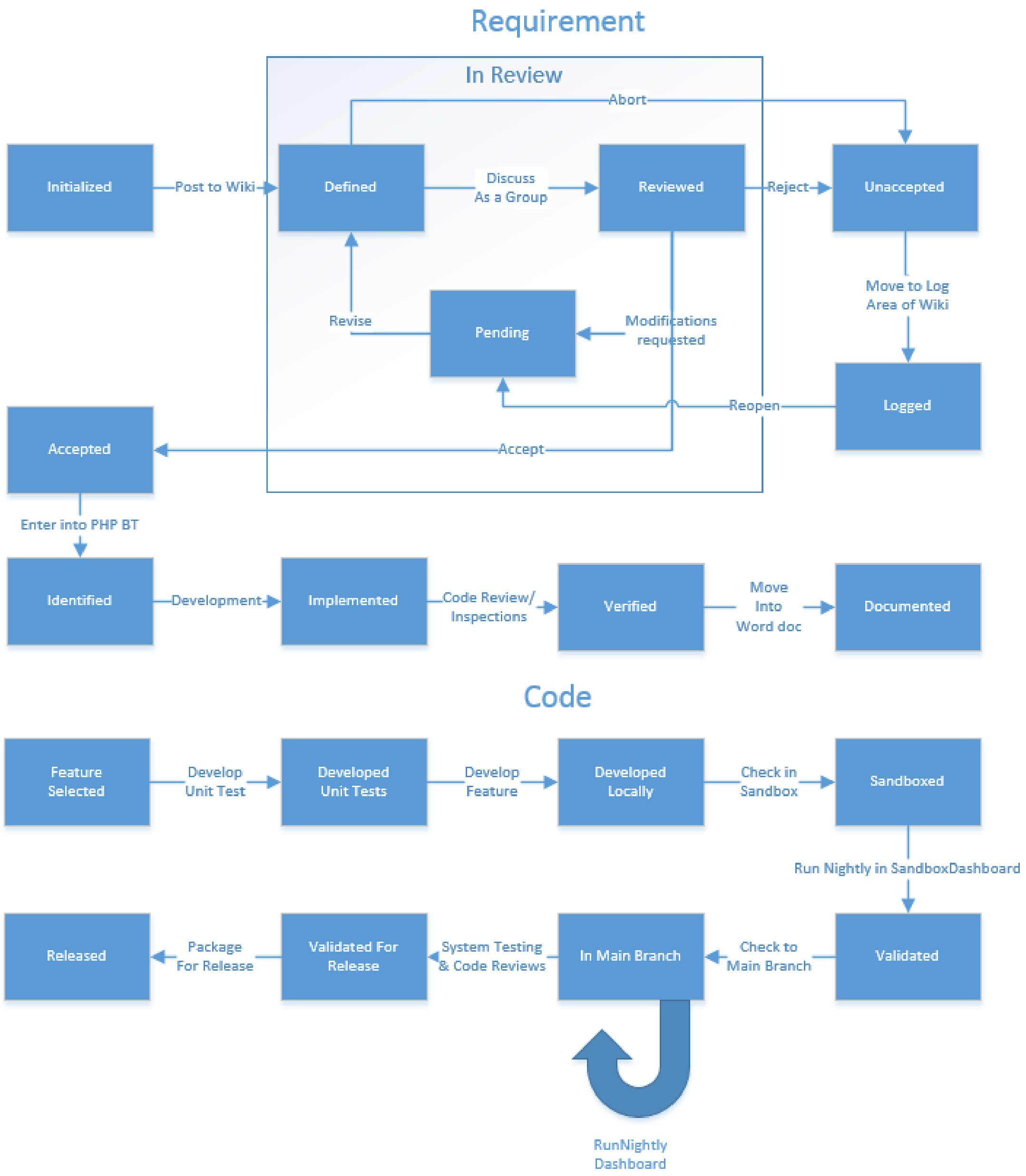

Figure 16. *Agile Method for Open-Source Safety-Critical Software (Gary et al. 2011)*

Table 41. *Agile Method for Open-Source Safety-Critical Software review*

Name	**Agile Method for Open-Source Safety-Critical Software**
Authors	Gary, K., Enquobahrie, A., Ibanez, L., Cheng, P., Yaniv, Z., Cleary, K., Kokoori, S., Muffih, B. and Heidenreich, J.
Published	2011
Domain	Open-source, safety-critical
Testing tools	CDash, CMake, Doxygen
Agile practices	Scrum, Pair Programming, CI, XP

Table 42. *Agile Method for Open-Source Safety-Critical Software characterization matrix*

	Model Criteria										**Representation Criteria**									**Methodological Criteria**						
	Process elements represented by the model					Process environments represented by the model					Information perspectives				Notation characteristics					Modelling procedure		Procedure coverage		Procedure definition		
							cultural								from the viewpoint of information quality			from the viewpoint of formal notation								
	agent	activity	artefact	role	event	organizational	creative	social inter-	environ-	scientific/technological	fuctional	behavioral	ogranisational	informative	informal	formal	automated	text	graphic	non-develop	developed	partial	sufficient	undefined	semi-defined	defined
Agile method	X	X	X	X	X	X	X	X	X			X				X		X	X		X		X			X

4.3 Data synthesis

4.3.1 Domain and characterization matrix

Review of the research data identified two challenges: the varying depth of detail, accuracy and complexity between model descriptions and the lack of standardized model for measuring test automation modeling. The lack of standardized model meant that researched articles varied in description, depth of detail, utility, environment, scope and domain. In effort to find common attributes between models, the domain of each model was classified by the intended domain of application for the model as described by the authors. If data was not available, the classification was deemed as customer-facing.

When categorizing the models by their domain of application, it can be noted that the majority of the models operated within the customer-facing interface. Seven (7) of the ten (10) selected models were categorized as operating in the "customer-facing" domain. Two of the models operated within safety-critical software development. One model was designed exclusively to be used for acceptance testing. One of the safety-critical models was also designed to be developed together with the open-source community. List of model domain categorizations is presented in Table 43.

Table 43. *Domain categorization of Agile test automation models*

Name	Domain
1. **Lean Canvas Model (2017)**	Customer-facing
2. **Quality Experience (2016)**	Customer-facing
3. **Agile Testing for Railway Safety-Critical Software (2016)**	Safety-critical
4. **Development Method for Acceptance Test Process (2016)**	Acceptance testing
5. **Dynamic Regression, Code Bisector and Code Quality (2015)**	Customer-facing
6. **N-tiered Test Automation System for Agile (2014)**	Customer-facing
7. **Test Automation Practices in Agile development (2012)**	Customer-facing
8. **Agile Test Framework (2011)**	Customer-facing
9. **Test Driven Development (2011)**	Customer-facing
10. **Agile Method for Open-Source Safety-Critical Software (2011)**	Safety-critical, Open-source

Prescriptive modelling was chosen to characterize different aspects of the examined test automation models. Deciding what features of model, representational or methodological criteria were fullfilled by a model description proved difficult. Decision was made to conclude all undefined or improperly described procedure definitions as undefined. The prescriptive modelling characterizations are described in Table 44.

Table 44. *Characterization matrix attributes*

	Model Criteria										**Representation Criteria**									**Methodological Criteria**						
	Process elements represented by the model					Process environments represented by the model					Information perspectives				Notation characteristics					Modelling procedure		Procedure coverage		Procedure definition		
							cultural								from the viewpoint of information quality			from the viewpoint of formal notation								
	agent	activity	artefact	role	event	organizational	creative ability	social interaction	environment flex-	scientific/technological	fuctional	behavioral	ogranisational	informative	informal	formal	automated	text	graphic	non-develop	developed	partial	sufficient	undefined	semi-defined	defined
1.		x	x		x							x			x			x	x		x	x		x		
2.	x	x	x	x	x						x	x				x		x	x		x	x				x
3.	x	x	x		x						x					x		x	x		x	x				x
4.					x									x		x		x	x		x	x		x		
5.					x									x	x				x		x	x			x	
6.														x		x			x		x	x		x		
7.	x	x	x	x		x	x	x	x	x		x			x			x	x		x		x		x	
8.		x	x			x	x	x			x				x			x	x		x		x		x	
9.	x	x	x	x	x	x			x	x	x	x				x		x	x		x		x			x
10.	x	x	x	x	x	x	x	x	x			x				x		x	x		x		x			x

When analyzing the results of the characterization matrix, observations can be noted. The model with most prescriptive model characterizations was *Agile Method for Open-Source Safety-Critical Software* (2011) with almost all of the model, representational and methodological criteria. Models, such as *Test Driven Development* (2011) and *Test Automation Practices in Agile Development* (2012) were the second most characterized. The model with least prescriptive model characterization was *N-tiered Test Automation System for Agile* (2014) with no model criteria and only small number of representational and methodological criteria.

Model criteria included process elements were represented reasonably well in most (seven out of ten) articles. The most commonly prescribed process elements were *activity* and *artefact* and the least prescribed was *role*. Process environment was not represented in over half of the models (six out of ten). The most commonly prescribed process environment was organizational (four out of ten) and the least scientific or biological environment. Representation criteria included information perspective which was most commonly characterized as behavioral (five models). Functional perspective was utilized in four cases (4) and informative perspective in three (3). Information quality was formally presented in six (6) cases. From the viewpoint of formal notation, all of the examined articles featured a process model depiction in graphic. Eight out of ten articles also featured the model definitions in text while two of the articles did not. Methodological criteria included the state of the modelling procedure, which was deemed developed in all ten (10) articles. In six (6) articles the procedure coverage was only partial and in four (4) sufficient. Procedure definitions were well-defined in four (4) articles, semi-defined in three (3) articles and undefined in three (3) articles.

From the characterization matrix it can be observed that six of the selected models did not present process environment elements and that four models did present. The six models that did not include process environment elements were more recently published (after 2015) than the other group but it can be coincidental. A division between the two groups might give some validation to the idea of categorizing the researched models into "technologically" described models and "technologically and organizationally" described models. Technologically described models do not take into consideration the organizational or human elements of the process and concentrate on describing the technological steps needed to execute the testing process. Technologically and organizationally described models discuss human elements of the process and how a test automation team should be managed and organized within the testing organization.

4.3.2 Agile practices and tools

The following Agile practices were mentioned in the researched articles: Kanban, Scrum, XP, CI, CD, MBT, SoC, SbE, ATDD, Dynamic Regression, Code Bisector, Code Quality, Test case management tool, DTS, User stories and Pair Programming. Most commonly mentioned Agile practices were Continuous Integration CI (6), Experimental Programming XP (3) and Scrum (3). The popularity of the CI practice might explain why most of mentioned tools were CI tools. One of the reviewed articles did not mention any Agile practices or tools. Mentioned Agile practices and their categories are listed in Table 45.

Table 45. *Mentioned Agile practices and their categories*

Agile practice	**Category**	**Times mentioned**
Continuous Integration CI	Development method	6
Experimental Programming XP	Development method	3
Scrum	Management method	3
User stories	Development method	1
Pair Programming	Development method	1
Acceptance Test Driven Development ATDD	Development method	1
Dynamic Regression, Code Bisector, Code Quality	Development method	1
Separation of Concerns SoC	Development method	1
Specification by Example SbE	Development method	1
Model Based Testing MBT	Development method	1
Continuous Deployment CD	Management method	1
Test case management tool	Management method	1
Defect Tracking System DTS	Management method	1
Kanban	Management method	1

The total number of different Agile practices mentioned was fourteen (14). Nine (9) of the Agile practices described in Chapter 3 (CI, XP, Scrum, Kanban, ATDD, SbE, DTS, User stories and Pair Programming) were mentioned in the researched articles. The majority of the practices were mentioned only once.

The mentioned Agile practices can be categorized into two groups, management and development methods. Management methods (5), such as Scrum and Kanban, focus on the management of the development and testing team. Development methods (9), such as User stories or Pair Programming, focus on managing the technical implementation of development and testing. In this categorization, CI is categorized as a way of developing software (development method) and CD as a managerial decision to empower the team (management method). When comparing the two categories, development and management, the number of Agile development methods is emphasized.

The following tools were mentioned in the researched articles: Jenkins, Travis CI, Apache Maven, JaCoCo, Checkstyle, Fitnesse, Cruise Control, Robot Framework, Perforce, Git, Selenium2 WebDriver, Excel, Cucumber, JUnit, Subversion, TestLink, Mantis Bug Tracker, Hudson CI, JMeter, CDash, CMake and Doxygen. Most commonly mentioned tools were open-source CI-server tool Jenkins (3), open-source source version control system Git (2) and testing tool JMeter (2). Many of the mentioned tools can be categorized as Continuous Integration and Deployment CI & CD tools, such as Jenkins, Travis CI and Hudson CI. Different test frameworks were mentioned, such as Robot Framework and JMeter and different Defect Tracking Systems DTS, such as Mantis Bug Tracker and TestLink. Four of the researched articles did not have any mentions of testing tools. Mentioned tools and their categories are listed in Table 46.

Table 46. *Mentioned tools and their categories*

Tool	*Category*	*Times mentioned*
Jenkins	Continuous Integration	3
Travis ci	Continuous Integration	1
Hudson ci	Continuous Integration	1
Cruise control	Continuous Integration	1
Perforce	Continuous Integration	1
Git	Source version control	2
Subversion	Source version control	1
Jmeter	Test framework	2
Fitnesse	Test framework	1
Jacoco	Test framework	1

Checkstyle	Test framework	1
Selenium2 webdriver	Test framework	1
Cucumber	Test framework	1
Junit	Test framework	1
Robot framework	Test framework	1
Mantis bug tracker	Defect Tracking System	1
Testlink	Defect Tracking System	1
Apache maven	Build tool	1
Cmake	Build tool	1
Doxygen	Build tool	1
Cdash	Build tool	1

The total number of different tools mentioned was twenty (20). The most mentioned category was Test framework with seven (7) tools, second Continuous Integration (5), third Build tool (4) and as last Source version control (2) and Defect Tracking System (2). The majority of the tools were only mentioned once.

5. DISCUSSION

5.1 Summary of Agile test automation models

Summary of the examined Agile test automation models and their domain of application, Agile practices and tools is presented in Table 47.

Table 47. *Summary of Agile test automation models*

Name	*Domain*	*Agile practices*	*Tools*
Lean Canvas Model (2017)	Customer-facing	Scrum, Test case management tool	N/A
Quality Experience (2016)	Customer-facing	Kanban, Scrum, XP, CI	N/A
Agile Testing for Railway Safety Critical Software (2016)	Safety-critical	CI, CD, MBT	Jenkins, Travis CI, Apache Maven, JaCoCo, Checkstyle
Development Method for Acceptance Test Process (2016)	Acceptance testing	SoC, SbE, ATDD	FitNesse
Dynamic Regression, Code Bisector and Code Quality (2015)	Customer-facing	Dynamic Regression, Code Bisector, Code Quality	Jenkins, Cruise Control, Robot Framework, Perforce, Git
N-tiered Test Automation System for Agile (2014)	Customer-facing	CI; Test case management tool	Selenium2 WebDriver, Excel, Git, Jenkins, JMeter, Cucumber, JUnit
Test Automation Practices in Agile Development (2012)	Customer-facing	CI, DTS	Subversion, TestLink, Mantis Bug Tracker, Hudson CI, JMeter
Agile Test Framework (2011)	Customer-facing	User stories, Pair Programming, XP	N/A
Test Driven Development (2011)	Customer-facing	N/A	N/A
Agile Method for Open-Source Safety Critical Software (2011)	Safety-critical, OS	Scrum, Pair Programming, CI, XP	CMake, CDash, Doxygen

5.2 Synthesized generic model

Using the data from the researched articles, a generic model for prescribing Agile test automation is synthesized. The purpose of the synthesized generic model is to generalize, what characteristics describe a typical Agile test automation model. The domain, model characteristics, practices and tools are chosen by their commonality in the research data.

The most common domain of application for the test automation models was customer-facing development. The prescriptive modelling of a test automation model should characterize following processs elements: *agent, activity, artefact* and *event*. The process environment is not required to be characterized. The information perspective is characterized as functional and formally presented in text and graphic form. Procedures should be developed, procedure coverage sufficient and procedure definitions defined.

The most commonly mentioned Agile development practice was CI and the most commonly mentioned Agile management practice was Scrum. The most commonly mentioned tool for Continuous Integration was Jenkins and for source version control Git. For test frameworks and build tools, there are multiple options depending on technological decisions, such as using JUnit and Apache Maven. Defect tracking is managed by Mantis Bug Tracker or TestLink. The synthesized generic model attributes are listed in Table 48.

Table 48. *Synthesized generic Agile test automation model characteristics*

Attribute	*Value*
Domain	Customer-Facing
Model criteria	The following process elements should be characterized: *agent, activity, artefact* and *event*.
Representation criteria	The information perspective is characterized as functional and formally presented in text and graphic form.
Methodological criteria	Procedures should be developed, procedure coverage partial and procedure definitions defined.
Management practice	Scrum
Development practice	Continuous Integration
CI tool	Jenkins
SVC tool	Git
Defect Tracking System	Mantis Bug Tracker, TestLink
Test framework	JUnit, JMeter, FitNesse, Cucumber, JaCoCo, Selenium2 WebDriver, CheckStyle, Robot Framework
Build tool	Apache Maven, CDash, CMake, Doxygen

5.3 Discussion

The discussion of Agile test automation models and their summary is conducted with the help of the research articles and the supporting questions

- How to evaluate different Agile test automation models?
- What characteristics describe Agile test automation models?
- What domains are found in Agile testing literature?
- What Agile practices are found in Agile testing literature?
- What tools are found in Agile testing literature?
- How does the synthesized generic model compare with Agile testing literature?

For the discussion, the forty (40) read articles excluded by the methodological inclusion criteria, will be used for the literary comparisons.

5.3.1 How to evaluate different Agile test automation models?

Evaluating and comparing different Agile test automation models against each other and the research literature proved difficult. The reviewed ten models were vastly different in their description, depth of detail, utility, environment, scope and domain. When comparing the ten selected models, only qualitative comparisons can be made. Analyzing the different objectives of each model proved valuable for understanding the models, their motivations and their application domain.

Lean Canvas Model (2017) is based on *Lean* philosophy and described as a "white board with several blocks used for the evaluation of business ideas" and based in using Scrum for managing the testing process. *Quality Experience* (2016) model was used to describe a mode of quality assurance and deployment with emphasis on conscious empowerment decision, the role of test automated feedback, rapid repair of defects and motivation effect that lead to frequent deployment. *Agile Testing for Railway Safety-Critical Software* (2016) model focused on safety-criticality in the Chinese railway industry where they suggested utilizing Continuous Integration and deployment, generating unit test coverage logic automatically and generating paths from Model Based Testing. *Acceptance Test Process* (2016) was designed for acceptance testing with test automation designed utilizing Separation of Concerns to separate design decisions into different layers. *Dynamic Regression, Code Bisector and Code Quality* (2015) presents three practices used for enhancing Agile methodologies through dynamically running unit and functional automation tests only for the changed code, a tool used for finding broken piece of code faster and defining code with seven quality axes. *N-tiered Test Automation System for Agile* (2014) describes test automation implementation details for Java-based GUI-testing with the idea of separating the test automation architecture into distinct application layers, similar to the *Acceptance Test Process* (2016). *Test Automation Practices in Agile Development* (2012) describes commonly utilized practices, such as looking at the big picture, colloborating with the customer, building a foundation for Agile core practices, providing and obtaining feedback, automating regression testing and adopting the Agile testing mindset and the whole team approach. *Test Driven Development* (2011) focuses on transforming the development conjecture to *positive confirmation* of software functionality, instead of *falsification* of software functionality, with similar ideas as *Quality Experience* (2016). *Agile Test Framework* (2011) focuses on business-to-business interoperability and has multiple defined roles for testing and maintaining the testing infrastructure. *Agile Method for Open-Source Safety-Critical Software* (2011) focuses on safety-criticality in the open-source environment and presents a way to apply Agile in the development of an image-guided surgical toolkit.

When researching the forty articles, multiple other proposals for test automation frameworks were found, such as *GUITAR* for GUI-testing (Nguyen et al. 2013), a keyword-driven test automation framework (Zhongqian et al. 2013), an automatic testing framework *Agilework* for web testing that is based on modular design (Wang et al. 2014), an automatic page object generator for web testing *APOGEN* that is based on the page object pattern where page objects are facade classes abstracting the internals of web pages into high-level business functions that can be invoked by the test cases (Mariani et al. 2017), *Chameleon model* that is based on the Test Pyramid (presented in Chapter 2.6) and aims to provide a high degree of adaptability to changing test environments (Thopate & Kachewarr 2012), an open-source system *ZiBreve* with the aim of supporting the process of refactoring implementation level tests to business-level specifications (Mugridge et al. 2011) and an automated Agile regression testing approach that is based on *Weighted sprint test cases prioritization WSTP* (Kandil et al. 2016).

The number of different use cases for test automation in Agile context was multiple and model comparisons were difficult. One observation is that many of the test automation frameworks were based on the same type of architecture; Garousi & Mäntylä (2016) found in their research that lower-level testing tools and frameworks were more similar or based on the same existing architecture or framework, such xUnit tools. On system testing level they found that tools were more diverse and more dependent on the application domain. Some of the articles were focused on the technical implementation of the test automation framework, such as *GUITAR* and *Chameleon model*, while others, such as *WSTP*, were more focused on the managerial and organizational aspects of test automation.

5.3.2 What characteristics describe Agile test automation models?

Prescriptive modelling was found not to be ideal for model characterization or decription. Many of the important differences between Agile test automation models were not captured by prescriptive modelling, such as process flow or utilized tools and methods. The research literature had multiple qualitative Agile test automation characterizations but no other examples of using prescriptive modelling or the characterization matrix.

Models, such as, *Quality Experience* (2016), focused on process environment elements and capturing top-level process of test automation and presented no tools for achieving it. *Test Driven Development* (2011) provided no tool and no practices. On the contrast, models, such as, *Test Automation Practices in Agile Development* (2012), provided multiple pratices and tools for its application. *Agile Test Framework* (2011) was the only model to mention User stories as part of Agile practices. Models, such as, *Agile Method for Open-Source Safety Critical Software* (2011), provided a full prescription of different process elements and process environments, while models, such as, *N-tier-ed Test Automation* (2014) provided no information on process elements and concentrated on the technical application of test automation. *Lean Canvas Model* (2017) is part of the larger *Lean* philosophy while *Dynamic Regression, Code Bisector and Code Quality* (2015) could be described as a set of best practices utilized by a single development team. *Development Method for Acceptance Test Process* (2016) concentrates on acceptance testing, while *Agile Testing for Railway Safety Critical Software* (2016) concentrates on safety-critical software.

When describing Agile test automation characteristics using the prescriptive modelling characterization matrix, it can be observed that almost half of the examined models presented both technological and organizational element characterization and half of the examined models only presented technological element characterization. This leads to the idea that process environment modelling is a preferred but not required characteristic for an Agile test automation model. Agile test automation model criteria for process elements included the following characteristics: *agent, activity, artefact* and *event*. The representation criteria and the information perspective are functional and formally presented in text and graphic form. The methodological criteria for procedures is be developed, procedure coverage sufficient and procedure definitions defined.

Prescriptive modelling examples of test automation models were not found in the researched literature for direct comparison. Using examples from the forty articles, Agile test automation can be characterized as having a continuous and smooth flow of delivering value to the customer and aiming to be highly focused and responsive to customer needs (Petersen & Wohlin 2010). Other examples mention the increased popularity of Agile methods in the central role of regression testing in maintaining software quality (Parsons et al. 2013). Regression testing in general was mentioned as the most frequently automated testing task with the most easily attainable benefits. Sfetsos & Stamelos (2011) found in their research that in the industry Test Driven Development was deemed to improve quality of software but studies conducted in academia were contradictory and results varied in different contexts. Mäntylä et al. (2014) found that rapid releases make testing more continuous whicle leads to proportionally smaller spikes before the main release.

5.3.3 What domains are found in Agile testing literature?

The research found a total of four (4) different domains of application for the selected ten Agile test automation models. The models were utilized in the following domains: customer-facing, safety-critical, acceptance testing and open-source development. The most common domain was customer-facing domain. The least commonly mentioned domain was acceptance testing and open-source development.

Agile methods were previously not seen as mature enough to be used in safety-critical applications, but the research found two examples of utilizing Agile testing in safety-critical domain, *Agile Method for Open-Source Safety-Critical Software* (2011) and *Agile Testing for Railway Safety-Critical Software* (2016). Both models use virtual test environments for simulating tests that would be hazardous or impossible to implement in real-life cases. The Agile method was also an open-source project utilizing the members of the open-source community to excercise the released code at an early stage and to find defects.

Most test automation models found in the research literature were designed for customer-facing domains, such as a behavior-driven automation framework (Sivanandan & Yogeesh 2014) and *Chameleon model* (Thopate & Kachewarr 2012). Web-based testing frameworks were particularly popular in customer-facing domain, such as *GUITAR* (Nguyen et al. 2013). Other domains of application found in the research literature included enterprise resource planning (ERP) systems in Sri Lanka (Hushalini et al. 2014). Examples of industries applying Agile test automation were also mentioned, such as banking, telecommunication and manufacturing in India (Jigeesh et al. 2015), mobile application development (Kirmani et al. 2017) and the use of Selenium tools in the telecommunication industry (Garousi & Mäntylä 2016).

5.3.4 What Agile practices are found in Agile testing literature?

The research found a total of fourteen (14) different Agile practices mentioned in the selected ten articles. The articles mention the use of the following Agile practices: Kanban, Scrum, XP, CI, CD, MBT, SoC, SbE, ATDD, Dynamic Regression, Code Bisector, Code Quality, Test case management tool, DTS, User stories and Pair Programming. The most commonly mentioned practice was Continuous Integration.

When dividing the mentioned Agile practices into *management* or *development* method categories, it can be noted that the majority of the practices focused on technical development implications and minority on the management of the test automation team. Continuous Integration, Experimental Programming and Scrum were mentioned as the most frequently used management practices. Number of different technical development practices, such as, User stories and Pair Programming are utilized depending on the technological ramifications of the system under test.

In the research literature, multiple examples of Agile practices and statistics of Agile practice usage were found. Kasurinen et al. (2009) studied software organizations, their software process and testing policies. In their survey, they found that the median percentage of automation in testing was ten (10) and the median for use of Agile methodologies was thirty (30) percent. On average, twenty-five (25) percent of development effort was spent on testing. Perkusich et al. (2015) mention a study stating that from a survey of 4000 practioners, Scrum was found the most popular Agile development process with 56% preference, with mentions of the use of Kanban and XP. Kirmani (2017) mentions multiple Agile practices not discussed in this research, such as Crystal, Feature Driven Development (FDD) and Dynamic System Development Method (DSDM). Kirmani (2017) argues that in literature, Extreme Programming and Scrum are the most prevalent Agile methods for mobile application development. Kirmani mentions TDD and Pair Programming as part of the Agile toolkit. Korhonen (2011) evaluated the impact of adopting Agile in software defect management practices. Korhonen refers to studies implicating that Pair Programming and CI help to improve code quality and reduce number of defects. In their research, they found that after twelve months from starting Agile transformation, the following daily Agile practices, such as User stories, short time-boxed iterations, Scrum, retrospectives and product backlog were still utilized. Technical Agile practices that were adopted by over fifty (50) percent of development teams, were refactoring, tests written at the same time as code and CI. The utilization of Test Driven Development, Pair Programming, collective code ownership and ATDD diminished to less than fifty (50) percent at the end of the twelve-month period. Korhonen found multiple recommendations for defect management, such as specifying faults to be reported, creating practices for prioritizing between bug fixes and feature development during sprint, evaluating fault reporting tools and evaluating multisite development impact on management. Based on their results, Korhonen concludes that the number of closed defects improved after changing defect reporting practices. Korhonen proposes three key practices to improve defect management that are product backlog prioritization, Continuous Integration with automated tests and a short sprint cycle. Jigeesh et al. (2015) performed an empirical study of Agile testing attributes in India. They found that there is a vast amount of literature discussing the Agile concepts and methodologies on Agile software development but that the literature is very much limited in Agile testing. In their research, they concluded that two Agile testing features, prioritization of features and iterative readiness for release, were regarded as the most important features in the surveyed industries (banking, telecommunication and manufacturing).

The research literature contains descriptions of Agile practice usage similar to the ten (10) selected models. CI was regarded as one the most important technical practices in the reviewed models as well as in the research literature, as Korhonen (2011) found in their research. Scrum was regarded as the most preferred Agile management practice in the reviewed models. The research literature also contained Agile practices and models that were not found in the reviewed models, such as as Crystal, Feature Driven Development (FDD) and Dynamic System Development Method (DSDM).

5.3.5 What tools are found in Agile testing literature?

The research found a total of twenty (20) different testing tools mentioned in the selected ten articles. The articles mention the use of the following testing tools: Jenkins, Travis CI, Apache Maven, JaCoCo, Checkstyle, Fitnesse, Cruise Control, Robot Framework, Perforce, Git, Selenium2 WebDriver, Excel, Cucumber, JUnit, Subversion, TestLink, Mantis Bug Tracker, Hudson CI, JMeter, CDash, CMake and Doxygen. The most commoly mentioned tool was Jenkins.

When dividing the tools into categories depending on their intended use, tools were found in five (5) categories: Continuous Integration, Source version control, Test framework, Defect Tracking System and Build tool. The most common category was Test framework with seven (7) different tools. Five (5) of the tools were categorized as Continuous Integration tools, four (4) as Build tools, two (2) as Source version control tools and two (2) as Defect Tracking Systems.

In the research literature, multiple examples of test automation tools and their usage were found. Kasurinen et al. (2009) surveyed the popularity of testing tools and found that test case management tools were the most popular category with 15 organizational units out of 31 utilizing them. Second in popularity were unit testing tools and third test automation implementation tools. Other categories mentioned performance testing tools, bug reporting tools and test design tools. Fawad et al. (2015) surveyed Pakistani software companies and their testing activities and found that the use of automated tools was not prevalent with 35.7% of respondents stating that no automated testing was performed in their organization. 40.4% of respondents state using no testing tools for testing and that only 19% used testing tools for every project. The cost of use was deemed as the biggest barrier of entry in adopting testing tools. Hushalini et al. (2014) performed an empirical study of the use of test automation in Sri Lankan's software development projects. They found usage of NUnit testing frameworks, Selenium, JIRA, Cucumber, SoapUI, Jenkins, Jython and Hudson and the use of Agile practices Kanban, Scrum, Extreme Programming. Parsons et al. (2013) mention the use of commercial testing tools, such as HP QC, HP QTP and Microsoft TFS and the use of open-source tools, such as Hudson, Jenkins and Ant. Parsons et al. found that dedicated tools were selected for testing specific technologies such as Javascript test frameworks for testing Javascript applications. Garousi & Mäntylä (2016) found that on system testing-level tools were more diverse and dependent on the application domain where as lower-level testing tools and frameworks were more similar or based on the same existing architecture or framework, such xUnit tools.

The research literature contains descriptions of test tool usage similar to the ten (10) selected models. CI tools were the most mentioned category in the reviewed models and frequently mentioned in literature, with the same tools, such as Jenkins and Hudson CI. Test case management tools were not found in the reviewed models. Keyword-driven test automation frameworks, such as Robot Framework and Cucumber were mentioned in both instances. The research literature contained mentions of multiple test tools not found in the reviewed models, such as Microsoft TFS, Ant and Jython. The popularity of Selenium tools in web-testing was mentioned (Garousi & Mäntylä 2016) in many articles.

5.3.6 How does synthesized generic model compare with Agile testing literature?

As no examples of using prescriptive modelling for test automation models were found in the researched literature, no direct comparison of the synthesized generic model attributes could be made. The synthesized generic model was designed to describe Agile test automation model elements and as such, is not a technical guide for implementing test automation and does not contain process model descriptions. However, the qualitative characteristics of the synthesized generic model could be compared to literature model examples.

The synthesized generic model is applied in the customer-facing domain which was also the most popular domain the researched literature (Sivanandan & Yogeesh 2014; Thopate & Kachewarr 2012; Nguyen et al. 2013). The generic model uses Scrum management method and Continuous Integration as development practice, which are both mentioned as the most popular choices in researched literature (Perkusich et al. 2015; Korhonen 2011). The CI tool for the generic model was Jenkins, which was mentioned in multiple articles (Hushalini et al. 2014; Parsons et al. 2013). Similar to the large number of different test frameworks in the generic model, multiple different test frameworks were used depending on the technical implementation and test level.

6. CONCLUSIONS

6.1 Research summary

Summary of the research results and the discussion of the research questions are presented in this Chapter 6.1. Critical evaluation of the conducted research is presented in Chapter 6.2. Suggestions for future research are presented in Chapter 6.3.

The objective of the research was to discover, what automation models, Agile practices and tools are found in Agile test automation literature and what kind of generic Agile test automation model can be synthesized from this literature. Two main research questions were formed after the research objective and research strategy was developed. After conducting a systematic literature review of Agile testing and test automation literature, ten test automation models were reviewed, categorized and characterized using prescriptive modelling and a generic characterization of an Agile test automation model was synthesized from the collected data.

The two main research questions were

- What test automation models, Agile practices and tools are found in Agile test automation literature?
- What kind of generic test automation model can be synthesized from Agile test automation literature?

The first research question was answered in Chapter 5.1 *Summary of Agile test automation models*. The domain of application, Agile practices, tools and characteristics of test automation models found in the researched ten articles are presented in Table 47 *Summary of Agile test automation models*. The research found ten (10) different test automation models for Agile testing, four (4) different domains of application, fourteen (14) Agile practices and twenty (20) tools. The discussion of the results affirmed that similar test automation models, Agile practices and tools were found in the researched literature.

The second research question was answered in Chapter 5.2 *Synthesized generic model*. A generic Agile test automation model was synthesized using the data collected in the first research question. The model was constructed from the most mentioned prescriptive characteristics, domain of application, Agile practices and tools. A summary of the generic Agile test automation model is presented in Table 48 *Synthesized generic Agile test automation model characteristics*. The generic model is applied in customer-facing development, using Agile practices Scrum and Continuous Integration with CI, SVC, DTS, test framework and build tools. The discussion of the generic model affirmed that the model characteristics reflected the Agile practices and tools found in the researched literature.

From the discussion of both research questions in Chapter 5.3, following points can be concluded:

- An Agile test automation model should be understood by its domain of application. Comparing models vastly different in their description, depth of detail, utility, environment, scope and domain, is futile without knowledge of their application. Understanding the different objectives and motivations of the model authors helps to understand the models.
- Prescriptive modelling was found not to be ideal for model characterization or decription. Descriptive and qualitative model characterizations capture model differences in a more insightful way. No mentions of prescriptive modelling in the researched articles.
- Continuous Integration and Scrum were found as the most popular Agile development and management practices. Continuous Integration tools were also a popular category of tools and other categories included Test frameworks, Defect Tracking Systems, Source version control tools and Build tools. Used system testing tools were more diverse and dependent on the application domain than lower-level testing tools.
- The synthesized generic model describes a typical Agile test automation model as customer-facing, utilizing Scrum and Continuous Integration Agile practices using Continuous Integration tools, Source version control tools, Defect Tracking System and Test frameworks. Other test automation models found in the researched literature contained similar characteristics.

6.2 Critical evaluation of research

When evaluating the quality of the conducted research, criticism can be applied to the search terms and vague or non-standard Agile testing definitions found in research literature, bibliographic databases, practical and methodological inclusion and exclusion criteria and prescriptive model characterizations.

Agile test automation is still comparatively young field of software testing, with Agile Testing beginning in 2009 with Lisa Crispin’s *Agile Testing*, and as such, all of the selected articles were relatively new, with oldest articles published in 2011 and newest in 2017. The keywords “Agile testing” and “Agile test automation” were too vague or too specific for the subject matter. The over-blown usage of “Agile” in testing literature and the effect it has on the search results, were not considered while performing the search. Keywords should have included “model” or “proposal” in addition to “framework”. This would have made the search results to emphasize more technical model proposals.

More bibliographic databases could have been added to the search, such as Google Scholar, to increase literature range and to reduce publisher bias. Using only Andor accessible bibliographic databases, such as Scopus and ScienceDirect limited the search results. More supporting literature could have been searched and referenced.

The practical and methodological inclusion and exclusion criteria could have been more specific to the research question. The practical inclusion criteria could have included Impact Factor to yield more academically valid results. The practical exclusion criteria could have included articles only released after 2010 for more recent results. The methodological inclusion criteria could have included more specifically that article must mention Agile practices and tools. The methodological exclusion criteria could have excluded articles with too vague model proposals. The total number of read articles and included model proposals could have been higher.

Criticism can be objected to the proposed model prescriptions. Evaluating the models proved difficult, as the practices differed widly in presentation and detail. Standardizing model elements to fit within the prescriptive modelling framework proved to be a challenge. The different domains of application was also a challenge and how to categorize them, as well as the Agile practices and tools used.

Criticism can also be applied to the source of the research literature. The researched literature used for discussing the research results contained forty (40) articles that were excluded by the methodological inclusion criteria in the first part of the bibliogprahic search. The books listed in Chapter 4.1.3 that were used as the main literary sources for the background research done in preparation to conducting the research, contain works from popular and authoritative writers, such as Crispin, Cohn and Adzic that have close connection to the Agile Testing movement.

6.3 Future research

Future research should be directed towards the frequency of use of the evaluated test automation models and how to efficiently evaluate and characterize them. Information about the real-life usage of the discussed models and their implications should be researched. Quantifiable research data about the used testing tools and Agile practices should be collected and analyzed. This means increasing the sample size and the number of used bibliographic databases.

REFERENCES

Acuña, S., de Antonio, A., Ferre, X., López, M. & Maté, L. (2001). The software process: Modelling, evaluation and improvement.

Adzic, G. (2011). Specification by Example. Manning Publications. 296 p.

Beck, K. et al. (2001). Manifesto for Agile Software Development. Available: http://agilemanifesto.org/

Chandler, D. & Munday, R. (2016). A Dictionary of Social Media. Oxford University Press.

Cohn, M. (2006). Agile Estimating & Planning. Pearson Education. 330 p.

Cohn, M. (2009). The Forgotten Layer of the Test Automation Pyramid. Mountain Goat Software. Available: https://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid

Cohn, M. (2017). Scrum Master. Mountain Goat Software. Available: https://www.mountaingoatsoftware.com/agile/scrum/roles/scrummaster

Crispin, L. & Gregory, J. (2009). Agile Testing: a Practical guide for testers and Agile teams. Addison-Wesley, 533 p.

DO-178B. (1992). Software Considerations in Airborne Systems and Equipment Certification. RTCA SC-167 & EUROCAE WG-12.

Duignan, J. (2016). A Dictionary of Business Research Methods. Oxford University Press.

Fink, A. (2014). Conducting Research Literature Reviews: From the Internet to Paper, Fourth Edition. University of California, Langley Research Institute.

Fowler, M. (2013). Given-When-Then. Accessed 14.10.2017. Available: https://martinfowler.com/bliki/GivenWhenThen.html

Gärtner, M. (2013). ATTD By Example. Addison-Wesley, 211 p.

Gregory, J. & Crispin, L. (2015). More Agile Testing: Learning journeys for whole teams. Addison-Wesley, 486 p.

Hendrickson, E. (2004). Agility for Testers. Pacific Northwest Software Quality Conference, 15 p.

IEEE Xplore, IEEE. (2017). Available: https://www.ieee.org/publications_standards/publications/periodicals/journals_magazines.html

ISTQB Glossary. (2016). Test level. Available: http://glossary.istqb.org/search/test%20level

Kusserow, A. & Groppe, S. (2014). Getting indexed by Bibliographic Databases in the Area of Computer Science. Open Journal of Web Technologies (OJWT). Volume 1, Issue 2. Institute of Information Systems (IFIS), University of Lübeck.

Leankit. (2017). What is Kanban?. Accessed 18.10.2017. Available: https://leankit.com/learn/kanban/what-is-kanban/

Marick, B. (2003). Agile testing matrix. Available: http://www.exampler.com/old-blog/2003/08/22/#agile-testing-project-2

Mitchell, J. L. & Black, R. (2015). Advanced Software Testing. Vol. 3. Second Edition. Rocky Nook, 24-26.

Ottosen, Gitte. (2016). Agile from a Testers Perspective. Capgemini Sogeti Danmark. 73 p.

Pichler, R. (2010). Agile Product Management with Scrum. Addison-Wesley, 133 p.

ScienceDirect. (2017). Browse all titles. Elsevier Inc. Available: https://www.sciencedirect.com/science/journals/all

Scopus. (2017). Content Coverage Guide. Elsevier Inc. Available: https://www.elsevier.com/__data/assets/pdf_file/0007/69451/0597-Scopus-Content-Coverage-Guide-US-LETTER-v4-HI-singles-no-ticks.pdf

Siddaway, A. (2014). What is a systematic review and how do I do one?. Available: https://www.stir.ac.uk/media/schools/management/documents/centregradresearch/How%20to%20do%20a%20systematic%20literature%20review%20and%20meta-analysis.pdf

Sogeti Nederland B. V. (2006). TMap Next: for result-driven testing. 752 p.

SpringerLink. (2017). Front Page. Springer Publishing. Available: https://link.springer.com/

Tampere University of Technology Library. (2017). Andor scientific search engine. Accessed 12.12.2017. Available: http://www.tut.fi/fi/kirjasto/ajankohtaista/andor-kaikkien-tiedonjanoisten-sankari-x158988c3

The National Center for Biotechnology Information. (2016). How to Conduct a Systematic Review: A Narrative Literature Review. Cureus. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5137994/

Trawick, B. W. & McEntyre, J. R. (2003). Bibliographic databases. National Center for Biotechnology Information, National Library of Medicine, National Institutes of Health, Bethesda.

University of Edinburgh. (2013). Systematic reviews and meta-analyses: a step-by-step guide. Centre for Cognitive Ageing and Cognitive Epidemiology. Accessed 20.12.2017. Available: http://www.ccace.ed.ac.uk/research/software-resources/systematic-reviews-and-meta-analyses

Wells, D. (2000). The Rules of Extreme Programming. Accessed 27.11.2017. Available: http://www.extremeprogramming.org/rules.html

APPENDIX A: List of research articles

1. Baseer, K. K., A Rama, M. R., & C, S. B. (2015). A systematic survey on waterfall vs. agile vs. lean process paradigms. I-Manager's Journal on Software Engineering, 9(3), 34–59.
2. Berlowski, J. A., Chrusciel, P. A., Kasprzyk, M. A., Konaniec, I. A. & Jureczko, M. B. (2016). Highly automated agile testing process: An industrial case study. E-Informatica Software Engineering Journal, 10(1), 69–87.
3. Bose, L. & Thakur, S. (2014). GRAFT: Generic & Reusable Automation Framework for agile testing. 5th International Conference - Confluence the next generation Information Technology Summit, Noida, 2014, 761–766.
4. Collins, E. F. & de Lucena, V. F. (2012). Software Test Automation practices in agile development environment: An industry experience report. 7th International Workshop on Automation of Software Test (AST), Zurich, 2012, 57–63.
5. Damm, L. & Lundberg, L. (2006). Results from introducing component-level test automation and Test-Driven Development. Journal of Systems and Software, 79(7), 1001–1014.
6. Day, P. (2014). n-Tiered Test Automation Architecture for Agile Software Systems. Procedia Computer Science, 28(2014), 332–339.
7. Elallaoui, M., Nafil, K., Touahni, R. & Messoussi, R. (2016). Automated Model Driven Testing Using AndroMDA and UML2 Testing Profile in Scrum Process. Procedia Computer Science, 83(2016), 221–228.
8. Engström, E. & Runeson, P. (2011). Software product line testing – A systematic mapping study. Information and Software Technology, 53(1), 2–13.
9. Fawad, M., Ghani, K., Shafi, M., Khan, I. A., Khattak, M. I., & Ullah, N. 2015. Assessment of quality assurance practices in Pakistani software industry. Technical Journal, 20(2), 89–94.
10. Garousi, V. & Mäntylä, M.V. (2016). When and what to automate in software testing? A multi-vocal literature review. Information and Software Technology, 76, 92–117.
11. Garousi, V., & Pfahl, D. (2016). When to automate software testing? A decision-support approach based on process simulation. Journal of Software: Evolution and Process, 28, 272–285.
12. Gary, K., Enquobahrie, A., Ibanez, L., Cheng, P., Yaniv, Z., Cleary, K., Kokoori, S., Muffih, B. & Heidenreich, J. (2011). Agile methods for open source safety-critical software. Software: Practice and Experience, 41(9), 945–962.
13. Gupta, R. K., Manikreddy, P. & GV, A. (2016). Challenges in Adapting Agile Testing in a Legacy Product. IEEE 11th International Conference on Global Software Engineering (ICGSE), Irvine, CA, 2016, 104–108.
14. Hametner, R., Winkler, D. & Zoitl, A. (2012). Agile testing concepts based on keyword-driven testing for industrial automation systems. IECON 2012 - 38th

Annual Conference on IEEE Industrial Electronics Society, Montreal, QC, Canada.

15. Hushalini, S., Randunu, R. P. A. A., Maddumahewa, R. M., & Manawadu, C. D. (2014). Software test automation in practice: Empirical study from Sri lanka. Compusoft, 3(11), 1232–1237.
16. Itkonen, J. & Mäntylä, M.V. (2014). Are test cases needed? Replicated comparison between exploratory and test-case-based software testing. Empirical Software Engineering, 19(2), 303–342.
17. Jigeesh, N., Chakraborty, S., & Chakravorty, T. (2015). An empirical study of agile testing attributes for higher customer satisfaction in IT projects in india. International Journal of Business and Information, 10(3), 365–386.
18. Kandil, P., Moussa, S., & Badr, N. (2017). Cluster-based test cases prioritization and selection technique for agile regression testing. Journal of Software: Evolution and Process, 29(6), e1794.
19. Kasurinen, J., Taipale, O., & Smolander, K. (2010). Software test automation in practice: Empirical observations. Advances in Software Engineering, 2010(4).
20. Kim E., Na J., Ryoo S. (2009). Developing a Test Automation Framework for Agile Development and Testing. Agile Processes in Software Engineering and Extreme Programming, Lecture Notes in Business Information Processing, Springer, Berlin, Heidelberg.
21. Kirmani, M. (2017). Agile methods for mobile application development: A comparative analysis. International Journal of Advanced Research in Computer Science, 8(5).
22. Korhonen, K. (2011). Evaluating the impact of agile adoption on the software defect management practices: A case study. Software Quality Professional, 14(1), 23–33.
23. Li, N., Guo, J.a, Lei, J.b, Li, Y., Rao, C.a, Cao. (2016). Towards agile testing for railway safety-critical software. ACM International Conference Proceeding Series, 24(18).
24. M. Polo, P. Reales, M. Piattini & C. Ebert. (2013). Test Automation. IEEE Software, 30(1), 84–89.
25. Mäntylä, M.V., Adams, B., Khomh, F., Engström, E. & Petersen, K. (2015). Empirical Software Engineering, 20(5), 1384–1425.
26. Mariani, L., Hao, D., Subramanyan, R. & Zhy, H. (2017). Software Quality Journal, 25(3), 797–802.
27. Mugridge, R., Utting, M., & Streader, D. (2011). Evolving web-based test automation into agile business specifications, Future Internet, 3(2), 159–174.
28. Munetoh, S. & Yoshioka, N. (2013). RAILROADMAP: An Agile Security Testing Framework for Web-application Development. IEEE Sixth International Conference on Software Testing, Verification and Validation, Luxembourg, 491–492.
29. Nguyen, B. N., Robbins, B., Banerjee, I. & Memon, A. (2014). Automated Software Engineering, 21(1), 65–105.

30. Nidagundi, P. & Novickis, L. (2017). Introducing Lean Canvas Model Adaptation in the Scrum Software Testing, Procedia Computer Science, 104 (2017), 97–103.
31. Nidagundi, P., Iela, K. & Lukjanska, M. (2014). Introduction to Lean Canvas Transformation Models and Metrics in Software Testing. Computational Methods in Social Sciences, 4(2), 23–31.
32. Parsons, D., Lal, R., & Lange, M. (2011). Test Driven Development: Advancing Knowledge by Conjecture and Confirmation. Future Internet, 3(4), 281–297.
33. Parsons, D., Susnjak, T. & Lange, M. (2014). Influences on regression testing strategies in agile software development environments. Software Quality Journal, 22(4), 717–739.
34. Perkusich, M., Gorgônio, K.C., Almeida, H. & Perkusich, A. (2016). Assisting the continuous improvement of Scrum projects using metrics and Bayesian networks. Journal of Software: Evolution and Process, 29(6).
35. Petersen, K. & Wohlin, C. (2011). Measuring the flow in lean software development. Software: Practice and Experience, 41(9), 975–996.
36. Poth, A. (2016). Effectivity and economical aspects for agile quality assurance in large enterprises. Journal of Software: Evolution and Process, 28(11), 1000–1004.
37. Prechelt, L., Schmeisky, H. & Zieris, F. (2016). Quality Experience: A Grounded Theory of Successful Agile Projects without Dedicated Testers. IEEE/ACM 38th International Conference on Software Engineering (ICSE), Austin, TX, 2016, 1017–1027.
38. Ramler, R. & Wolfmaier, K. (2006). Economic perspectives in test automation: balancing automated and manual testing with opportunity cost. Proceedings of the 2006 international workshop on Automation of software test (AST '06), ACM, New York, NY, 2006, 85–91.
39. Saleem, R. M., Qadri, S., ul Hassan, I., Bashir, R. N. & Ghafoor, Y. (2014). Testing Automation in Agile Software Development. International Journal of Innovation and Applied Studies, 9(2), 541–546.
40. Sfetsos, P. & Stamelos, I. (2011). Software quality and agile practices: A systematic literature review. Software Quality Professional, 14(1), 15–22.
41. Shim, J.-A., Kwon, H.-J., Jung, H.-J. & Hwang, M.-S. 2016. Design of acceptance test process with the application of agile development methodology. International Journal of Control and Automation, 9(2), 343–352.
42. Sivanandan, S. & Yogeesha, C. B. (2014). Agile development cycle: Approach to design an effective Model Based Testing with Behaviour driven automation framework. 20th Annual International Conference on Advanced Computing and Communications (ADCOM), Bangalore, 2014, 22–25.
43. Sivanandan, S. (2015). Fail fast - fail often: Enhancing agile methodology using dynamic regression, code bisector and code quality in Continuous Integration (CI). International Journal of Advanced Computer Research, 5(19), 220–224.

44. Thopate, H. & Kachewar, R. R. (2012). Chameleon model based automation framework design for testing in agile environments. CSI Sixth International Conference on Software Engineering (CONSEG), Indore, 1–4.
45. Tort, A., Olivé, A. & Sancho, M-R. (2011). An approach to test-driven development of conceptual schemas. Data & Knowledge Engineering, 70(12), 1088–1111.
46. Wang, F., Wang, Y. W., & Li, J. K. (2014). Agile software development using an automatic testing framework. Applied Mechanics and Materials, 543-547, 3449–3453.
47. Watkins, J. (2009). Agile Testing: How to Succeed in an Extreme Testing Environment. Cambridge University Press.
48. Woo, J., Ivezic, N. & Cho, H. (2012). Agile test framework for business-to-business interoperability. Information Systems Frontiers, 14(3), 789–808.
49. Wu, Z. Q., Li, J. Z., & Liao, Z. Z. (2013). Keyword driven automation test. Applied Mechanics and Materials, 427-429, 652–655.
50. Zhu, H., Wong, E. & Belli, F. (2008). Advancing test automation technology to meet the challenges of model-driven software development: report on the 3rd workshop on automation of software test. Proceeding ICSE Companion '08 Companion of the 30th international conference on Software engineering, 1049–1050.