



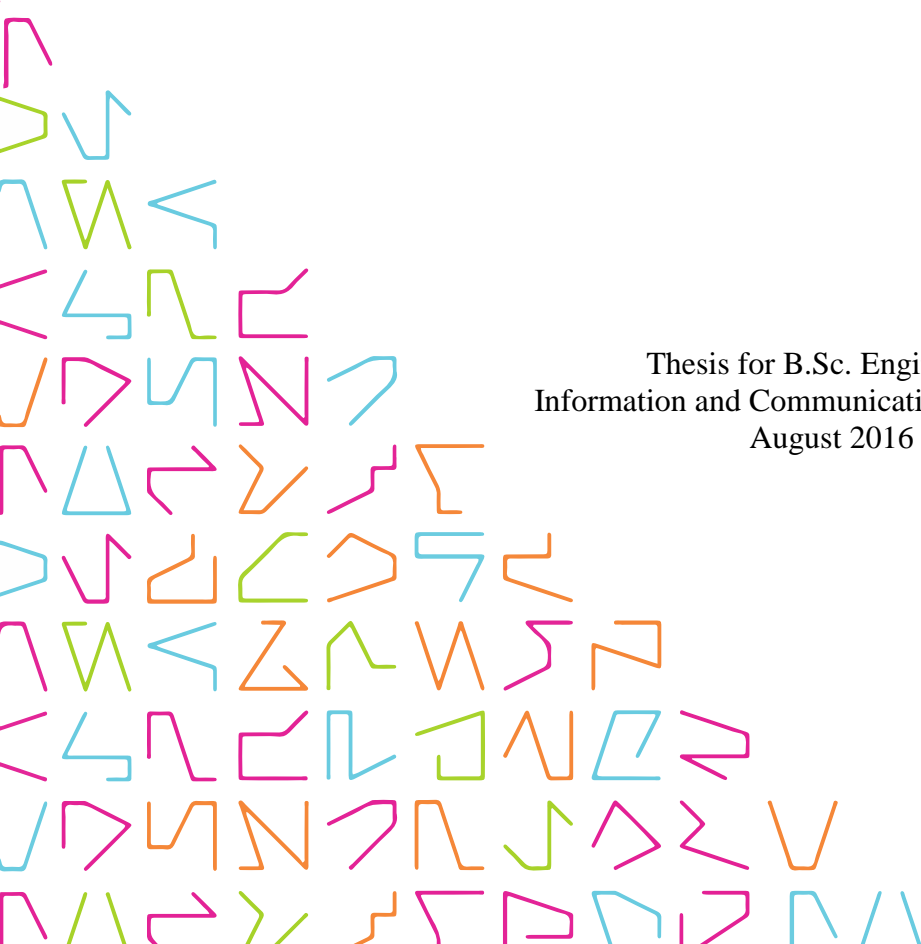
TAMPEREEN  
AMMATTIKORKEAKOULU

# IMPROVING BROWSER-BASED UI TEST AUTOMATION

Case study at Konecranes Siebel

Perttu Laamanen

Thesis for B.Sc. Engineering  
Information and Communications Technology  
August 2016



## TIIVISTELMÄ

Tampereen ammattikorkeakoulu  
Tieto- ja viestintäteknikan koulutusohjelma

PERTTU LAAMANEN:  
Improving browser-based UI test automation

Opinnäytetyö 20 sivua  
Elokuu 2016

---

Tekninen parannus on jatkuvaa ohjelmistokehityksessä. Testiautomaatio on trendikäs ja moderni laadunvarmistusprosessi. Web-applikaatioiden käyttöliittymiin keskittyvä testiautomaatio on aina ollut altis ylläpitoon liittyville haasteille. Arvaamattomat muutokset aiheuttavat usein vaikeuksia testiautomaation relevanttiudelle. Muutosten ennalta-arvaamattomuus liittyy yleensä ohjelmiston kehittämisen monivaiheisuuteen. Tämä ilmenee etenkin suurissa IT<sup>1</sup>-organisaatioissa. Tämän tutkielman innoitti testiautomaatioprojektin ylläpitotyö Konecranes Global Oy:llä.

Tutkielma perustuu ohjelmistotuotantomallin muutoksen (vesiputousmallista kanbaniin) tarkkailuun testiautomaation näkökulmasta. Tutkimuksella pyrittiin löytämään testiautomaation parannusmahdollisuuksia ja tunnistamaan tuotantomallin muutoksen vaikutuksia testiautomaatioon.

Tutkielman käsitellään perusta moniosaisen testiautomaatiorakenteen ymmärtämiseksi. Samalla selitetään kuinka testiautomaatio toimii yhteistyössä kohdeohjelmiston ja sen tuotantoprosessin kanssa. Tuotantoprosessit kuvaillaan yksityiskohtaisesti, jotta lukija saisi kunnollisen testiautomaation näkökulman tuotantomallien muutokseen.

Seuraavaksi tarkkailujakso analysoidaan ja havaitut kehitysideoita tuodaan esille. Tuloksissa selitetään tärkeät tekniset löydöt kohdeohjelmiston testiautomaation kehitystä varten. Dynaamisen testidatan huomattiin olevan merkittävä tekijä julkaisuihin liittyvän testiautomaation sopeuttamisen nopeuttamiseksi. Sopeuttamisen nopeuttaminen parantaa testiautomaation laatua ylläpidettävyyden kannalta. Kanbanista johtuva selkeämpi tuotantovaiheiden seuranta mahdollisti vakaamman ylläpidettävyyden testiautomaatiolle.

Tutkimus nosti esille merkittävän kehitysideoita testiautomaation skriptausprosessin parantamiseksi. Ajatus parannuksen taustalla on toteuttaa skriptausprosessi käyttäen grey-box testausmenetelmää.

---

Asiasanat: testiautomaatio, kanban, dynaaminen testidata

<sup>1</sup> Tietotekniikka

## ABSTRACT

Tampereen ammattikorkeakoulu  
Tampere University of Applied Sciences  
ICT Engineering

PERTTU LAAMANEN:  
Improving browser-based UI test automation

Bachelor's thesis 20 pages  
August 2016

---

Technical improvement is continuous in software development. Test automation is a trending modern quality assurance procedure. Automated testing focused on web application user interfaces has always been subject to maintainability challenges. Unpredicted changes are commonly causing complications to test automation relevance. The unpredictability of these changes is usually related to multiphasing within software development processes. This occurs especially in big IT<sup>1</sup> organizations. This thesis was inspired by test automation project management work at Konecranes Global Oy.

The thesis is based on observing software development process model change (from waterfall to kanban) from the test automation perspective. During the period of research, study was focused on finding any possible improvements aside from identifying model transition effects on the automation.

The thesis discussion starts with laying groundwork for comprehending the manifold test automation structure while also explaining how it co-operates with the target software and its development process. The development processes are described in detail to give the reader a proper test automation perspective for the process model transition.

Next the observation period is analyzed and identified improvements are expounded. Results explicate important technical findings to improve the automated testing of the target software. Dynamicity of the test data is found to be an important factor for shortening release related test automation adjustment process. Shortening the adjustment process increases test automation quality in terms of maintenance. Kanban made it easier to track development phases. This stabilized maintainability of test automation.

The study raised a major improvement idea for enhancing the test automation scripting process. Idea behind the enhancement is to implement the grey-box testing method to the scripting process.

---

Key words: test automation, kanban, dynamic test data

<sup>1</sup> Information technology

## CONTENT

1	INTRODUCTION.....	1
1.1	Research target.....	1
2	BACKGROUND.....	2
2.1	Software development process .....	2
2.1.1	Waterfall.....	2
2.1.2	Kanban .....	3
2.2	Software testing .....	3
2.2.1	Grey-box testing.....	4
2.3	Test data.....	4
2.3.1	Test data structure .....	4
2.3.2	Dynamic test data.....	6
2.3.3	Dynamic test data creation with SQL and big data.....	6
2.4	Test Automation .....	6
2.4.1	Selenium WebDriver and IDE .....	7
2.4.2	Robot Framework .....	7
2.5	Jenkins .....	9
3	METHOD.....	10
3.1	Research period.....	10
3.2	Waterfall solution .....	10
3.2.1	Release orientation.....	10
3.2.2	Test automation with waterfall.....	11
3.2.3	Test data with waterfall.....	12
3.3	Improving script processing.....	12
3.4	Transitioning to kanban .....	13
3.4.1	Kanban and test automation.....	13
3.5	Implementing dynamic test data .....	14
3.6	Defining test automation quality.....	14
4	RESULTS.....	16
4.1	Waterfall impact on test automation.....	16
4.2	Non-dynamic test data .....	16
4.3	Kanban effectiveness .....	17
4.4	Smoother flow from dynamic test data.....	17
5	CONCLUSION .....	19
	BIBLIOGRAPHY .....	20

## Terms

API	Application programming interface, a code connection point for different application components
Big data	Large amount of structured and unstructured datasets
Boolean	Programming data type with two possible values: true and false
CI	Continuous integration, process of merging code to a shared repository
Dashboard	Tool for website administration
Development process	Model used to structure, control and plan target development.
End user	The target user of an application
Framework	An abstract design that coordinates and sequences activity.
GUI	Graphical user interface, visual interface of an application
IDE	Programmer utility tool for source code editing, build automation and debugging
Input	Data entered into software by a user.
Jenkins	Tool for continuous integration
Keyword	Word that represents a single action to be done to software
Output	Data displayed by software.
Python	High-level, general-purpose programming language
Regression testing	Testing previously tested features to validate no impact from new changes.
Repository	Data structure for version controlling systems
Robot Framework	Test automation framework
Selenium	Testing framework for web applications
Slave machine	Server for automated test script execution
Software deployment	Activities of taking a system into use.
System testing	Testing phase to check the whole system's functionality.
Test automation	Automated execution of software test scripts
Test script	Steps written as an instruction to execute a test, synonymous with test case
UAT	User acceptance testing, final testing phase for end users

## 1 INTRODUCTION

This research was conducted with Konecranes Global Oy. Konecranes is a global crane manufacturing and service company. The company has around 12000 employees in 48 countries. The study was carried out together with managing and programming the Konecranes test automation project. The project team worked on Oracle's Siebel customer relationship management and field service systems. Siebel is used through a browser-based graphical user interface (GUI). Software development process for Siebel consists of following phases in a chronological order: conception, development, system testing, user acceptance testing (UAT) and publication. Automated testing is done in testing specific environments. Konecranes Siebel test automation uses keyword-driven customized Robot Framework, Selenium WebDriver and Google Chrome browser environments. Automated testing is done only on the GUI layer of the Siebel application.

### 1.1 Research target

The problems which the research tries to alleviate are how changing the software development model from waterfall to kanban affects test automation and how relevance of the test automation project results could be increased. The primary target of the research is to find out what are the impacts of software development model change on test automation. Secondary target is to enhance maintainability and stability of the project. From a scientific perspective, the research is done in order to find evident technical improvements for Siebel web client user interface (UI) test automation. The company's aspect of the research is to improve test automation relevance towards testing processes.

## 2 BACKGROUND

The following section gives short insight of the research environment and terms to be understood for research comprehension.

### 2.1 Software development process

A software development method is a model that is used to structure, control and plan system development. (CMS, 2008). This research was done comparing the waterfall model with the kanban framework. Other known methodologies and frameworks are scrum and test-driven development (TDD). In both development models there were separate physical environments for development, system testing, user acceptance testing and production. It's important to note that implementations of development models can be different between companies.

#### 2.1.1 Waterfall

The waterfall development model is a sequential phasing process where slight overlapping is acceptable. Planning and schedules are focused. (CMS, 2008). For the waterfall model in this research, the system under development is first defined by business requirements. This definition is turned into design which will be implemented into a solution. Finally the quality is tested and after publication the product will be maintained. The development cycle is visualized in figure 2.1.

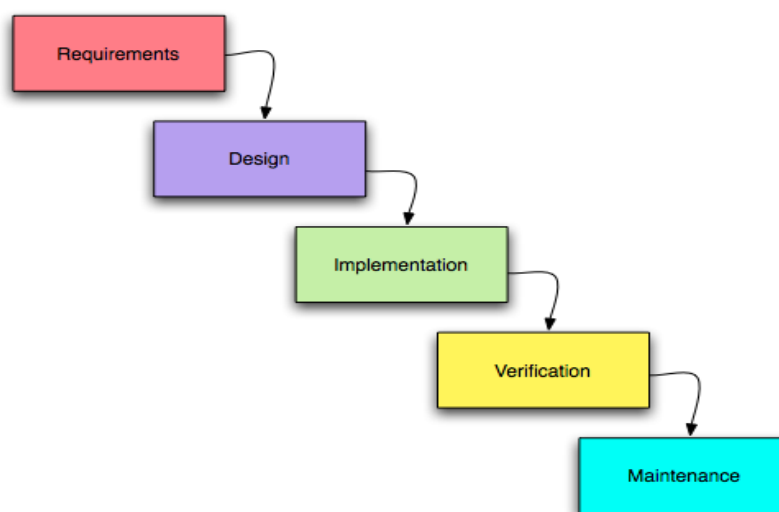


Figure 2.1: Waterfall development model. (Hoadley, 2005).

### 2.1.2 Kanban

Kanban originated from Toyota's production system. Toyota implemented a just-in-time production style to eliminate waste. Waste consists of process situations which add no value to production. (Ohno, 1988.) Kanban is a framework of lean and agile development processes. This production model was implemented to software development to achieve lean benefits – to eliminate unnecessary work. (Ohno, 1988; J. P. Womack, 2007). Siebel team's implementation of kanban was based on a physical kanban board. Ideologically visualizing the development optimizes load balancing between phases and helps target the work to correct people. (K. Scotland, n.d.). In practice the kanban board displays a to-do list for the software development team.

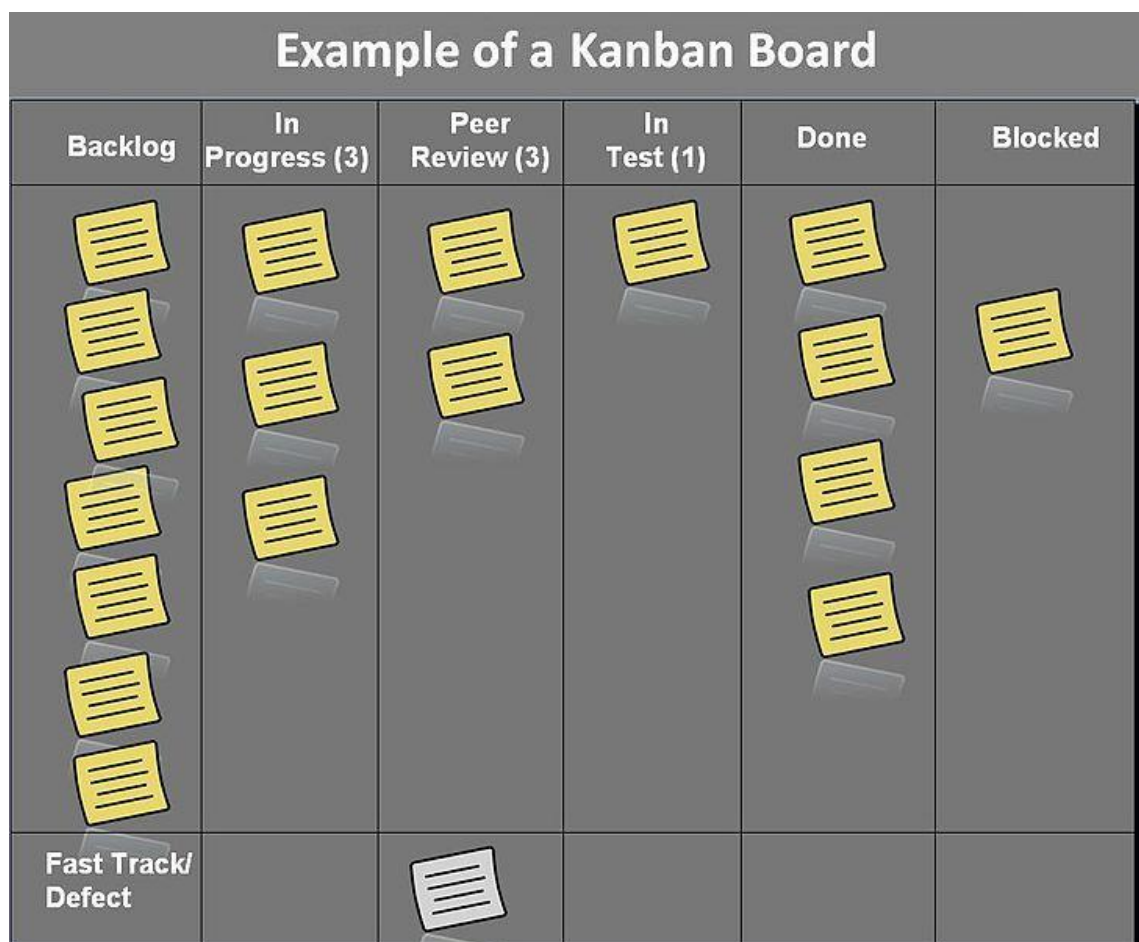


Figure 2.2: Example of a kanban board. (Mitchell, 2012).

### 2.2 Software testing

Testing is a major software development process and thus an important part of quality assurance for the stakeholders. The principle of testing is investigating a product with



predefined conditions of expected functionality. (Kaner, 2006.) Siebel testing was done on three levels. First developers tested their developments, then the team working with Siebel proceeded with system testing and finally a set of end users conducted user acceptance testing. Tests were executed from designed test scripts or synonymously test cases. These scripts were documented in a project tracking tool.

### **2.2.1 Grey-box testing**

Grey-box testing consists of black- and white-box testing. The box methods define whether or not the tester knows about the software's internal structures and hardware solutions. White-box (or glass-box) testing requires in depth knowledge whereas black-box testing only investigates external attributes and behavior. An example of a black-box testing target would be an application's expected behavior from the user's point of view. Large-scale web applications are bound to multiple hardware and software components which makes grey-box testing an essential foundation for the testing phase. (Nguyen et al., 2003.) A popular practice of the grey-box method is when a test designer is aware of the software's internal structure and the tester is not. (Kaner, 2003).

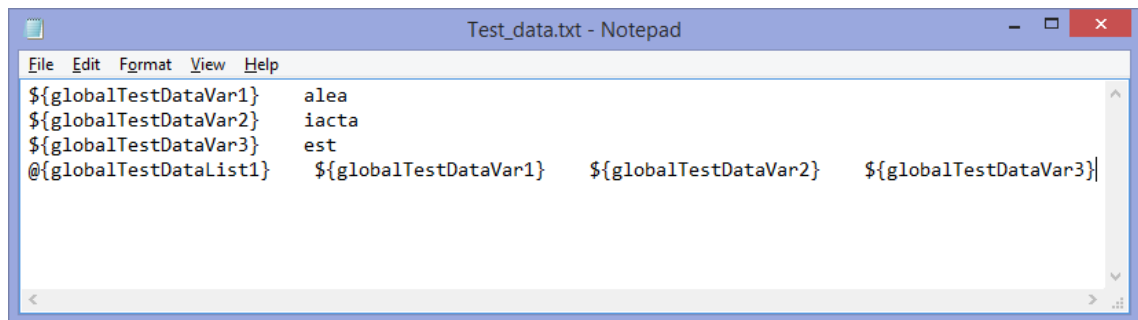
## **2.3 Test data**

Most test scripts require test data to be executed. Test data is data selected with specific test script-dependent criteria. (E. J. Weyuker, 1988). Test data is often tied to the database of the system. Common examples of test data are test user credentials. A good example of data-heavy test scripts is an end-to-end test script. In Siebel development, the user acceptance testing phase consisted of end-to-end test cases. An end-to-end case is a complete business process with precise steps and result expectations. The heaviness of test data in these test cases accumulates from the amount of inputs and output validations.

### **2.3.1 Test data structure**

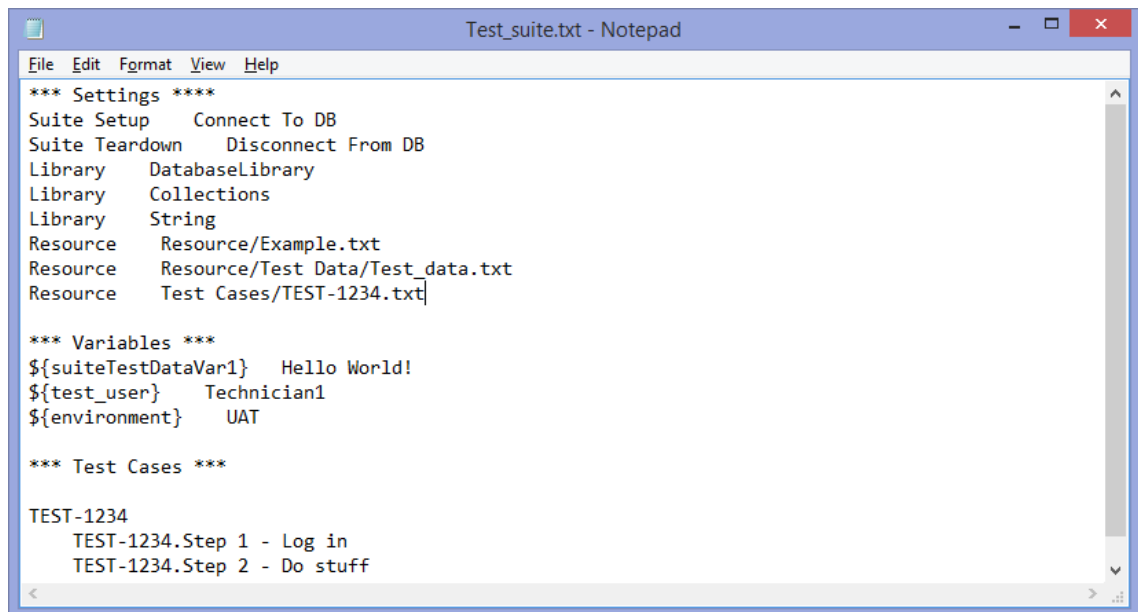
The test data structure in the research's test automation repository consisted of three layers: test data, test suite and test case files. Test data files (screenshot 2.1) were plain variable storage files with variable and list variable declarations. A common example of data stored into test data file variables is test user information which could be used for

UI data validation in the tests. Test suites (screenshot 2.2) were configuration files for sets of tests that required the same settings or variables like test users and resource files. Test suite files were used as the executable scripts that called resource files and test cases. This created a virtual network between the three layers. Test case files (screenshot 2.3) contained the actual test scripts in keywords. Test suite, data and case files were all written in Robot Framework syntax.



```
Test_data.txt - Notepad
File Edit Format View Help
${globalTestDataVar1}    alea
${globalTestDataVar2}    iacta
${globalTestDataVar3}    est
@[globalTestDataList1]    ${globalTestDataVar1}    ${globalTestDataVar2}    ${globalTestDataVar3}
```

Screenshot 2.1: Test data file example.



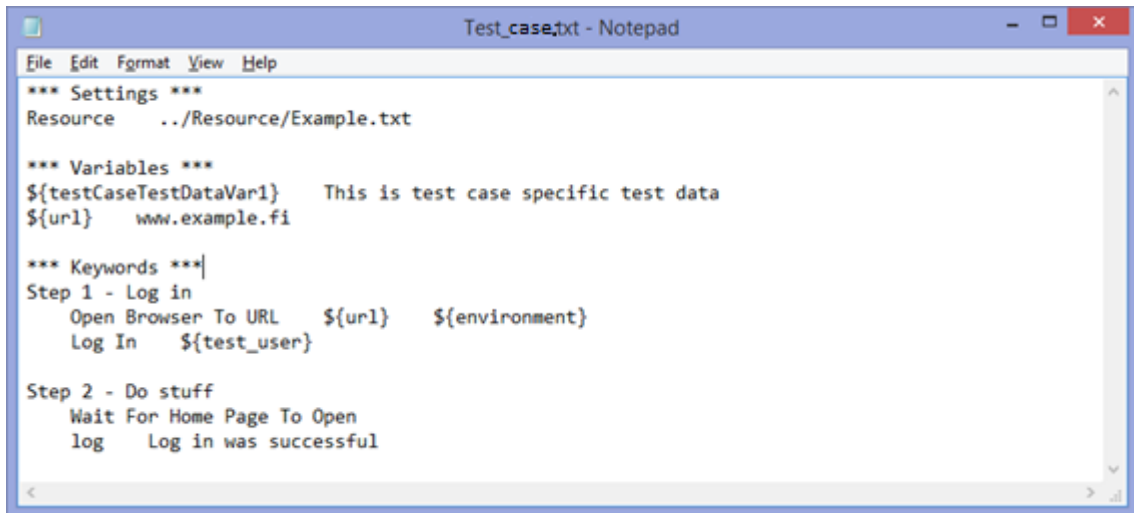
```
Test_suite.txt - Notepad
File Edit Format View Help
*** Settings ****
Suite Setup    Connect To DB
Suite Teardown    Disconnect From DB
Library    DatabaseLibrary
Library    Collections
Library    String
Resource    Resource/Example.txt
Resource    Resource/Test Data/Test_data.txt
Resource    Test Cases/TEST-1234.txt

*** Variables ***
${suiteTestDataVar1}    Hello World!
${test_user}    Technician1
${environment}    UAT

*** Test Cases ***

TEST-1234
    TEST-1234.Step 1 - Log in
    TEST-1234.Step 2 - Do stuff
```

Screenshot 2.2: Test suite file example.



```

Test_case.txt - Notepad
File Edit Format View Help
*** Settings ***
Resource    ../Resource/Example.txt

*** Variables ***
${testCaseTestDataVar1}    This is test case specific test data
${url}    www.example.fi

*** Keywords ***
Step 1 - Log in
    Open Browser To URL    ${url}    ${environment}
    Log In    ${test_user}

Step 2 - Do stuff
    Wait For Home Page To Open
    log    Log in was successful

```

Screenshot 2.3: Test case file example.

### 2.3.2 Dynamic test data

Manual test data generation is one of the factors slowing down software testing. Test data is mostly generated manually and with a black-box approach, the test designer may need to go deep into the system to define it. Dynamic test data generation is the process of automatically identifying input data which satisfies the selected testing criterion. (K. Bogdan, 1990.) In Siebel, preset dynamic test data was generated automatically from the system's database. Dynamic test data was also generated during the execution of test cases in for example situations where calculation algorithms were verified.

### 2.3.3 Dynamic test data creation with SQL and big data

Big data is a term used for large amounts of structured and unstructured datasets. Big data software is a challenging environment for analyzing, querying and maintaining data. (J. M. Cavanillas et al., 2016.) The customized Siebel in this research was a very data-heavy environment. Dynamic test data was generated with complex SQL queries that were run against the Siebel database. SQL stands for "structured query language" which is a syntax designed for database management.

## 2.4 Test Automation

Reducing costs and doing more is always a target for improvement in software development. For this reason, automating tests with external tools has become a trendy way

of facilitating the testing teams' workload. To run an automated test, a test script based on functional specifications is scripted for an external tool to execute on the software under development. (Dustin et al., 2008.) The possibility of constant repetitive testing is an additional benefit of test automation. Automated tests can be triggered by continuous integration (CI) commits. CI means merging development work into a shared main repository. (Fowler & Foemmel, 2006).

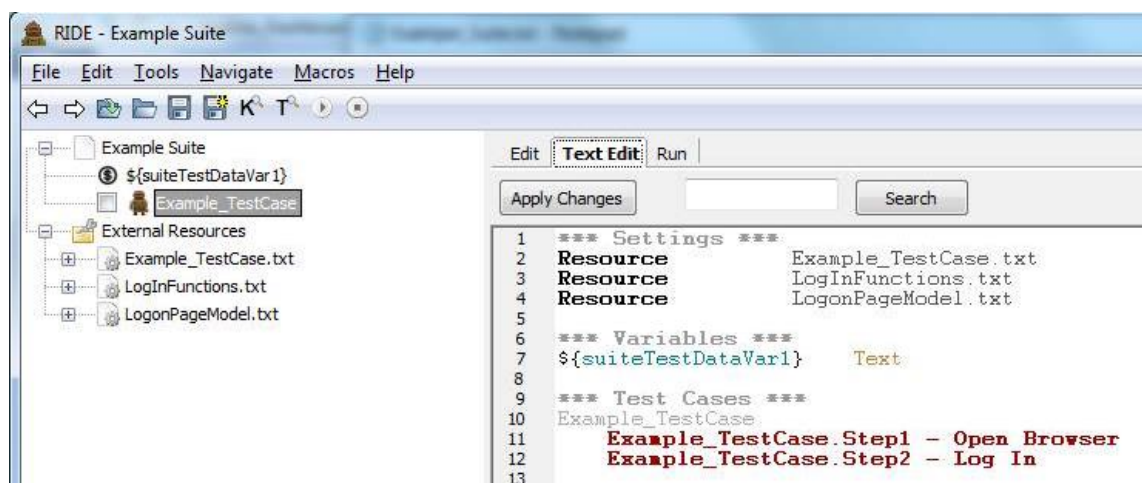
#### **2.4.1 Selenium WebDriver and IDE**

Selenium WebDriver is a browser controller application programming interface (API) which allows different programming environments to control a browser. Sending a command to Selenium WebDriver e.g. a button click event will trigger it in a browser instance controlled by Selenium. The Selenium integrated development environment (IDE) is a Mozilla Firefox plugin that allows the user to record actions on the browser instance and then run it as an automated script generated with Selenium. These scripts can be exported in different formats. (The Architecture of Open Source Applications: Selenium WebDriver, n.d.)

#### **2.4.2 Robot Framework**

Robot Framework is a keyword-driven python-based generic framework. (Robot Framework Homepage, n.d.). Siebel test automation was set up with Robot Framework and Selenium WebDriver. A picture of the Robot Framework IDE can be found from screenshot 2.3. Selenium provides keyword libraries for Robot Framework. These libraries contain basic functions for processing a browser instance. Keyword-driven syntax is a programming framework where functions are called with actual words. (D. R. Faught, 2004). An example of a keyword in this case would be: "Lists Should Be Equal" which calls lower layer functions in a structure that eventually leads to a Python library. Functionally the example keyword would take two list variables as parameters and compare them to output a Boolean result. Keywords were used in multiple layers. An example of the layers would be a keyword of "Open tab X" that contained lower layer functions "Click Link To X" and "Wait For X To Appear". Robot Framework has its own simplified syntax which is interpreted to Python in execution.

The Robot Framework structure in the research test automation was built from standard and custom Python libraries. These libraries compiled into standard and custom keywords that were mixed together into higher keyword layers. Customized keywords were high layer entities consisting of keywords from the standard libraries. Higher layer keywords were used to complete longer user interface tasks with less scripting. A customized Python library was created for moving some customized keywords to a lower code layer. Relationship of the libraries and keywords under the syntaxes is displayed on figure 2.4. Robot Framework automatically generates logs after finishing a script. These logs present the outcome of test scripts and their steps all the way to the lowest keyword layer.



Screenshot 2.3: Robot Framework IDE (RIDE).

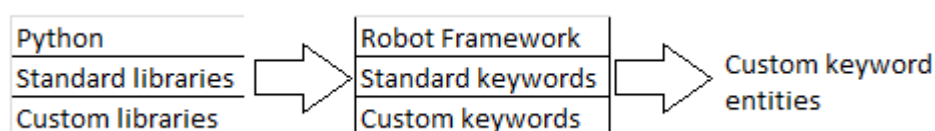
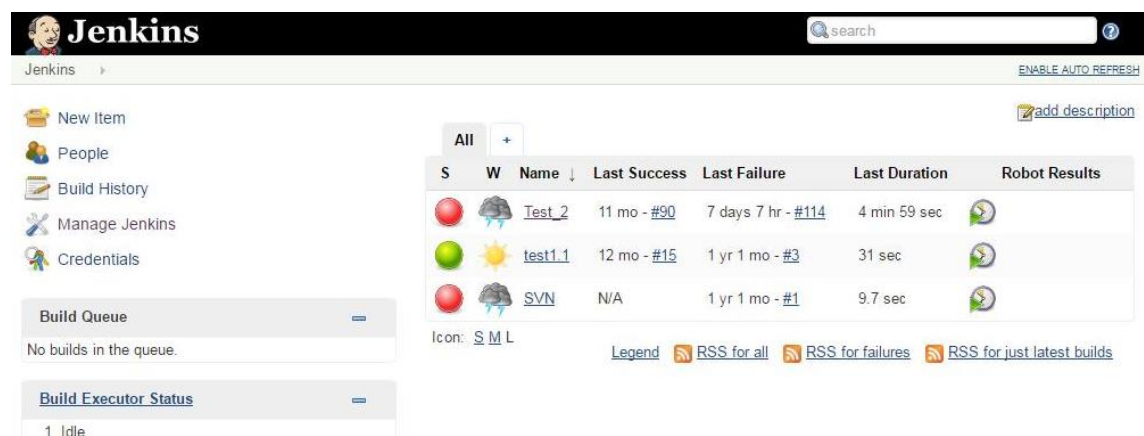


Figure 2.4: Syntaxes, libraries, keywords and their relations.

## 2.5 Jenkins

Automated testing under research was maintained with Jenkins – a continuous integration system. Jenkins is a panel that controls script execution on slave machines. (Distributed builds, n.d.). For Siebel test automation, Jenkins was used to run Robot Framework scripts with batch commands. The test automation team used Jenkins to automatically run all test suites on 5 slave machines once every day. Additionally, a Powershell script was used to compile daily results into a custom website (screenshot 3.1) that was accessible from the Jenkins dashboard. An example of the Jenkins dashboard is displayed in screenshot 2.5.



The screenshot shows the Jenkins dashboard interface. At the top, there is a search bar and a navigation menu. The main content area displays a table of build results. The table has columns for 'S' (Status), 'W' (Weather icon), 'Name', 'Last Success', 'Last Failure', 'Last Duration', and 'Robot Results'. The 'Build Queue' section shows 'No builds in the queue.' and the 'Build Executor Status' section shows '1 Idle'.

S	W	Name ↓	Last Success	Last Failure	Last Duration	Robot Results
		<a href="#">Test_2</a>	11 mo - #90	7 days 7 hr - #114	4 min 59 sec	
		<a href="#">test1.1</a>	12 mo - #15	1 yr 1 mo - #3	31 sec	
		<a href="#">SVN</a>	N/A	1 yr 1 mo - #1	9.7 sec	

Icon: [S](#) [M](#) [L](#)

Legend [RSS for all](#) [RSS for failures](#) [RSS for just latest builds](#)

Screenshot 2.5: Jenkins dashboard.

### **3 METHOD**

This section is a detailed review of what was done to observe and improve test automation quality. Research was conducted during a year in a test automation project team. The team consisted of the researcher and vendors or summer trainees. Work was done in the company headquarters and by working remotely from home office. The primary target of the research was to find what the impacts of software development model change are on Siebel's test automation. The secondary research target was to find improvement ideas for the project.

#### **3.1 Research period**

The research started in late summer 2015 and ended in August 2016. A waterfall model was in use from the beginning of the research until spring 2016. Transitioning to kanban and dynamic test data generation started during spring 2016.

#### **3.2 Waterfall solution**

The first software development process used during the research was the waterfall model. The model's process cycle started with specifying a release with a set of new development items. First these items went from backlog to concept design. From design they went to the first physical environment for development. After development they would undergo system testing and user acceptance testing (UAT) in their own environments. Finally the release went to production.

##### **3.2.1 Release orientation**

During the rounded 8-month period of the waterfall solution, the system was upgraded with two releases as per scheduled by business. The big, release-oriented software development model caused overload in some development phases due to large item amounts moving at the same time. Major releases required full regression test cycles in all testing phases, as test automation was not yet reliable enough to run them alone. Running the regression test cycles manually is considered impractical. (E. Dustin, 2002).

### 3.2.2 Test automation with waterfall

Automated tests were initially executed in the environments for development, system testing and UAT. The development environment was removed from test automation scope because of instability and the possibility of being recompiled anytime by developers. Test automation version management was structured with release-dependent repositories. After items were moved from development to system testing, new item-related automated test scripts failed. To have the tests passing again, scripts required adjustments to the new release within its dedicated test automation repository. Due to the nature of the waterfall model, bulks of items were moved at the same time causing big leaps in automated test pass rate percentages. An impact analysis of the new release's effects on test automation was done with the testing manager. An approximate description of release challenges in test automation is depicted in figure 3.1.

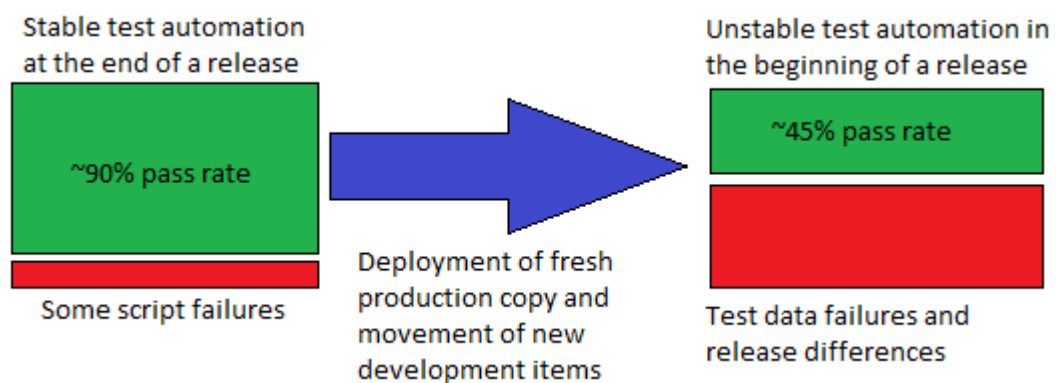


Figure 3.1: Waterfall deployment and test automation results.



### 3.2.3 Test data with waterfall

Siebel test automation used lots of test data that was dependent on the release. Testing environments were deployed with the latest copy of production one week before the start of testing phases. Hardcoded test data had to be manually changed every time the database was replaced. The structure of the test automation repository allowed usage of global test data variables that reduced the maintenance work.

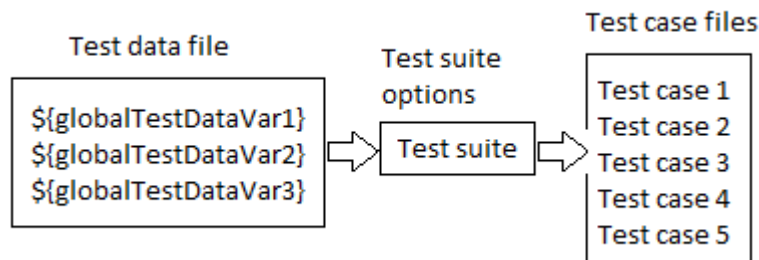


Figure 3.2: Global variables routed to test case files through a test suite (see chapter 2.3.1 for details on test data structure).

### 3.3 Improving script processing

In the beginning of 2016, the test automation team decided to improve the performance of the automation. Some of the core customized keywords created from standard Robot Framework keywords were programmed into a customized Python library. In other words core functions were implemented to a lower code layer. Executing the functions on a lower layer was expected to improve computing speed as the new code didn't have to be interpreted. An example of this is shown in figure 3.3.

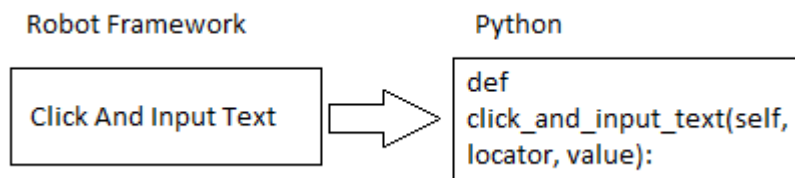


Figure 3.3: Keyword movement to lower codebase.

### **3.4 Transitioning to kanban**

In spring 2016, the company decided to change its information technology (IT) organization's development approach to kanban. In practice the new kanban approach meant monthly releases of small items that were compiled to 3 annual production releases. A physical kanban board was set up in a meeting room dedicated for kanban. Meetings were held twice a week to address progress and issues regarding the items on the board. At the end of the research, a web camera was going to be installed to present the kanban board for online meeting participants.

#### **3.4.1 Kanban and test automation**

With the kanban method, development items were flowing through the development phases continuously. Testing environments would only be taken down for fresh database deployments. New development items could be followed on the kanban board. This simplified the work required to track down root causes for pass rate decreases. It was decided that system testing environment would be deployed with a fresh UAT database copy instead of one from production. This resulted in integrity of test data in both of its working environments. Regression testing was reduced due to test automation improvements and simplified flow of single development items. Reduction of regression testing highlighted the importance of test automation stability.

### 3.5 Implementing dynamic test data

During spring 2016, the team decided to implement dynamic test data within the automated test scripts because manual adjustment after every fresh database deployment to environments was a long process. Automated test data generation was done by fetching the data with SQL queries through a direct connection with the environment's database. It was implemented in the Robot Framework scripts in a way that manual input was no longer required to get fresh test data for test cases. Figure 3.4 shows the process flow of automatic test data generation. For some test automation solutions, test data must always be set manually. An example would be when the automated test script requires integration data from another application. Test automation of this research was originally created to be well maintainable – thus ruling out data-dependent integration test scripts. After observing a release transitioning in the beginning of year 2016, it was noted that most of the work spent on adjustment was actually from setting up the test data for the new environment.

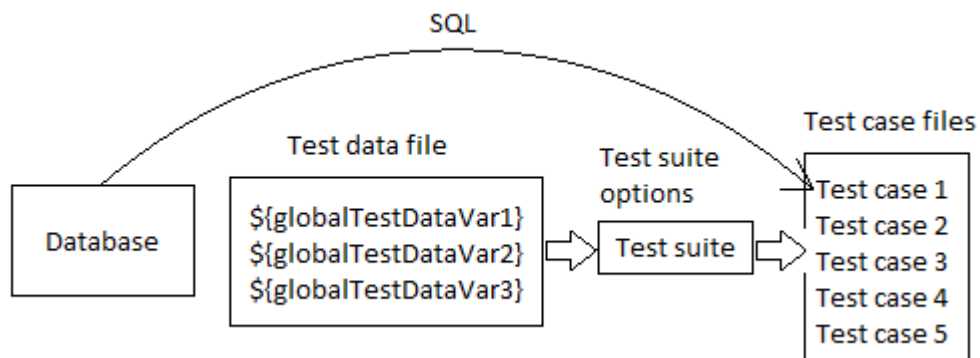


Figure 3.4: Fetching test data from the database.

### 3.6 Defining test automation quality

Before analyzing the results, it is necessary to define what test automation quality in this case of study is. It is important to understand that combinations of test data generation solutions and software development models have their individual optimums depending on the test automation target. Expectations for the test automation in this research were to observe the state of testing environments, monitor performance, save resources in testing phases and point out valid deployment issues. To meet the expectations, the automated tests were run in scheduled Jenkins batch commands which logged results to a

dashboard. The dashboard was created to monitor test automation quality. An example of the result dashboard is shown in screenshot 3.1.

## Test automation status

Last updated on date & time

Vendor #1		System test	
	Total	Pass	Pass rate
Regression run	94	75	80%
Performance run	12	12	100%
<b>Total</b>	106	87	82%

\*Latest log is from: date & time

Screenshot 3.1: Dashboard for test automation results.

For Siebel, test automation quality was defined by a stable high pass rate, quick recoverability, flexible adjusting and good maintainability. The quality definition was limited by the software's nature of being a data-heavy web UI-based CRM system. A constantly high overall-pass rate represented a good readiness for analyzing new deployment operability. An indicator of quick recovery is a situation where a code structure that causes sudden pass rate decreases can be easily identified and fixed. Quality in terms of flexible adjusting referred to the simplicity of automating test scripts for new features. All the previous quality factors in addition to proper development and monitoring tools were a part of Siebel test automation project's good maintainability.

## 4 RESULTS

Software development model change showed a stabilizing effect on test automation maintainability. Implementation of dynamic test data shortened test automation adjustment process. For both process and technical structure, positive outcome was identified throughout the implementation and changes done within the research observation scope.

### 4.1 Waterfall impact on test automation

Adjusting automated scripts to the new deployments was slightly confusing with the waterfall model. Deploying a set of development items during a short period of time made it hard to analyze what the root causes behind pass rate decreases were due to possibility of having simultaneous scripting failures, test data mismatches and actual release indifferences. A good example was when a database copy was deployed to one of the testing environments and new release items were moved right after the deployment. In this situation, the non-dynamic test data would conflict with the fresh database copy causing script failures while the new development items would affect the pass rate in a similar manner. Identification of root causes would take longer as they weren't predictable. Longer development phases allowed longer adjustment periods for bringing automated scripts up to date with the new release. This led to nearly perfect pass rates by the end of testing phases which were very useful for validating hot fixes. Hot fixes were made for bugs that were found after production deployment and tested in the UAT environment.

### 4.2 Non-dynamic test data

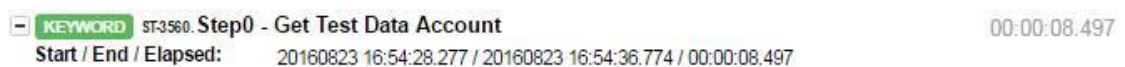
Automated scripts used non-dynamic test data for the whole period of the waterfall model. Fresh database copies from production to the testing environments always mismatched most of the manually set test data. Manual inputs are not practical in test automation maintenance, but in some situations they may be necessary. In this case all test data requirements needed to be checked from an original archived test script and with those requirements it was searched from the system. After suitable test data was found, it was edited into a test data script file. This process was time-consuming and complex for a new employee.

### 4.3 Kanban effectiveness

Transitioning to kanban was helpful for the test automation team. Smaller items going through testing environments one by one made it much easier to sustain test automation stability. Maintenance work could be prioritized since pass rate decreases were more predictable this way. The new item movements from development to testing environments were visualized on the kanban board. Tracking these movements increased the predictability of test script failures. Kanban development flow was supposedly more occupied than development with the waterfall model. The new flow of work meant less time and people for running regression tests. The flow together with improvements of the test automation project led to reduced manual regression testing. Reduction of regression testing saved time and raised the importance of test automation.

### 4.4 Smoother flow from dynamic test data

Implementing dynamic test data lead to a much faster and smoother flow of test automation adjustment. After a portion of test data was set to be fetched with SQL in the beginning of automated test scripts, the automation partly adjusted itself to new database deployments. Creating the SQL scripts to fetch the test data was a challenging operation because of Siebel's massive database and its complexity with out-of-the-box and customized tables mixed together. Creation of the dynamic test data proved to be worth the effort during the summer of 2016, as a lot of time was saved in manual test data set-up. Dynamic test data was a major factor in the test automation stability which is important in any successful kanban implementation. Test data type comparison is shown on table 4.1. Scripts for getting dynamic test data had no major impact on test automation performance. Dynamic test data script can be seen in screenshot 4.1 where it is displayed as a test case step on a log generated by Robot Framework. This example took only 8 seconds to execute.



KEYWORD ST-3580.Step0 - Get Test Data Account 00:00:08.497  
 Start / End / Elapsed: 20160823 16:54:28.277 / 20160823 16:54:36.774 / 00:00:08.497

Screenshot 4.1: Test case step for getting test data.

Table 4.1: Dynamic test data versus non-dynamic test data.

Setup	Dynamic test data	Non-dynamic test data
Process length	Short	Long
Description	Create SQL query, create script to execute SQL on test case, copy and paste script to test cases	Define test data from test script, obtain test data through UI, set test data into test data files
Frequency	Once	Every release
Pros	Test data script creation is a one-time operation	Test data is as good as configured to be
Cons	Database may contain bad data	Time-consuming operation on every release

## 5 CONCLUSION

The type of test automation used in this research was more suitable to approach with a kanban model than a waterfall model because of simpler adjustment workflow. Less work was needed to maintain the scripts and the results were more stable. In other words the research was beneficial from the introductory company aspect, as test automation relevance towards testing increased. The increased relevance allowed testing costs to be reduced. For the initial scientific perspective, technical improvements were identified and implemented. These technical improvements were a good solution for particularly this test automation project and they may not be implementable to other web UI test automation projects. This can be considered a limitation of this study. This study could be continued by researching software development model impact on test automation with different models and test automation structures.

Making the test data dynamic was a key operation to increase maintainability. This improvement was left unfinished during the research, but it will definitely be completed in the future. Most test data can be generated dynamically though some situations may require very technical solutions.

As for finding other improvements for the scientific research perspective, ideas were composed by identifying the durations of different test automation processes. The creation of new scripts is one of the slower processes along with setting up manual test data. This study raised an idea of optimizing the scripting process by implementing a grey-box method to browser UI test automation. In practice it would be done by having black-box testers record their test cases with Selenium IDE and exporting the results using a formatter. The exported files would then be sent to a white-box participant like a test automation developer who then configures the test cases into a new or existing test suite. Implementation of this new process gives ground for research continuation.

Although the results of this research were positive, some technical improvements could have been taken into consideration when the test automation structure was originally designed. From the beginning, dynamic test data could have been a requirement and the structure could have been programmed mostly to the Python level.



## BIBLIOGRAPHY

C. Kaner. Exploratory Testing, 2006. URL <http://www.kaner.com/pdfs/ETatQAI.pdf>. April 23, 2016.

Centers for Medicare & Medicaid Services (CMS) Office of Information Service, 2008. URL <http://www.cms.gov/Research-Statistics-Data-and-Systems/CMS-Information-Technology/XLC/Downloads/SelectingDevelopmentApproach.pdf>. August 25, 2016.

T. Ohno, 1988. Toyota Production System: Beyond Large-Scale Production, pages 1-2.

J. P. Womack, 2007. The Machine That Changed The World, pages 48-51.

K. Scotland, Aspects of Kanban. URL <http://www.methodsandtools.com/archive/archive.php?id=104>. August 25, 2016.

E. J. Weyuker, 1988. The evolution of program-based software test data adequacy criteria. Abstract, pages 1-2.

H. Q. Nguyen, B. Johnson, M. Hackett, 2003. Testing Applications on the Web: Test Planning for Mobile and Internet-Based Systems, pages 20-22, 7,

E. Dustin, J. Rashka, J. Paul, 2008. Automated Software Testing: Introduction, Management and Performance, pages 3-4.

P. A. Hoadley, 2005. URL [https://upload.wikimedia.org/wikipedia/commons/5/51/Waterfall\\_model.png](https://upload.wikimedia.org/wikipedia/commons/5/51/Waterfall_model.png). August 21, 2016.

I. Mitchell, 2012. Example of a Kanban board. URL [https://upload.wikimedia.org/wikipedia/commons/thumb/f/f5/Kanban\\_board\\_example.jpg/726px-Kanban\\_board\\_example.jpg](https://upload.wikimedia.org/wikipedia/commons/thumb/f/f5/Kanban_board_example.jpg/726px-Kanban_board_example.jpg). August 21, 2016.

M. Fowler, M. Foemmel. Continuous integration, 2006. URL [http://www.dccia.ua.es/dccia/inf/asignaturas/MADS/2013-14/lecturas/10\\_Fowler\\_Continuous\\_Integration.pdf](http://www.dccia.ua.es/dccia/inf/asignaturas/MADS/2013-14/lecturas/10_Fowler_Continuous_Integration.pdf). April 25, 2016.

K. Bogdan, 1990. A Dynamic Approach of Automated Test Data Generation, page 1.

J. M. Cavanillas, E. Curry, W. Wahlster, 2016. New Horizons for a Data-Driven Economy, A Roadmap for Usage and Exploitation of Big Data in Europe, pages 3-5.

The Architecture of Open Source Applications: Seleniums WebDriver. URL <http://www.aosabook.org/en/selenium.html>. August 25, 2016.

Robot Framework Homepage. URL <http://robotframework.org/>. August 25, 2016.

D. R. Faught, 2004. Keyword-Driven Testing. URL <https://www.stickyminds.com/article/keyword-driven-testing>. August 25, 2016.

Distributed builds, Jenkins website. URL <https://wiki.jenkins-ci.org/display/JENKINS/Distributed+builds>. August 25,2016

E. Dustin, 2002. Effective Software Testing, 50 Specific Ways to Improve Your Testing. Item 39: Automate Regression Tests When Feasible.