Santeri Vaara

Client Software for Visualizing Test Automation Result

Metropolia University of Applied Sciences

Bachelor of Engineering

Information and Communications Technology

Thesis

7 September 2018

Author Title Number of Pages Date	Santeri Vaara Client Software for Visualizing Test Automation Result 47 pages 7 September 2018
Degree	Bachelor of Engineering
Degree Programme	Information and Communication Technology
Professional Major	Smart Systems
Instructors	Juhana Sillanpää, squad group leader Hannu Markkanen, researcher teacher

This bachelor’s thesis documents the development of client software as a part of a new test analysis tool. The client software includes communication with the server for fetching data and a graphical user interface for visualizing it. This project was conducted for a Finnish telecommunications company operating globally.

As a starting point, software builds are tested daily with regression testing for ensuring that software works the same way as it did before changes. The tests are made with Robot Framework and they are executed in a Jenkins server. Jenkins server is used for continuous integration, which enables test automation. After executing tests, the test results are seen in a Jenkins build’s web-page with help of Robot Framework plugin. There are multiple Jenkins builds executing thousands of tests daily. The tester's job is to analyze the failed tests and to ensure that test automation works. In addition to Jenkins web-page, the test results are stored into a data storage server. Storage server contains over a year of unused test result data.

The purpose of this thesis was to develop a client software for visualizing the test result data from storage server. Client software is part of a new tool, which has got a server software and a data manager software. Java was chosen as a programming language and JavaFX for developing the graphical user interface. JavaFX is a Java software platform generally used for creating computer desktop applications. JavaFX has a declarative way of coding user interface elements similarly to hypertext markup language. This declarative way was widely used in development of the graphical user interface. JavaFX offers lots of ready-made classes for visualizing data. Using these classes was essential for visualizing test result data with different charts and tables. User interface components could also be created with a program called Scene Builder. Scene Builder enabled building the user interface by dragging and dropping components from component menu.

As a result of the thesis, end-users can see summary of software build’s test results, statistics and tests' history. Each of these features were created into the client software interface. In addition to these features, client software supports other tools used in tester’s daily work and it is able to retrieve data from the server.

Keywords	Continuous Integration, Continuous Delivery, Jenkins, Robot Framework, Test automation, Java, JavaFX, Graphical User Interface, Client

Tekijä Otsikko Sivumäärä Aika	Santeri Vaara Asiakasohjelmisto testiautomaatiotulosten visualisointiin 47 sivua 7.9.2018
Tutkinto	insinööri (AMK)
Tutkinto-ohjelma	tieto- ja viestintätekniikka
Ammatillinen pääaine	Smart Systems
Ohjaajat	esimies Juhana Sillanpää tutkijaopettaja Hannu Markkanen

Insinöörityössä kehitettiin asiakasohjelmisto, joka toimii osana uutta testien analysointityökalua. Asiakasohjelmisto sisältää tiedonsiirron palvelimelta datan hakemiseen ja graafisen käyttöliittymän datan visualisointiin. Projekti toteutettiin maanlaajuisesti toimivalle suomalaiselle televiestintäyritykselle.

Insinöörityön tilaajayrityksessä ohjelmistopaketit testataan päivittäin regressiotesteillä varmistamaan, että ohjelmisto toimii samalla tavalla kuin ennen muutoksia. Testit tehdään käyttäen Robot Frameworkia, ja ne suoritetaan Jenkins-palvelimella. Jenkins-palvelinta käytetään jatkuvaan integraatioon, mikä mahdollistaa testiautomaation. Testien suoritusten jälkeen Robot Framework -liitännäinen mahdollistaa tulosten näkyvyyden Jenkinsin verkkosivulla. Jenkins suorittaa tuhansia testejä päivittäin, ja testaajan työ on analysoida epäonnistuneet testit sekä varmistaa, että testiautomaatio toimii. Jenkins-verkkosivun lisäksi testien tulokset tallennetaan tiedontallennuspalvelimelle. Tallennuspalvelin sisältää yli vuoden verran dataa, jota ei käytetä mihinkään.

Insinöörityön tarkoituksena oli kehittää asiakasohjelmisto, joka havainnollistaa tiedontallennuspalvelimella olevat testitulokset. Asiakasohjelmisto on osana uutta työkalua, jossa on olemassa palvelinohjelmisto ja tiedonhallintaohjelmisto. Projektin ohjelmointikieleksi valittiin Java, ja graafisen käyttöliittymän kehittämiseksi valittiin JavaFX. JavaFX on Java-ohjelmistoalusta, jolla voidaan luoda työpöytäsovelluksia. JavaFX:llä voidaan koodata työpöytäsovellusten käyttöliittymiä deklaratiivisella tavalla, joka on samanlainen kuin hypertekstimerkintäkieli. Tätä deklaratiivista tapaa käytettiin laajalti projektin käyttöliittymän kehittämisessä. JavaFX tarjoaa paljon valmiita luokkia tiedon visualisointiin. Näitä luokkia käyttäen pystyttiin visualisoimaan testitulosten tietoja erilaisilla kaavioilla ja näkymillä. Käyttöliittymäkomponentteja pystyttiin myös luomaan Scene Builder -nimisellä ohjelmalla. Tällä ohjelmalla pystyi graafisesti rakentamaan käyttöliittymänäkymiä raahaamalla ja pudottamalla komponentteja ohjelman komponenttivalikosta. Scene Builder -ohjelma kirjoitti automaattisesti siinä tehdyt muutokset deklaratiiviseen muotoon käytetyssä tiedostossa.

Opinnäytetyön tuloksena asiakasohjelmiston loppukäyttäjät voivat tarkastella yhteenvetoa ohjelmistopaketin testituloksista, tilastoista ja testien historiasta. Jokaiselle näille toiminnoille luotiin näkymä asiakasohjelmiston käyttöliittymässä. Lisäksi asiakasohjelmisto tukee muita käytettyjä työkaluja ja osaa hakea haluttuja tietoja palvelimelta.

Avainsanat	jatkuva integraatio, jatkuva toimitus, Jenkins, Robot Framework, testausautomaatio, Java, JavaFX, graafinen käyttöliittymä, asiakasohjelmisto

Contents

List of Abbreviations

PCI	Product Continuous Integration
CI	Continuous Integration
CD	Continuous Delivery
SCM	Source Control Management
GUI	Graphical User Interface
URL	Uniform Resource Locator
ATDD	Acceptance Test-Driven Development
API	Application Programming Interface
OS	Operating System
XML	Extensible Markup Language
JVM	Java Virtual Machine
RAM	Random-access memory
OOP	Object-Oriented Programming
IDE	Integrated Development Environment
JDK	Java development kit
JRE	Java Runtime Environment

RIA	Rich Internet Applications
AWT	Abstract Window Toolkit
JFC	Java Foundation Classes
CSS	Cascading style sheet
SDK	Software development kit
HTML	Hypertext Markup Language
POM	Project Object Model
TGZ	Tar Gnu Zip
TCP	Transmission Control Protocol
UDP	Unix Datagram Protocol

1 Introduction

This bachelor’s thesis focuses on the development of client software for a new test analysis tool. The tool will be used for analyzing and visualizing software test results. The thesis project is conducted for a Product Continuous Integration (PCI) team in Nokia Solutions and Networks. Nokia Networks is a business unit of Nokia Corporation that designs and manufactures equipment and software for telecommunications networks. PCI team’s main responsibility is to keep test automation ongoing in continuous integration server and to test daily software builds with regression testing. The new test analysis tool consists of backend server and a client. The client development is explained in this thesis.

PCI team consists of testers, who ensures product quality by performing regression tests. Regression tests are executed in a continuous integration environment. Continuous integration forms test automation. Most important tools of the tester’s work are Jenkins and Jira. Jenkins is used for running tests in a CI server. There are lots of Jenkins jobs for testing different domains of the product and they are all executed parallel with each other. After all Robot Framework tests are executed with latest software build from Source Control Management (SCM), the test results are visible in the Jenkins jobs. Logs and test results are stored into a log server after Jenkins job executions. The log server contains over a year of data and thus the test data is huge. Jira is a task management software built for designing and monitoring workflow.

Since there are an enormous amount of tests running in multiple Jenkins jobs daily, it is hard to keep track of all the test results. Also, analysis of the root cause for failed test case is time consuming because of large amount of manual work. This thesis explains how test result data is used in client side of this new test analysis tool for quickening analysis time.

2 Test automation tools and technologies

This chapter presents the problem that this thesis will solve by implementing a new software tool. Before the problem is presented, concepts of continuous integration, continuous delivery, Jenkins, and Robot Framework will be clarified. These concepts are tools and techniques which are important in presenting the project problem and essential in making of this thesis.

2.1 Software development process

Normally product development with multiple developers is built by combining code changes from developers into one source code repository. Source code repository is built into a build package and then sent to testing server for final testing before product release into production server (see Figure 1.). If there are any bugs found in testing server, developers will be notified to fix the bugs and the cycle of building the product will start again. This may be a fine development process and could work with small products with fewer commits.

However, if the product is huge and there are multiple developers, there can be problems with this kind of development process. If there is a bug or a feature that breaks the product, developers must wait till the complete software is re-developed and re-built for the test results until they can implement more features for the product. This takes a long time and it will slow down the product development process by a lot. Additionally, if the product has huge amount of source code and testing fails, the developer would have to locate the bug from entire source code of the software. Locating and fixing the bug could be very time consuming which would also slow down the process a lot. If there are lots of bugs, developers would have to consume time to fix bugs instead of developing new features for the product.

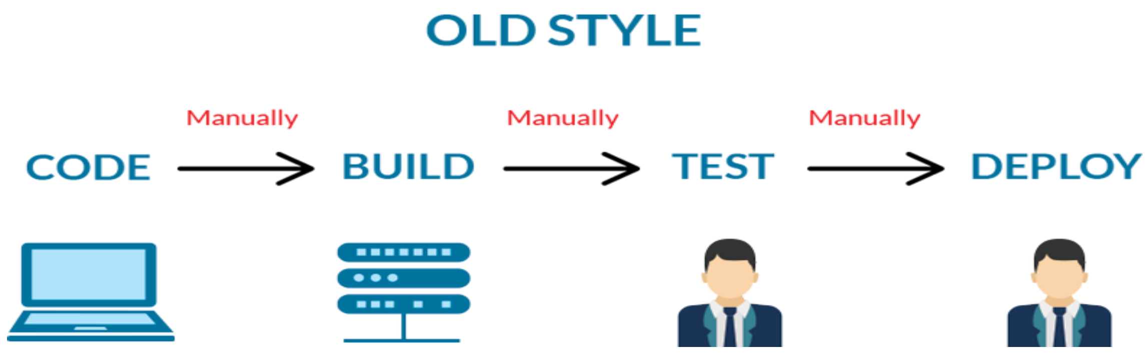


Figure 1. Old style of development process [1].

These issues within software development process can be resolved with Continuous Integration (CI). CI is a development practice enabling automatic software building, testing, and deploying in a CI server. Developers commit their code changes into the source code repository. CI server checks the source code repository for commits, subsequently automatically pulls the commits from source code repository. After that, the whole software package is built and tested. If there are any bugs in the code, developers will know which commit has caused the bug instead of going through the whole source code of the product. Locating malfunctioning code within the software is easier therefore fixing the bugs is faster. With CI, every commit is tested. As a result, developers do not have to wait a long time for test results. Figure 2 shows the CI process. First developers commit to source code repository. Then source code is built and tested in a CI server. After testing, developers get the feedback of how the product worked in tests. CI speeds up the development process and therefore frequency of new working software releases is increased. CI enables developers to get frequent feedback of their product's status. Overall, CI enables automated and fast software build testing and deployment. Everyone can see the results and software can be maintained in one single source code repository. This will even make it easy for anyone to get the latest version of software.

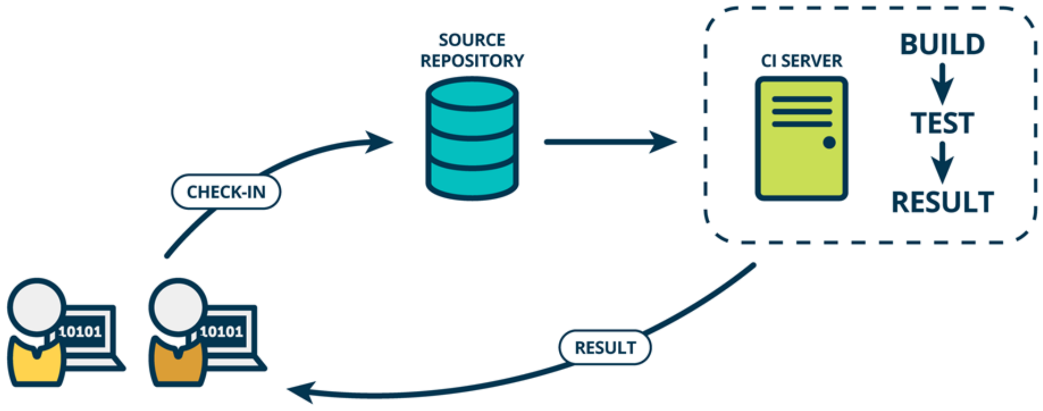


Figure 2. Software developers commit into source code repository which then is immediately built and tested in CI server [2].

For CI to properly work, developers must commit very frequently into the source code repository to reduce the number of bugs there might be. However, developers shouldn’t push erroneous or untested code into the repository. In addition, testers should monitor tests in CI machine. If defects occur, it is important to find root cause and report it for fixing. Testers are responsible for keeping the CI process ongoing with automated testing. The build could be broken in CI “trunk” by failing tests in the most important testing levels. Trunk means the head branch of software repository.

CI is pre-requisite for Continuous Delivery (CD). As a continuous integration, continuous delivery is a software development practice. It aims for building, testing, and releasing software is short cycles. CD depends on automation process of CI, so that release cycle can be fast with good quality. In CD, decision to release the build into production environment is done manually. Releasing can also be automated but then, continuous delivery changes more into a term continuous deployment. Continuous deployment requires no human intervention with releasing software builds. Releasing a build in continuous delivery is a business decision requiring knowledge about the quality of the product. Quality may be poor if regression tests have not gone well and therefore, the build cannot be released into production. [3]

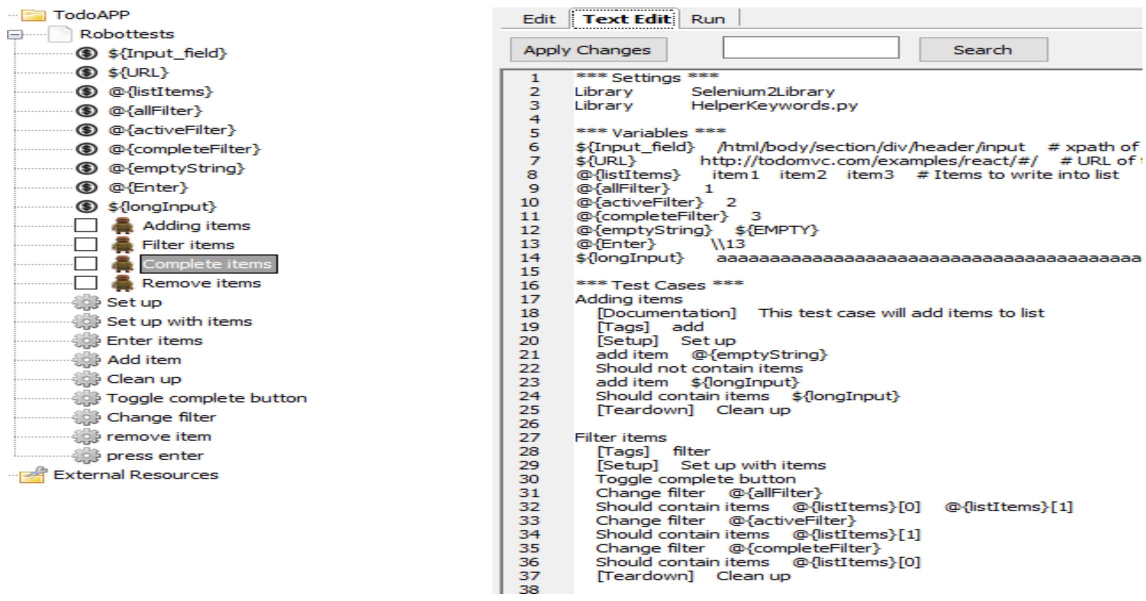
2.2 Jenkins

Multiple CI servers are available for use. Most popular CI servers are Jenkins, GitLab CI, Bamboo and Travis CI. Jenkins is the chosen tool for test automation by the subscriber of this project, Nokia. Jenkins is an open source automation tool with thousands of plugins. Jenkins is written with Java and it implements CI with help of plugins. Plugins make Jenkins very flexible and therefore can be configured into many purposes. Jenkins is a cross-platform CI server with web-based Graphical User Interface (GUI). Jenkins can hold lots of items depending on size of Jenkins server disk size.

Jenkins items are usually called Jenkins jobs and each job can be configured differently and given input parameters for execution. If Jenkins jobs are used for CI, they must have SCM, which fetches the newest commits from source code repository. Jenkins job does not need to trigger only from SCM. Other jobs, Jenkins job Uniform Resource Locator (URL) or manually building the project from web browser can trigger a Jenkins job. After Jenkins job is triggered, it starts a new build with configured build step and given parameters. Build step executes any tests or scripts configured. After the build step is done, post-build actions are executed. Post-build actions include archiving results, publishing results, and triggering other jobs. [4]

2.3 Robot Framework

Robot Framework is a generic test automation framework for acceptance testing and acceptance test-driven development (ATDD) [5]. Acceptance testing focuses on testing that system meets the business requirements and whether it is acceptable for delivery. Test cases are created to ensure that system works the way it was specified. The idea for test automation in Robot Framework is to reduce manual labor work, execute tests fast, reuse as much as possible to reduce complexity and most importantly be repetitive. Test automation should also increase testing coverage for finding as much defects as possible. Robot Framework implements testing with keyword driven testing approach. In testing, keywords represent the functionality of the application. Figure 3 displays four test cases: adding items, filter items, complete items and remove items. What these test cases test is defined with keywords below test cases. Many of those keywords come from Selenium library, which is a web testing library designed for automated testing of web applications. Besides Selenium, Robot Framework has a lot of libraries with all sorts of keywords for testing different functionalities. They are not just web testing libraries but for variety of different applications. Designing test cases with keywords is easy since keywords are reusable on multiple tests and they are easy to read and make. In addition to Robot keywords, Robot Framework has a lot of application programming interfaces (API) to help make custom keywords that can be developed with Python or Java.

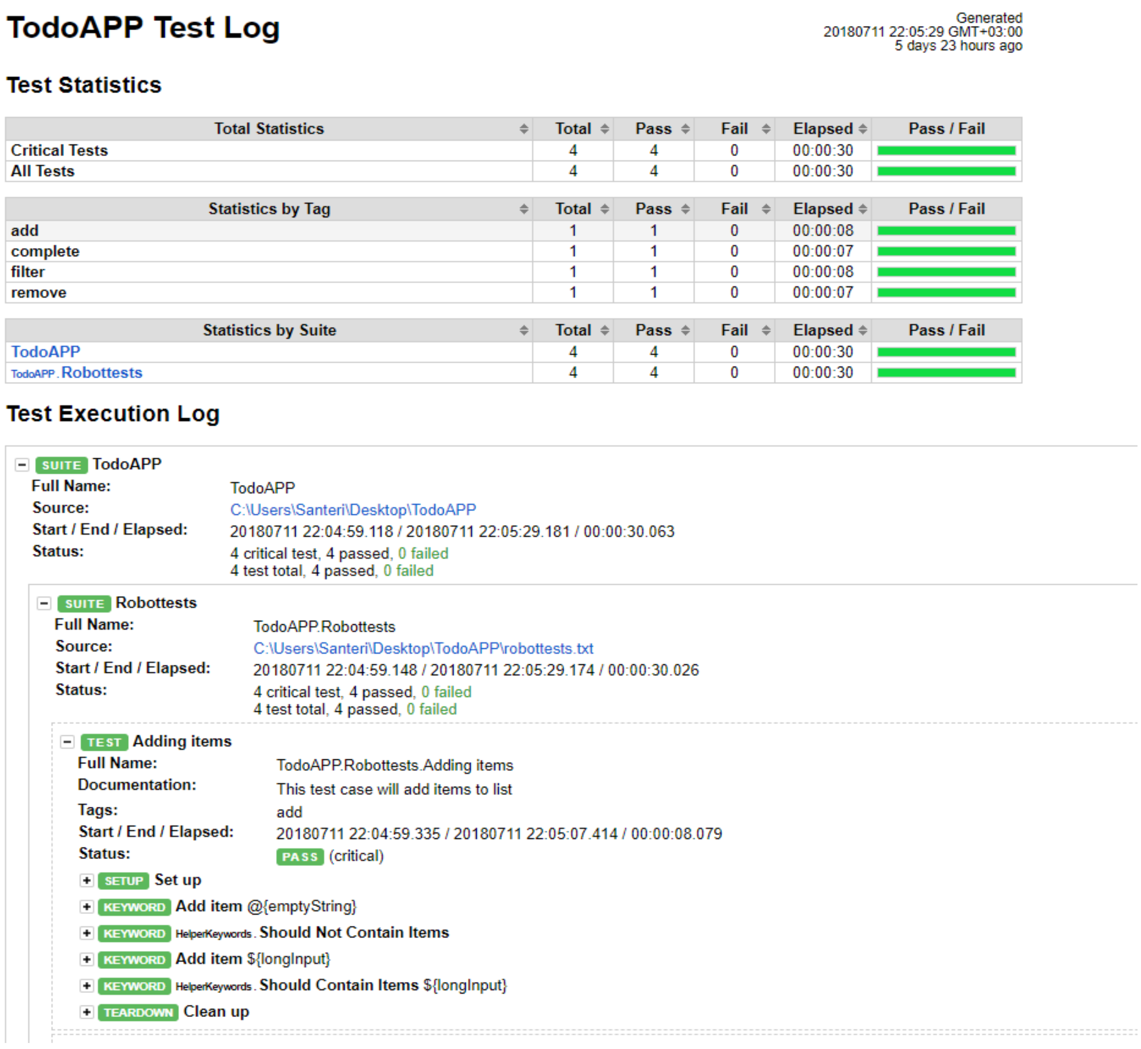


```
TodoAPP
  Robottests
    ${Input_field}
    ${URL}
    @{listItems}
    @{allFilter}
    @{activeFilter}
    @{completeFilter}
    @{emptyString}
    @{Enter}
    ${longInput}
    Adding items
    Filter items
    Complete items
    Remove items
    Set up
    Set up with items
    Enter items
    Add item
    Clean up
    Toggle complete button
    Change filter
    remove item
    press enter
External Resources
```

```
*** Settings ***
Library          Selenium2Library
Library          HelperKeywords.py

*** Variables ***
${Input_field}    /html/body/section/div/header/input    # xpath of
${URL}            http://todomvc.com/examples/react/#/    # URL of
@{listItems}     item1    item2    item3    # Items to write into list
@{allFilter}      1
@{activeFilter}    2
@{completeFilter}    3
@{emptyString}    ${EMPTY}
@{Enter}         \\13
${longInput}     aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

*** Test Cases ***
Adding items
    [Documentation]    This test case will add items to list
    [Tags]    add
    [Setup]    Set up
    add item    @{emptyString}
    Should not contain items
    add item    ${longInput}
    Should contain items    ${longInput}
    [Teardown]    Clean up

Filter items
    [Tags]    filter
    [Setup]    Set up with items
    Toggle complete button
    Change filter    @{allFilter}
    Should contain items    @{listItems}[0]    @{listItems}[1]
    Change filter    @{activeFilter}
    Should contain items    @{listItems}[1]
    Change filter    @{completeFilter}
    Should contain items    @{listItems}[0]
    [Teardown]    Clean up
```

Figure 3. Example of Robot test cases implemented for testing a web application with help of Selenium2Library and Robot Integrated Development Environment (RIDE).

As a results of robot tests, output files are generated in the form of machine-readable format, Extensible Markup Language (XML). By default, the generated XML file is called output.xml. This output.xml contains lots of data about the test execution. For example, there is start and end time of test case executions, timestamps, documentation, test case name, keywords, tags, status (pass/fail) and error messages. Output.xml is difficult to be read by user and that's why log.html and report.html files are generated based on the output.xml. Log file contains lots of details about executed test cases. It shows test suite, test case and keywords used in tests as seen in figure 4. A test suite is a collection of test cases in a file. A test case is a collection keywords defined to test a specific functionality in an application. Log files are useful for exploring which test case failed and what keyword caused the test case to fail. Typically, every time failed test case is being analyzed, tester goes through logs to find the root cause.



TodoAPP Test Log

Generated
20180711 22:05:29 GMT+03:00
5 days 23 hours ago

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	4	4	0	00:00:30	🟩
All Tests	4	4	0	00:00:30	🟩

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
add	1	1	0	00:00:08	🟩
complete	1	1	0	00:00:07	🟩
filter	1	1	0	00:00:08	🟩
remove	1	1	0	00:00:07	🟩

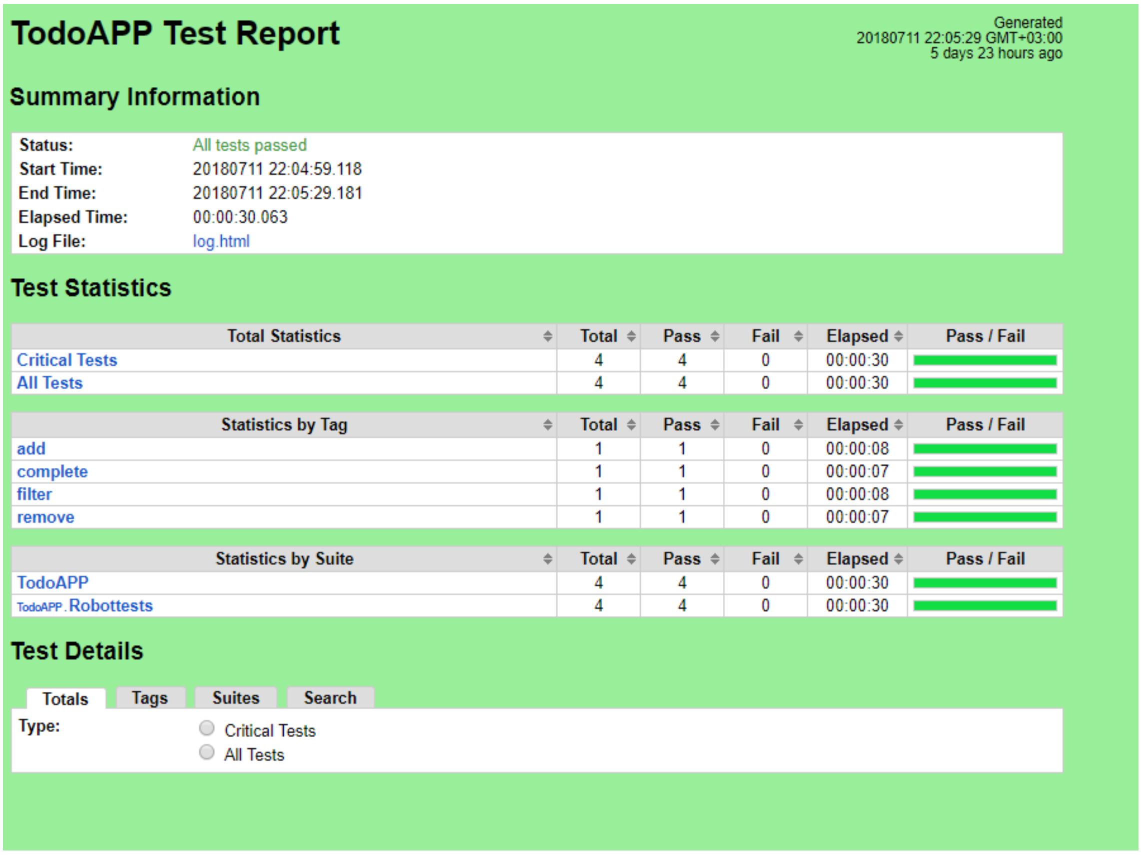
Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
TodoAPP	4	4	0	00:00:30	🟩
TodoAPP . Robottests	4	4	0	00:00:30	🟩

Test Execution Log

```
− SUITE TodoAPP
  Full Name:            TodoAPP
  Source:               C:\Users\Santeri\Desktop\TodoAPP
  Start / End / Elapsed: 20180711 22:04:59.118 / 20180711 22:05:29.181 / 00:00:30.063
  Status:               4 critical test, 4 passed, 0 failed
                        4 test total, 4 passed, 0 failed

  − SUITE Robottests
    Full Name:            TodoAPP.Robottests
    Source:               C:\Users\Santeri\Desktop\TodoAPP\robottests.txt
    Start / End / Elapsed: 20180711 22:04:59.148 / 20180711 22:05:29.174 / 00:00:30.026
    Status:               4 critical test, 4 passed, 0 failed
                          4 test total, 4 passed, 0 failed

    − TEST Adding items
      Full Name:            TodoAPP.Robottests.Adding items
      Documentation:        This test case will add items to list
      Tags:                 add
      Start / End / Elapsed: 20180711 22:04:59.335 / 20180711 22:05:07.414 / 00:00:08.079
      Status:               PASS (critical)
      + SETUP Set up
      + KEYWORD Add item @{emptyString}
      + KEYWORD HelperKeywords . Should Not Contain Items
      + KEYWORD Add item ${longInput}
      + KEYWORD HelperKeywords . Should Contain Items ${longInput}
      + TEARDOWN Clean up
```

Figure 4. Robot log containing details about test cases and keywords.

Report.html contains the overview of tests execution results. It shows total number of tests executed, pass, and fail numbers. Additionally, tags and all the names of test cases with documentation, error message and duration are shown as seen in figure 5. Report.html is useful for quickly seeing how many tests have failed and for what reason. If there are multiple test cases failed with same error message, they might have failed for the same reason and thus gives an idea of root cause before going deeper into logs for analyzing. Report.html has a red background color if there are any failed tests and green if everything is passed. Test cases can also be filtered by suite and tags.



TodoAPP Test Report

Generated
20180711 22:05:29 GMT+03:00
5 days 23 hours ago

Summary Information

Status:	All tests passed
Start Time:	20180711 22:04:59.118
End Time:	20180711 22:05:29.181
Elapsed Time:	00:00:30.063
Log File:	log.html

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	4	4	0	00:00:30	🟩
All Tests	4	4	0	00:00:30	🟩

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
add	1	1	0	00:00:08	🟩
complete	1	1	0	00:00:07	🟩
filter	1	1	0	00:00:08	🟩
remove	1	1	0	00:00:07	🟩

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
TodoAPP	4	4	0	00:00:30	🟩
TodoAPP.Robottests	4	4	0	00:00:30	🟩

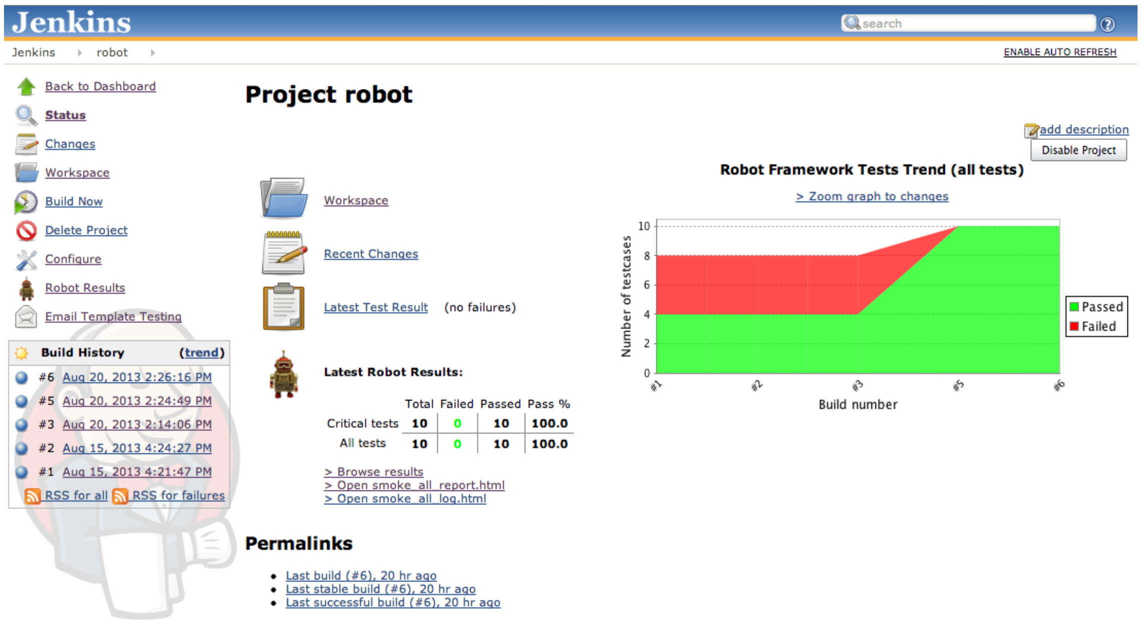
Test Details

Type: ○ Critical Tests ○ All Tests

Figure 5. Successful robot report containing summary of test case execution.

2.4 Integration of Robot Framework and Jenkins

Jenkins supports Robot Framework with a useful plugin, Robot Framework plugin. This plugin allows publishing Robot Framework tests in a Jenkins job build. User can configure Jenkins job to execute robot tests in build step by giving path to the robot tests directory and then execute them with robot command. Jenkins supports multiple Operating Systems (OS) with their command window. For windows system, there is build step for executing windows batch command and for Linux system, there is execute shell. After fetching tests and executing them, publishing test results is done in post-build actions with section "Publish Robot Framework test results". This section is from Robot Framework plugin and may be selected from add post-build action button after installing the plugin to Jenkins system. Robot Framework plugin requires a directory of where the robot results are output after execution. As a result, Jenkins job page shows the robot results as seen in figure 6. Threshold of build result percentage may be given, if different color is wanted for marking job passed, failed or unstable.



Project robot

Workspace

Recent Changes

Latest Test Result (no failures)

Latest Robot Results:

	Total	Failed	Passed	Pass %
Critical tests	10	0	10	100.0
All tests	10	0	10	100.0

> Browse results
> Open smoke_all_report.html
> Open smoke_all_log.html

Build History (trend)

- #6 Aug 20, 2013 2:26:16 PM
- #5 Aug 20, 2013 2:24:49 PM
- #3 Aug 20, 2013 2:14:06 PM
- #2 Aug 15, 2013 4:24:27 PM
- #1 Aug 15, 2013 4:21:47 PM

RSS for all RSS for failures

Permalinks

- Last build (#6), 20 hr ago
- Last stable build (#6), 20 hr ago
- Last successful build (#6), 20 hr ago

Figure 6. Jenkins job page showing Robot results. [6]

2.5 Problem presentation

Testers in PCI team analyze Robot Framework test results from multiple Jenkins jobs daily. In CI, several new software builds are available daily of which must be tested by executing thousands of tests in multiple Jenkins jobs. Execution of tests is automatic, but analysis of test results is not. Testers analyze the failed test cases manually by looking for root cause in the logs. In addition to finding the root cause of the problem, failure must be reported to developers for fixing with help of Jira.

Analyzing test results in CI is time-consuming. It is caused by manual work such as clicking failed test results in Jenkins jobs, searching for root cause in logs and creating or updating Jira tickets. Every time a Jenkins job has executed all tests, test results are updated into a storage server called log server. All the data in log server was unused and therefore, it was desired to create a new analysis tool that utilizes these unused logs in test analysis. With history data of test results, testers could identify similar previously occurred failure. Also, a summary of test results in one build could be easily observed instead of going through every Jenkins job.

New analysis tool already has a backend server and data management for parsing data from log server and storing it into a database. Database is located on the server. Database has four collections: build collection, Jenkins job collection, test case collection and a Jira collection. Build collection contains the software build id and version number. Jenkins job collection contains data from Jenkins job execution. Test case collection contains data of all the test cases. Jira collection contains all the data of Jira tickets made by testers. Data is there, but analysis tool has no client software for visualizing it. This thesis solves this problem by implementing the client software as a part of the analysis tool.

2.6 Thesis objectives

Objectives of this thesis was to study the design and implementation of a client software based on features mentioned in the user stories. A user story is a brief and to the point description of wanted feature told by the user. User stories are shown in listing 1.

User stories
As a tester, I would like the client to have a functional and easy to use graphical user interface for the end users.
As a tester, I would like that client should be able to communicate and ensure data is fetched from server.
As a tester, I would like the client software to support legacy tools and their features.
As a tester, I would like that client should be able to visualize data with different charts and tables.
As a tester, I would like to see easily one test case execution history if pass or failed. If failed, I would like to see which build failed and error code why failed.
As a tester, I would like to see if JIRAs created for this test case.
As a tester, if test case fails, I would like to see if there are changes in that domain test case repository or software repository
As a tester, I would like to see domain test case success history, which test case failed and which build.
As a domain developer, I would like to have list of top 10 longest test case.
As a tester, I would like to see all test case results of build visualized

Listing 1. User stories of wanted features in client software.

3 Technologies of client software development

This chapter introduces the technologies used in the client software. This project is solely programmed with Java. Java is not just a programming language, it is technology. Java was chosen as the main technology for implementation since it works on multiple operating systems. This is due to code execution happening on Java Virtual Machine (JVM). Java also has many useful libraries and one of them is JavaFX, which can make good-looking user interfaces.

3.1 Java

Java technology contains both Java programming language and Java platform. A platform is commonly defined as the hardware environment or software environment on which programs are executed. Java platform differs from common platforms due to the fact that it is a software platform on top of hardware platform. It consists of Java API and JVM. Java is an Object-Oriented Programming (OOP) language which means that with good planning, code is easily readable, reusable, and maintainable. Java is simple, dynamic, and robust and it all runs on JVM [7]. Java has many excellent libraries, frameworks, tools, and integrated development environments (IDE) for an application development.

When selecting a programming language, it depends on which purpose it will be used for. For high speed and low memory requirements, low-level programming language like C could be the best choice. For enterprise projects, higher-level language like Java is often very good because applications are easier to deploy to any enterprise environment and any platform. Portability is one of the best strengths in Java and key reasons why it is so widely used in enterprises [8]. Also, portability is one of the reasons why it was chosen as the programming language in this bachelor’s thesis

3.1.1 Java architecture

Java language code is saved as .java files. Java compiler converts these files into a Java bytecode, which is not understandable by any platform except JVM. JVM resides in the random-access memory (RAM) of an operating system. When JVM is fed with java bytecode, it identifies the OS it is working on and converts the bytecode into native machine code for the processor as seen in figure 7. This makes compiled Java code OS independent and runnable in all kinds of devices.

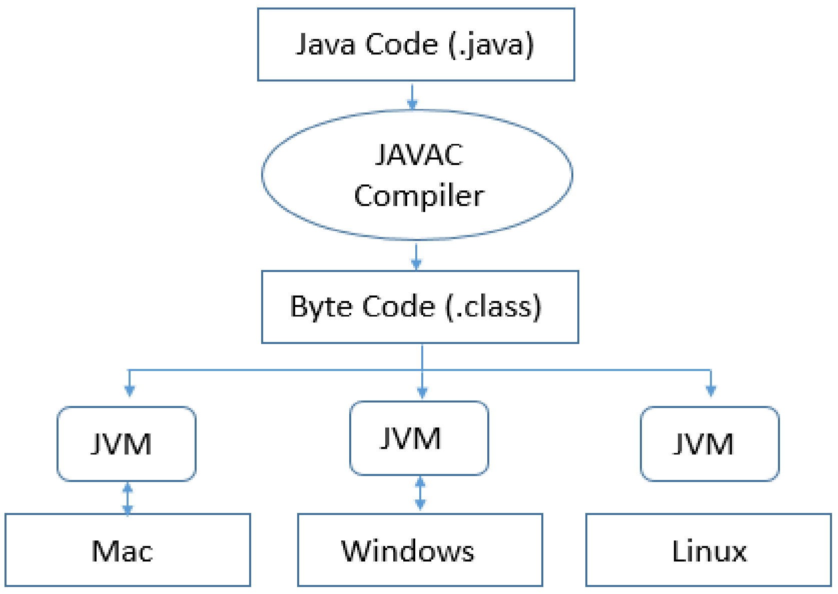


Figure 7. Architecture of Java program from Java source code to any OS machine code [9].

3.1.2 Java development kit

Java development kit (JDK) is the software development environment provided by Java. Essentially, it is a bundle of software components used in Java applications. It contains Java Runtime Environment (JRE), Java, Java compiler and development tools (see figure 8.). JRE is the implementation of JVM, but it also contains other libraries and files e.g. browser plugins for applet execution. Executing Java programs only need JRE for JVM and library dependencies. For Java programming, JDK is needed for its Javadoc, Java Debugger, and Java compiler. Java compiler is given the Java source code for compiling it into Java bytecode which is understood by JVM. Javadoc is needed for creation of documentation inside the source code. Java debugger is needed for development of the source code since it can help with errors occurring in JVM.

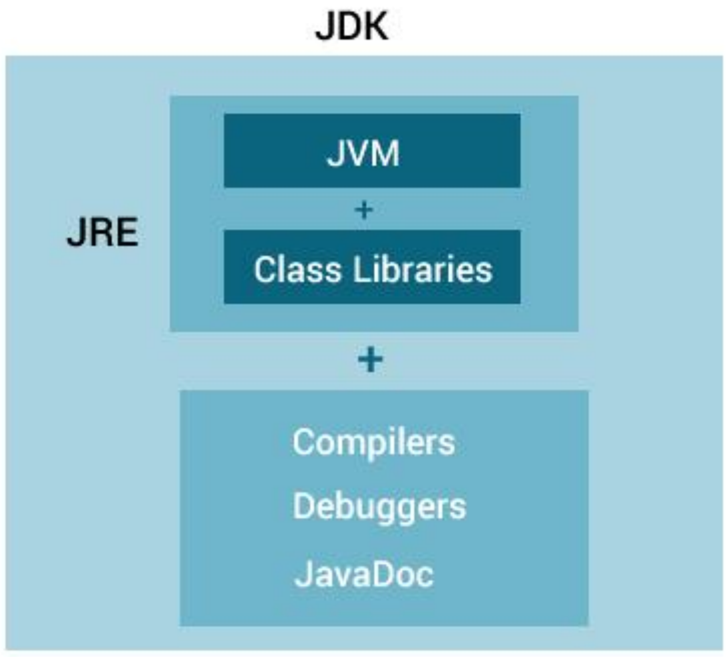


Figure 8. Contents of JDK. JRE is needed for Java program to run but for Java development, whole JDK is required [10].

Standard Java programming language and JavaFX is included in JDK. Both are heavily used in implementation of this bachelor’s thesis with version 8 of JDK.

3.2 JavaFX

JavaFX is a Java software platform generally used for creating computer desktop applications. JavaFX can also be used to create Rich Internet Applications (RIAs) that resemble desktop application but are web applications [11]. In addition to desktop applications, JavaFX can be run in browser by adding Java applet plug-in. Generally, JavaFX is used for creating good looking GUI applications. JavaFX works on wide range of devices like computers, mobile phones, tablets. JavaFX works on multiple operating systems since JavaFX is available on multiple JVM. These JVM have runtimes for windows, Linux and macOS [12].

At first JavaFX version 1.0 was a scripting language used with Java. Intention for JavaFX was to allow users to make user interfaces easier but there was too much complexity. All of GUI components had to be built with large amounts of code with procedural way of coding. This made code complex to follow and did not result in good looking user interfaces.

3.2.1 JavaFX structure

After version 2.0, JavaFX was no longer a scripting language. A declarative way of coding was introduced with JavaFX markup language (FXML). All of JavaFX functionality was moved into a Java API. JavaFX API incorporated everything that the GUI needed. Users could use normal Java syntax in creation of GUI applications [13]. Figure 9 shows the JavaFX API architecture which contains various set of classes and interfaces that rich internet applications need.

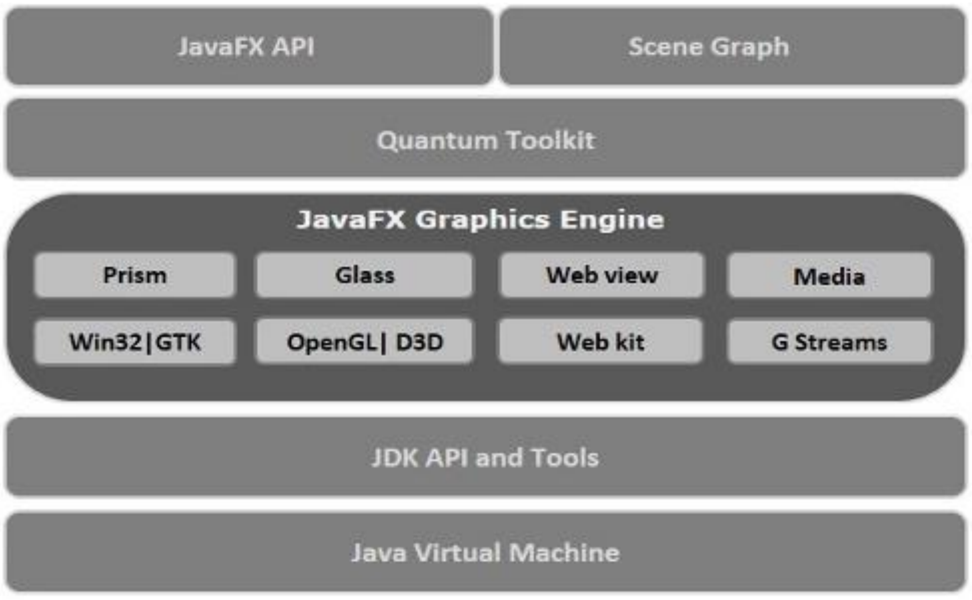


Figure 9. Architecture JavaFX API and its components [14].

For most important and essential functionalities of JavaFX is Animation, Cascading style sheet (CSS), Event, Geometry, Stage and Scene. Most of these belong to the Scene Graph. When designing JavaFX graphic implementation methods, levels of application and their layout need to be considered.

There are multiple layers of structure in a JavaFX application. The application structure is a hierarchy structure. Figure 10 shows JavaFX application structure, in which the lowest level is the UI elements. UI elements are contained on a scene at the middle level.

At the top level is Stage. Stage is the container for JavaFX application. Stage can be thought of being the application window e.g. a browser window. It can have different styles. Decorated style for white background and platform specific decorations. Transparent style for transparent background and no decorations. Undecorated style for white background and no decorations. Stage cannot be seen unless it has platform specific decorations on application window borders or no scenes on top of it. Every operating system has different style of window decorations. Within the Stage is a scene. Scene is an area where graphical content is laid out. In other words, scene is the container for UI elements. There can be multiple scenes within a stage which is like a webpage in a browser.

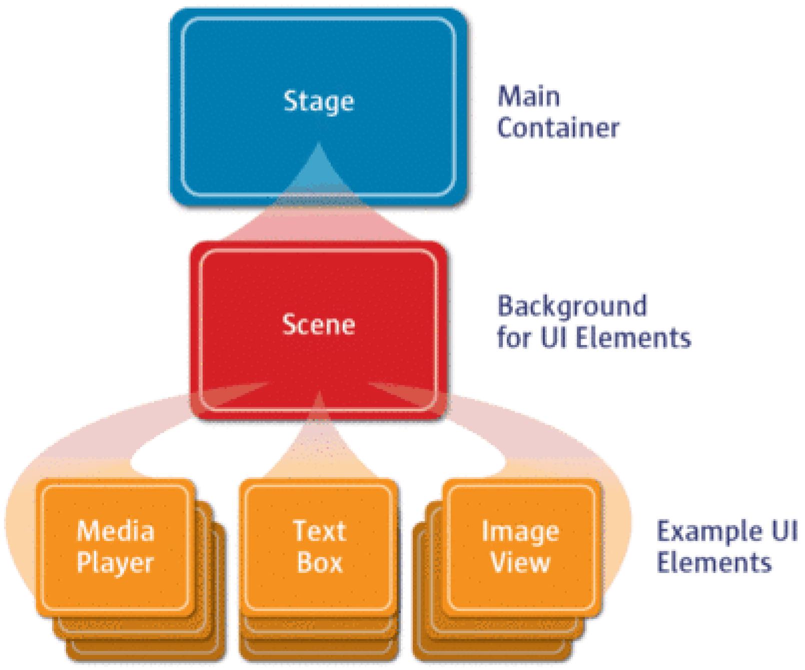


Figure 10. JavaFX application structure [15].

UI elements are in tree kind of structure. Root node at bottom, branch node in middle and leaf node on top. A node is an UI element, which is visualized on a scene graph. Root node is the parent node for all other nodes. Root node cannot have parent of its own, only children. Children of root node are either leaf nodes or branch nodes. Leaf nodes cannot have children because they already have some functionality therefore nothing cannot be placed on top of them. Branch nodes are also children of root node and they can have children of their own. Figure 11 shows group as root node, which has three children: circle, rectangle, and region. Circle and rectangle are leaf nodes as they cannot have children. Region is a branch node because it has two children: text and image view.

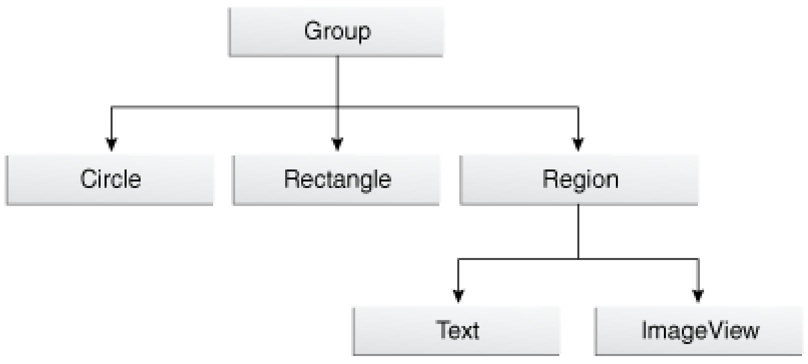


Figure 11. Node structure in a scene graph [16].

Good examples of leaf nodes are for instance, Image View, MediaView, and any text area. Branch nodes display an area in a form where more content can be put in. Its children are either a branch node or a leaf node. For example, root node and branch node are layout panes. Layouts are usually container classes called pane. Most common layouts used in JavaFX applications are HBox, VBox, BorderPane, StackPane, AnchorPane and a GridPane. Also, these layouts are much used in this thesis project. HBox positions the nodes inside it into a horizontal row. VBox does the same, but in a vertical row instead of horizontal. BorderPane positions nodes to top, bottom, left, right or center of the layout. StackPane allows nodes to be put on top of another like a stack. With Anchor pane, nodes can be anchored to specific distance from the pane. GridPane positions the nodes into a grid by rows and columns.

3.2.2 JavaFX coding styles

JavaFX can be both coded with a procedural way or with a declarative way. A problem with procedural way is complexity. Every node must be created as an object from the classes JavaFX API provided. Then assign events, actions, method calls to them in plain code. Every action for one node object must be coded. This results in a large amount of code in one Java file, and thus becomes hard to follow. Debugging errors would be nightmare, even if IDE debugger helps to find the errors.

In listing 2, application is coded with a procedural style. When main method is called on startup, it calls for launcher method. This constructs application class and then calls start method which in turn creates a new thread for running the application. In start method, stage is first created and then a button object is created and handed an event to print out “Hello World!” in console. Afterward, root node StackPane is created and button is added into it as its child node. StackPane is added into a new scene object with width and height hardcoded. Stage window is given a title, scene and then called to show itself to user.

```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

/**
 *
 * @author Santeri
 */
public class Demo2 extends Application {

	@Override
	public void start(Stage primaryStage) {
		Button btn = new Button();
		btn.setText("Say 'Hello World'");
		btn.setOnAction(new EventHandler<ActionEvent>() {

			@Override
			public void handle(ActionEvent event) {
				System.out.println("Hello World!");
			}
		});

		StackPane root = new StackPane();
		root.getChildren().add(btn);

		Scene scene = new Scene(root, 300, 250);

		primaryStage.setTitle("Hello World!");
		primaryStage.setScene(scene);
		primaryStage.show();
	}

	/**
	 * @param args the command line arguments
	 */
	public static void main(String[] args) {
		launch(args);
	}

}
```

Listing 2. Procedural coding of simple JavaFX application code for printing “Hello World!” on push of a button in GUI.

Another option is the declarative way with FXML. FXML is an XML based markup language. Markup languages are widely used for storing and transporting data in both human-readable and machine-readable format. Basic XML does not actually do nothing but hold data inside tags [17]. Many APIs has been developed based on XML and one widely used is Hypertext Markup Language (HTML). HTML is used for creating web pages and web applications. In addition to basic XML data storage, it has a functionality in displaying data, but it has predefined tags that XML does not have. FXML works similarly with JavaFX scene graph. Nodes can be written into markup language in separate FXML file and then they can be accessed from Java application code. In listing 3, same code functionality is written as in listing 2, but in declarative way with FXML. Nodes being in tags keeps the scene graph logic. Outermost node is the root node and inner indentations are its children. Tagged nodes can have properties coded in them similar way as in HTML. Also, event names can be declared in node tags. With event names in FXML, application code knows which node and event method are linked together. For application logic code to recognize nodes, there must be an fx:id defined to a node. Also, for CSS to recognize a node, node must have a normal id. This way all the structure of GUI code is separated from Java code used in creating the logic for application and it makes it easier to manage components in GUI.

```
<AnchorPane id="AnchorPane" prefHeight="200" prefWidth="320"
xmlns:fx="http://javafx.com/fxml/1" fx:controller="demo.FXMLDocumentControl-
ler">
    <children>
        <Button layoutX="126" layoutY="90" text="Click Me!" onAction="#handle-
ButtonAction" fx:id="button" />
        <Label layoutX="126" layoutY="120" minHeight="16" minWidth="69"
fx:id="label" />
    </children>
</AnchorPane>
```

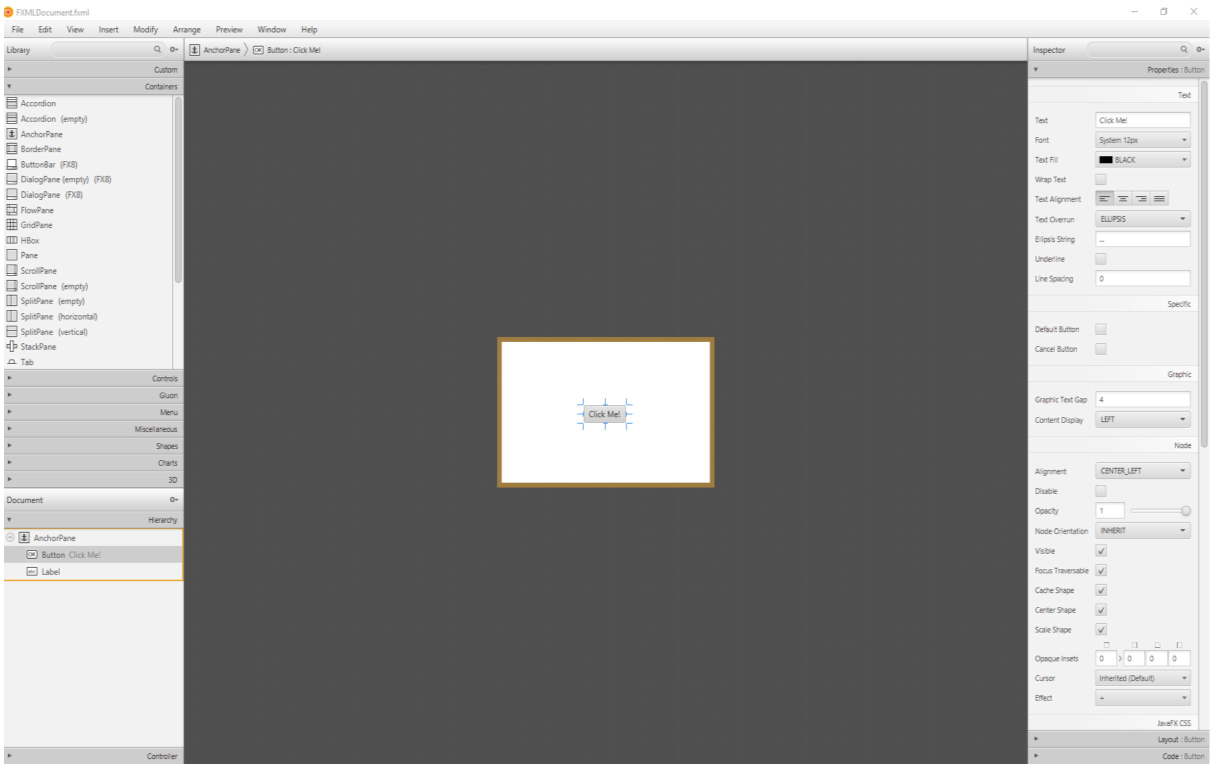
Listing 3. Declarative way of coding for printing button.

3.3 Scene builder

Application logic code is in java files. There is Java controller class for controlling the FXML file. Also, another Java file for main method and start method, same as in procedural JavaFX. After launch is called in main Java file, stage is set, and new scene object is given a parent by *FXMLLoader* class from FXML API. *FXMLLoader* calls load method, which loads the fxml file's object hierarchy by parsing the FXML document and thus building the scene graph. This links the fxml file to the application code. Every FXML file must have a controller. Controller class file is defined to FXML file by *fx:controller* property declared in the FXML file. As the FXML file is loaded, controller class is instantiated by controllers initialize method. Initialize method handles nodes and their properties declared in FXML file by having "@FXML" annotation in variables, events, and methods.

FXML style is preferred due to the lack of manual coding. It would be very time consuming to code nodes into FXML file with procedural way of JavaFX. Oracle offers a GUI building tool called Scene Builder. Figure 12 shows a scene builder program. Left side of the scene builder shows different kind of nodes, hierarchy, and controller. Middle displays the GUI and on the right, there is node properties. All JavaFX scene graph nodes and components can be graphically dragged and dropped into a scene graph hierarchy. With Scene Builder, node properties, alignment, styling, layout and adding events can be modified.

Basically, everything that can be done in FXML, can be done in Scene Builder. With Scene Builder, no code needs to be written into FXML file. Scene Builder generates all properties to nodes in FXML file upon saving the graphically built scene graph. CSS files can be specified in Scene Builder, so user can see all the styling as scene graph is being built. FXML file doesn't need to be compiled every time change have been made to it, so after dragging components to scene graph, hit save and the changes are there.



Text	Click Me!
Font	System 12px
Text Fill	■ BLACK
Wrap Text	☐
Text Alignment	
Text Overrun	ELLIPSIS
Ellipsis String	...
Underline	☐
Line Spacing	0
Default Button	☐
Cancel Button	☐
Graphic Text Gap	4
Content Display	LEFT
Alignment	CENTER_LEFT
Disable	☐
Opacity	1
Node Orientation	INHERIT
Visible	✓
Focus Traversable	✓
Cache Shape	✓
Center Shape	✓
Scale Shape	✓
Opaque Insets	0 0 0 0
Cursor	Inherited (Default)
Effect	+

Figure 12. Scene Builder visualization tool.

4 Software implementation

This chapter explains the development of client software and its features described in the user stories. Each subchapter explains in detail what was implemented, how they work and what the use of them for the end users are.

4.1 Implementation of graphical user interface

The new tool consists of several different modules. Including client module, server module, model module, data manager module and Jira fetcher module. Client and server modules are standalone programs which can be build, run, or debugged. Most modules depend on other internal modules. Each module has a Project Object Model (POM). POM is an XML configuration file for a Maven build system. Maven enables working with multiple modules by collecting modules to build and sorting them in correct building order resulting modules being interdependent. After building modules, Maven compiles modules into jar binary files which can be executed with JVM.

In this work, the most important module to look at is the client module. Figure 13 shows the general structure of the client which consists of main class, communication with server, searching functionality and control of data visualization.

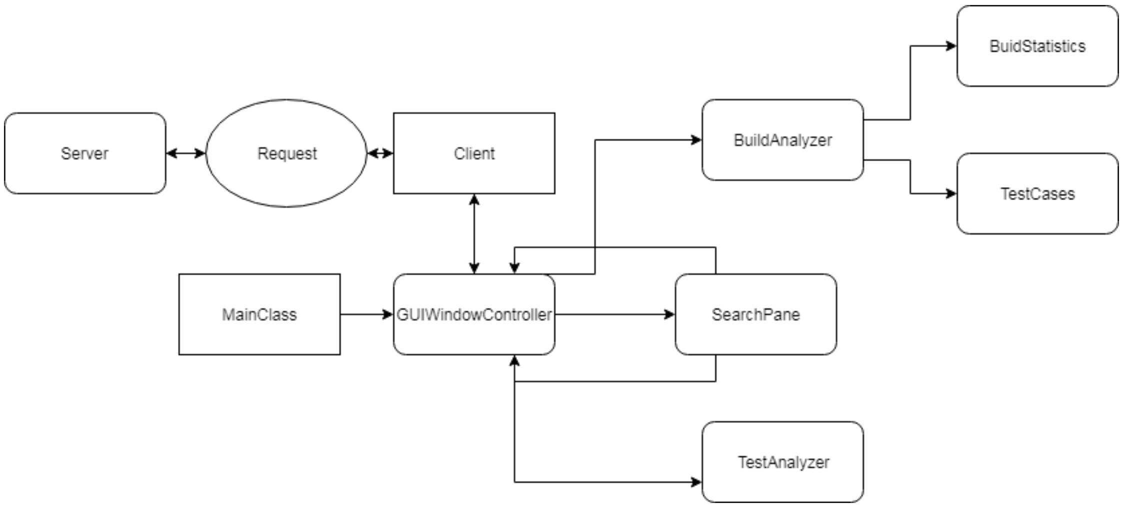
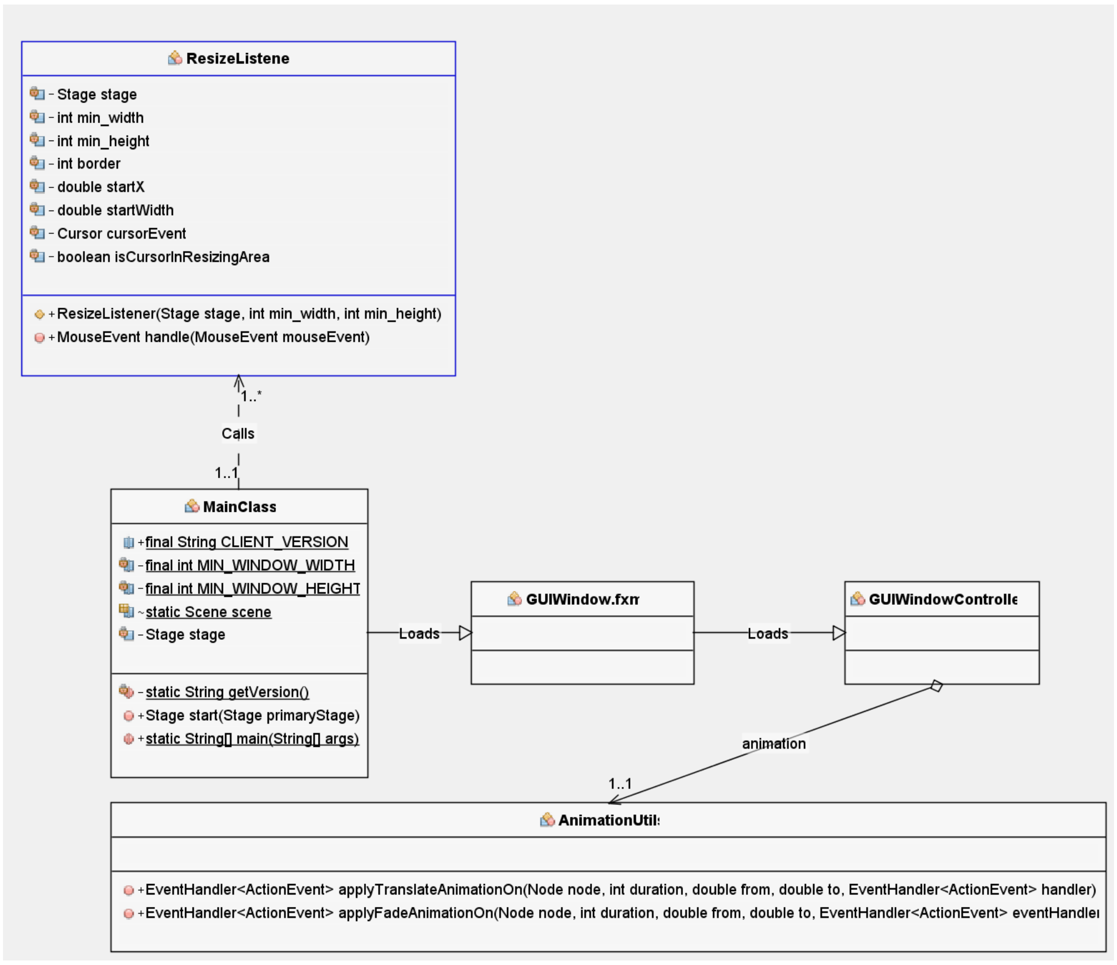


Figure 13. General structure of client software.

Client module has a main java class, which is first executed upon running the client software. This main class is responsible for setting the stage and loading the main window of the application into a scene. Main window is called *GUIWindow.fxml*. Main class gives the stage a minimum width and minimum height which are the width and height application window has on start. Also, main class uses a custom-made class called *ResizeListener* for resizing the GUI (see Figure 14.).



ResizeListene
- Stage stage
- int min_width
- int min_height
- int border
- double startX
- double startWidth
- Cursor cursorEvent
- boolean isCursorInResizingArea
+ResizeListener(Stage stage, int min_width, int min_height)
+MouseEvent handle(MouseEvent mouseEvent)
1..*
Calls
1..1
MainClass
+final String CLIENT_VERSION
- final int MIN_WINDOW_WIDTH
- final int MIN_WINDOW_HEIGHT
~ static Scene scene
- Stage stage
- static String getVersion()
+Stage start(Stage primaryStage)
+static String[] main(String[] args)
Loads
GUIWindow.fxm
Loads
GUIWindowControlle
animation
1..1
AnimationUtil:
+EventHandler<ActionEvent> applyTranslateAnimationOn(Node node, int duration, double from, double to, EventHandler<ActionEvent> handler)
+EventHandler<ActionEvent> applyFadeAnimationOn(Node node, int duration, double from, double to, EventHandler<ActionEvent> eventHandle

Figure 14. Implementation of main class in UML-chart.

4.1.1 GUI introduction

As a generic requirement, GUI should be easy to use for the end users. At the end of the project, GUI looks like any desktop application and it has the same basic functionality buttons as any browser does. Additionally, all the required features are accessible from icons on left side (see figure 15.). Pressing an icon shows the content in the middle of the application and highlights it. The content is shown on a node called *StackPane*. By default, *StackPane* shows homepage which now only displays a welcome text. In future, it could show news of software build releases. *GUIWindow.fxml* has a controller called *GUIWindowController* which is responsible for every functionality of nodes in *GUIWindow.fxml*. Content of *StackPane* is changed by a method in *GUIWindowController* called *setContent*. *SetContent* closes searching window if it's open, clears children of *StackPane* and adds content given to it by a parameter. Other methods can use this method to change what is displayed in GUI. Also, when content is changed, there is a fading animation for making the transition look smooth. This animation comes from custom made class *AnimationUtils*.

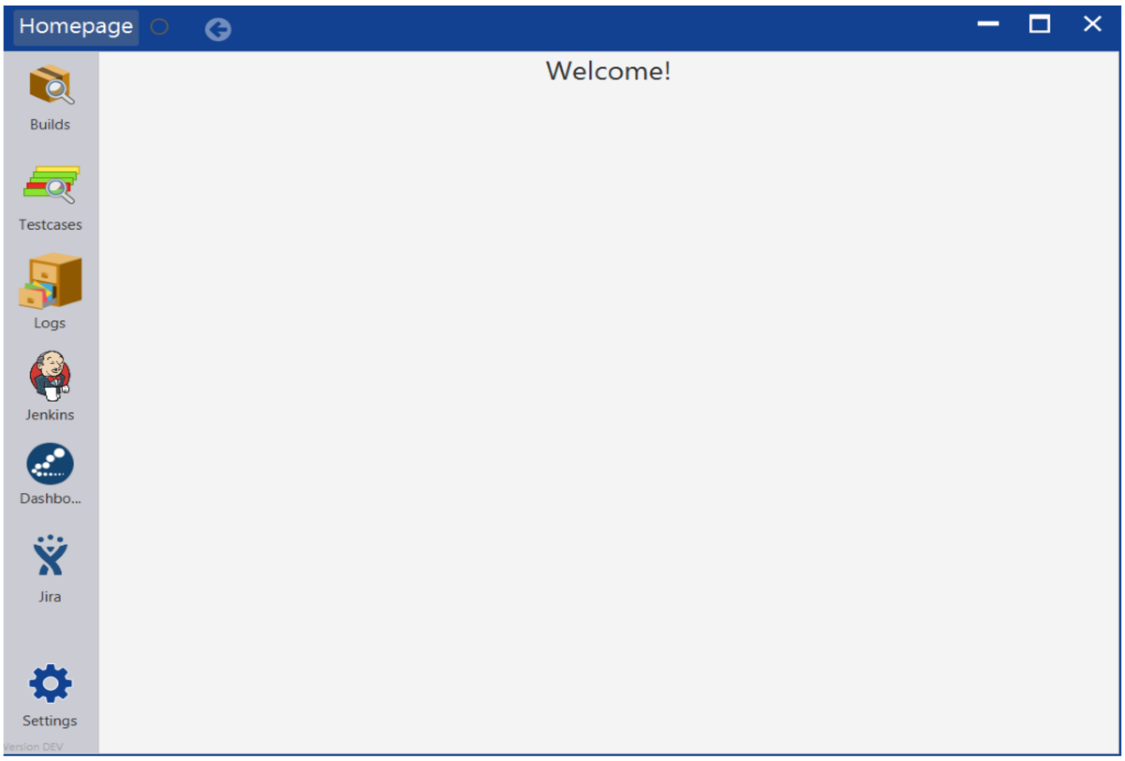


Figure 15. The main window of the application GUIWindow.fxml.

4.1.2 Resizing window

Main class sets the stage style into transparent for making application window borders custom styled instead of OS style. With transparent style, it means that there are no operating system decorations on window borders. This introduces a problem. Window cannot be resized since there are no borders. JavaFX has no way of handling resizing on undecorated borders. Solution is to create a new custom-made java class called *ResizeListener* in the same package as the main java class. *ResizeListener* object is created in main class. Object takes stage, minimum width, and minimum height as parameters. *ResizeListener* will handle mouse events on window. *Mouse_pressed*, *mouse_moved* and *mouse_dragged* events are needed for resizing the window (see listing 4).

```
ResizeListener resizeListener = new ResizeListener(stage, MIN_WINDOW_WIDTH,
MIN_WINDOW_HEIGHT);
scene.addEventFilter(MouseEvent.MOUSE_MOVED, resizeListener);
scene.addEventFilter(MouseEvent.MOUSE_PRESSED, resizeListener);
scene.addEventFilter(MouseEvent.MOUSE_DRAGGED, resizeListener);
```

Listing 4. ResizeListener object created with event filters which will be called when scene receives event of specified type.

If the mouse is pressed on window borders, screen x and y values and scene width and height are stored into variables. These are the starting point values. Mouse event *mouse_moved* will check if mouse is moved on the borders and if stage is maximized. If it returns true, then stage cannot be moved because window is on full screen. If false, cursor type will be changed into corresponding cursor depending on how the cursor is placed on the border (see figure 16).

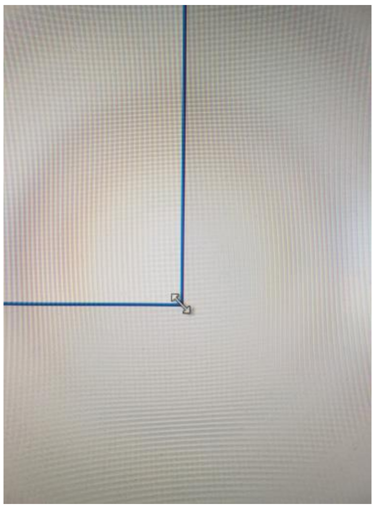
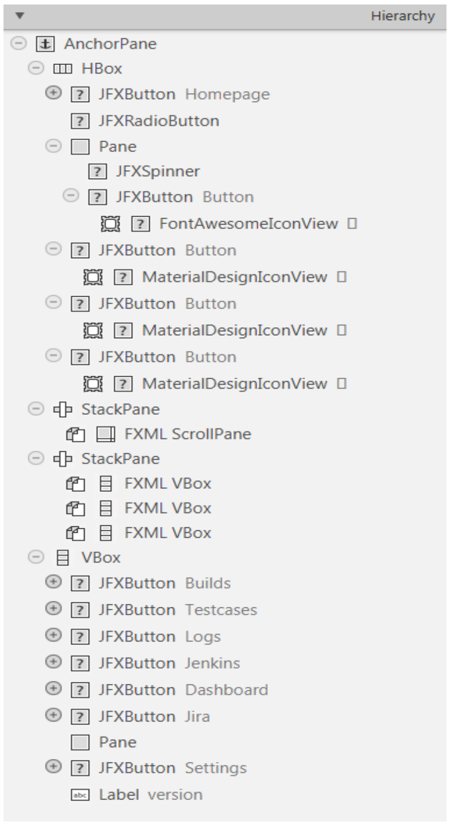


Figure 16. Mouse moved over south east corner on application window changing cursor type.

Mouse_dragged event will change the window sizes upon dragging the mouse by calculating new width and height from the screen, scene, and mouse event variables. After calculation, location of stage on screen and size of stage is set by JavaFX window library class. Additionally, there are restrictions to limit window minimum width and height. Window has minimum width and height for keeping the GUI components in the right places.

4.1.3 GUI component hierarchy

AnchorPane is the parent node for *GUIWindow.fxml*. *AnchorPane* was chosen as parent because it makes children of anchor pane be tied to the edges by *AnchorPane* constraints. This means that resizing the window also resizes child nodes inside anchor pane. At the start of the project, this was a problem. Nodes would not resize upon resizing the window. Hardcoding child node constraints enables them to look cohesive and they do not overlap with another node. *GUIWindow* has four important child nodes in its hierarchy as is shown in figure 17. *HBox* for the header pane, *StackPane* for showing content, *StackPane* for including searching functionality in *VBox* and a *VBox* for buttons with icons on left side of the window.



```
Hierarchy
AnchorPane
  HBox
    JFXButton Homepage
    JFXRadioButton
    Pane
      JFXSpinner
      JFXButton Button
        FontAwesomeIconView
    JFXButton Button
      MaterialDesignIconView
    JFXButton Button
      MaterialDesignIconView
    JFXButton Button
      MaterialDesignIconView
  StackPane
    FXML ScrollPane
  StackPane
    FXML VBox
    FXML VBox
    FXML VBox
  VBox
    JFXButton Builds
    JFXButton Testcases
    JFXButton Logs
    JFXButton Jenkins
    JFXButton Dashboard
    JFXButton Jira
    Pane
    JFXButton Settings
    Label version
```

Figure 17. Scene Builder view of node hierarchy in application's main window.

The header pane is made with *HBox*. *HBox* contains nodes used in undecorated window header which is styled with Nokia blue color. There is a button for *homepage*, radiobutton for checking connection status, *Pane* containing loading icon and previous button. Lastly, there is minimize, maximize, and close buttons. Clicking previous button highlights the previously selected icon and sets the content to previous content. If there is no previous content, previous button is disabled. Previous content and previous selected attributes are tracked by local variables such as *previousParent* and *previousSelected*. *PreviousSelected* gets the highlighted selection from looking up nodes with class ".selected" in CSS file. Whenever a node is selected, it is given a style class "selected". When it is not selected, style class is removed.

Since the stage is transparent, there are no existing header buttons for basic desktop application functionality. Usual header buttons are: max window button, minimize window button, and close window button. In this project, these buttons are manually added into the fxml file with help of Scene Builder and JFoenix. All three functionalities are in *JFXButtons* with graphic as its children. Graphic can have children of *imageView* or an icon from FontAwesomeFX library. These buttons are given a style class which is used in the CSS file to provide the white color as seen in figure 18.

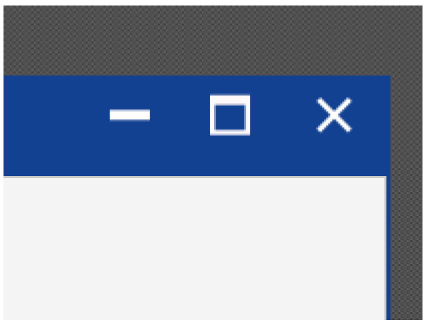


Figure 18. Buttons for minimize, full screen and close.

Adding the buttons into FXML file and coloring them does not give them any functionality. Minimize, maximize, and close buttons has their own methods with "@FXML" annotations. Methods are called *minimize*, *toggleFullscreen* and *quit*. They take *ActionEvent* class from JavaFX event library as a parameter. *ActionEvent* represents variety of events such as event of clicking a button or pressing a key. Minimize method sets the stage iconified and *toggleFullscreen* checks if stage is already maximized. If it is not maximized, it will get the visual bounds of the screen and set the stages width and height to match those values. Quit method is simple. Quit will close the stage and call system exit which will terminate the currently running JVM.

4.1.4 SearchPane

The GUI needs a searching functionality since there are a lot of data at disposal. As a solution, testers can search for builds and testcases with *SearchPane*. *SearchPane* is a custom made FXML file included into the *StackPane*.

Parent node of *SearchPane.fxml* is a *VBox*. *VBox* contains a text field for search input, a button for entering the input and a *TabPane* with two tabs for both software versions. Both tabs have children of *ListView*, which is a list for displaying the data. As displayed in Figure 19, there are two includes of *SearchPane* because two functionalities need it. Buttons “Builds and “Test cases” open the *SearchPane* displaying different data for each purpose. Each of these buttons has a toggle method in *SearchPaneContainer* class. Whenever toggle is called, the chosen *SearchPane* opens and sets itself to active *SearchPane*. If there is another *SearchPane* already open, it closes.

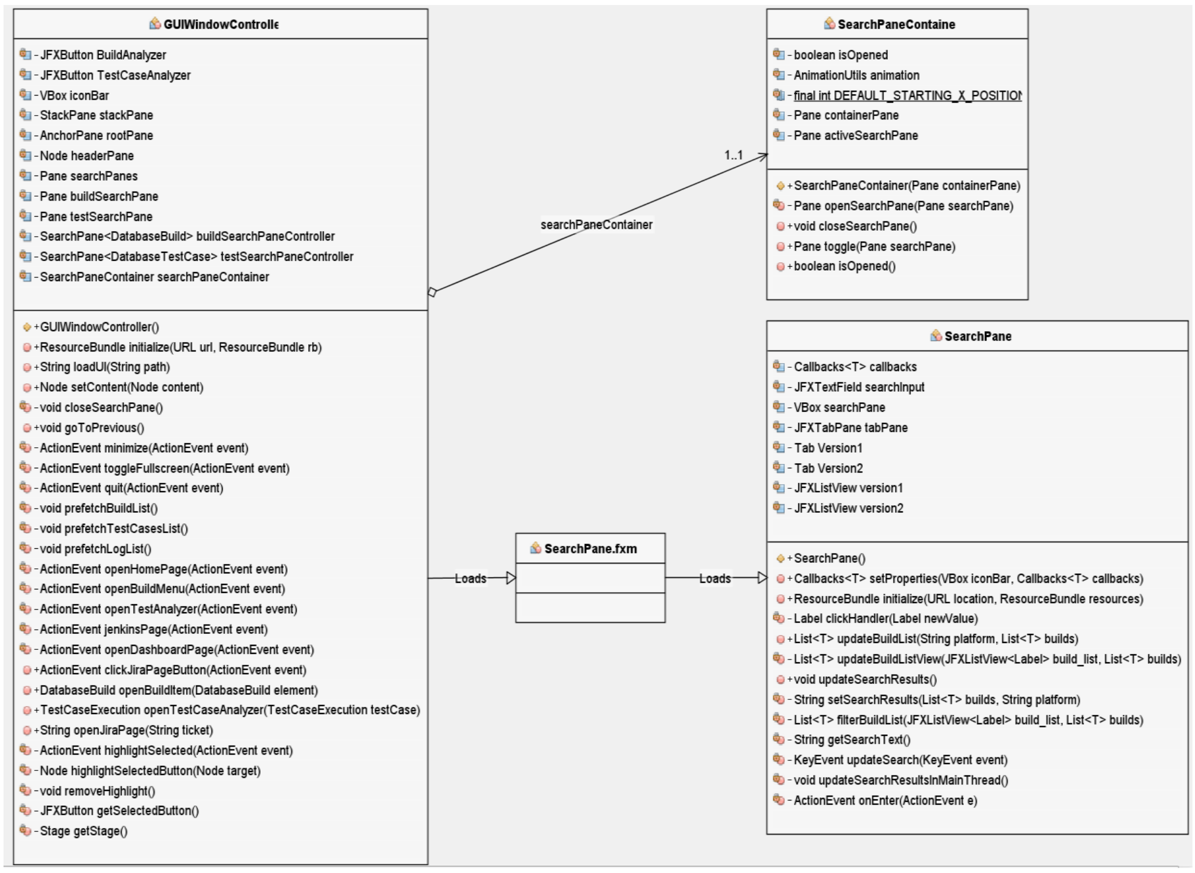

GUIWindowControlle
- JFXButton BuildAnalyzer
- JFXButton TestCaseAnalyzer
- VBox iconBar
- StackPane stackPane
- AnchorPane rootPane
- Node headerPane
- Pane searchPanes
- Pane buildSearchPane
- Pane testSearchPane
- SearchPane<DatabaseBuild> buildSearchPaneController
- SearchPane<DatabaseTestCase> testSearchPaneController
- SearchPaneContainer searchPaneContainer
+GUIWindowController()
+ResourceBundle initialize(URL url, ResourceBundle rb)
+String loadUI(String path)
+Node setContent(Node content)
- void closeSearchPane()
+void goToPrevious()
- ActionEvent minimize(ActionEvent event)
- ActionEvent toggleFullscreen(ActionEvent event)
- ActionEvent quit(ActionEvent event)
- void prefetchBuildList()
- void prefetchTestCasesList()
- void prefetchLogList()
- ActionEvent openHomePage(ActionEvent event)
- ActionEvent openBuildMenu(ActionEvent event)
- ActionEvent openTestAnalyzer(ActionEvent event)
- ActionEvent jenkinsPage(ActionEvent event)
- ActionEvent openDashboardPage(ActionEvent event)
+ActionEvent clickJiraPageButton(ActionEvent event)
+DatabaseBuild openBuildItem(DatabaseBuild element)
+TestCaseExecution openTestCaseAnalyzer(TestCaseExecution testCase)
+String openJiraPage(String ticket)
- ActionEvent highlightSelected(ActionEvent event)
- Node highlightSelectedButton(Node target)
- void removeHighlight()
- JFXButton getSelectedButton()
- Stage getStage()
searchPaneContainer
1..1
SearchPaneContaine
- boolean isOpened
- AnimationUtils animation
- final int DEFAULT_STARTING_X_POSITION
- Pane containerPane
- Pane activeSearchPane
+SearchPaneContainer(Pane containerPane)
- Pane openSearchPane(Pane searchPane)
+void closeSearchPane()
+Pane toggle(Pane searchPane)
+boolean isOpened()
Loads
SearchPane.fxm
Loads
SearchPane
- Callbacks<T> callbacks
- JFXTextField searchInput
- VBox searchPane
- JFXTabPane tabPane
- Tab Version1
- Tab Version2
- JFXListView version1
- JFXListView version2
+SearchPane()
+Callbacks<T> setProperties(VBox iconBar, Callbacks<T> callbacks)
+ResourceBundle initialize(URL location, ResourceBundle resources)
- Label clickHandler(Label newValue)
+List<T> updateBuildList(String platform, List<T> builds)
- List<T> updateBuildListView(JFXListView<Label> build_list, List<T> builds)
+void updateSearchResults()
- String setSearchResults(List<T> builds, String platform)
- List<T> filterBuildList(JFXListView<Label> build_list, List<T> builds)
- String getSearchText()
- KeyEvent updateSearch(KeyEvent event)
- void updateSearchResultsInMainThread()
- ActionEvent onEnter(ActionEvent e)

Figure 19. UML-chart showing *SearchPane* functionality.

SearchPane appears with another animation called *transition animation*. This animation slides the *SearchPane* from left side under the icon pane into the front of the *StackPane* (see figure 20.). *SearchPaneController.java* is the controller of *SearchPane.fxml* and it handles filtering the list and updating it. Clicking on an item on the *ListView* opens content on *StackPane* depending on which *SearchPane* was open. The content will be further explained in data visualization subchapter.

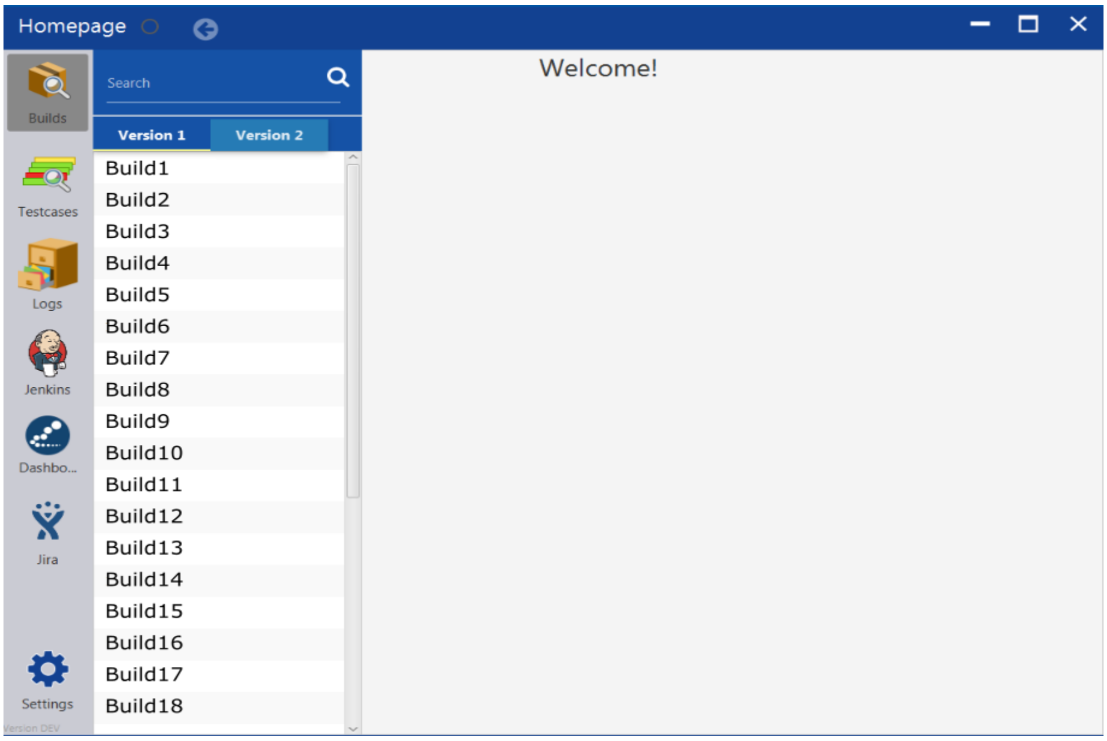


Figure 20. SearchPane.fxml included in *GUIWindow.fxml*.

4.2 Client data fetching

To use the server data in client, data must have serializable object classes. These classes are called models in this project. There is a model for each database collection. Models does not just contain the data but also helper methods. Models are used both in client and server. Model module also has a command class. Command class contains enums, which are data types for setting variables to predefined constants [18].

In order to show data in GUI, client must communicate with the server to get the data. *Client.java* is used for requesting the data from server. Client has a method called *request* for it. *Request* uses command enum to tell the server what it wants server to do.

Server receives command and fetches data from database based on command it got. Some commands need extra parameters that are send with the command enum. For example, the *FETCH_BUILD_LIST* command takes a search query as only parameter (see listing 5)

```
Client.request(CommandEnum.FETCH_BUILD_LIST, (List<DatabaseBuild> response) ->
{
     buildSearchPaneController.updateBuildList("version1", response);
}, new Parameters("PRODUCT", "VERSION1"));
```

Listing 5. *Request* method taking command enum and parameters for fetching a list of a model type *DatabaseBuild*. Then updating *SearchPane* with list of builds.

Also, request can take a callback function that will be called after response from server is received. If no callback has been given, *client.java* returns a future object. Future object has a single method called get. When get is called, the caller thread will sleep until the response has been received. Afterward, response is returned to caller. Difference of callback and future is that callback will notify the caller once the response is received. Caller can do other work while response is not yet received. Future will wait and block until response is received or do some work and occasionally check if response is received. Figure 21 shows how the process of data fetching happens in client.

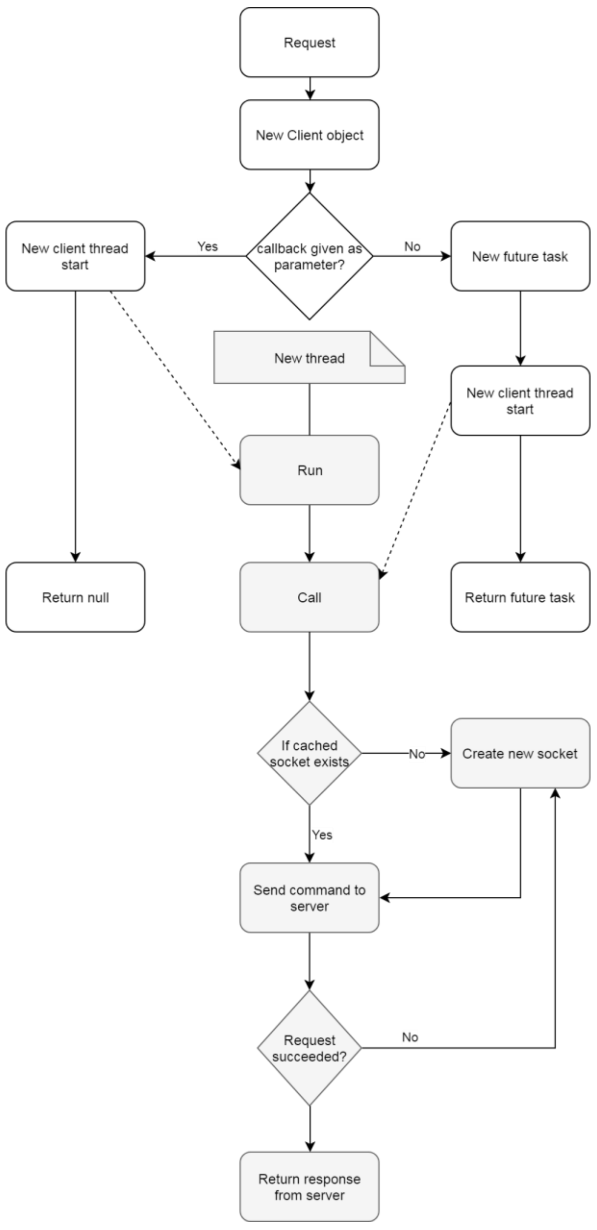


Figure 21. Flowchart of data fetching.

4.2.1 Client-server communication

In order to obtain data transfer between client and server, sockets need to be created. Both client process and server process must establish their own socket for both to send and receive data. Only a client socket needs to know the address it is connecting to. On creation, client socket is given a server host address and a port number as parameter. Server socket does not need to know existence of client. Server socket only listens for connections with listen system call. After finding a connection, it accepts a connection request from client with accept system call. Accept call blocks until client relates to the server. After connection is formed, client and server can start sending and receiving data. However, both need the socket type to be the same.

Widely used socket types are stream socket and a datagram socket [19.]. Stream socket uses Transmission Control Protocol (TCP) for sending packets through and checking if they are received. Datagram socket uses Unix Datagram Protocol (UDP) which sends packets faster than TCP. However, it lacks the benefit of handshaking. Lack of handshaking means that it does not check if messages are received and therefore can drop packets. In this project, client socket and server socket use TCP with *ObjectInputStream* and *ObjectOutputStream*. *ObjectInputStream* can read data in form of serialized object and *ObjectOutputStream* can write serialized object data. Therefore, self-made models must be implemented as serializable. Also, strings and array lists are objects which can be input and output in object streams.

As the application is started, *GUIWindowController* initialization method is called. Initialization calls for methods requesting data to the *SearchPanes*. Each of these methods fetches a list of data models: List of builds and test cases. Request for data from client goes through internet into the server which fetches data from database. Then data goes from server back to client. Route might seem long but data fetching only takes a few milliseconds. High speed enables data to be instantly accessible upon starting the client.

4.3 Data visualization

After client software has received data from a server, it can be visualized with help of models. Visualizing data into different charts and tables helps end-users to better understand it. Pie chart visualizes build overall status and a *TreeTableView* visualizes all the test cases and their details in one build. Also, testers can see test case history data, which is visualized with a line graph.

Selecting an item from any of the three *SearchPane*'s *ListView* reads the content of an FXML file into the middle *StackPane*. Each FXML file is different and has a controller class dedicated to them. However, there is a problem that the *SearchPane* cannot tell directly the new controller what search item was selected. Solution for this is to make all communications between controllers go through the *GUIWindowController* (see Figure 22.). To make this possible, every controller must extend an abstract controller class that stores reference to the main controller.

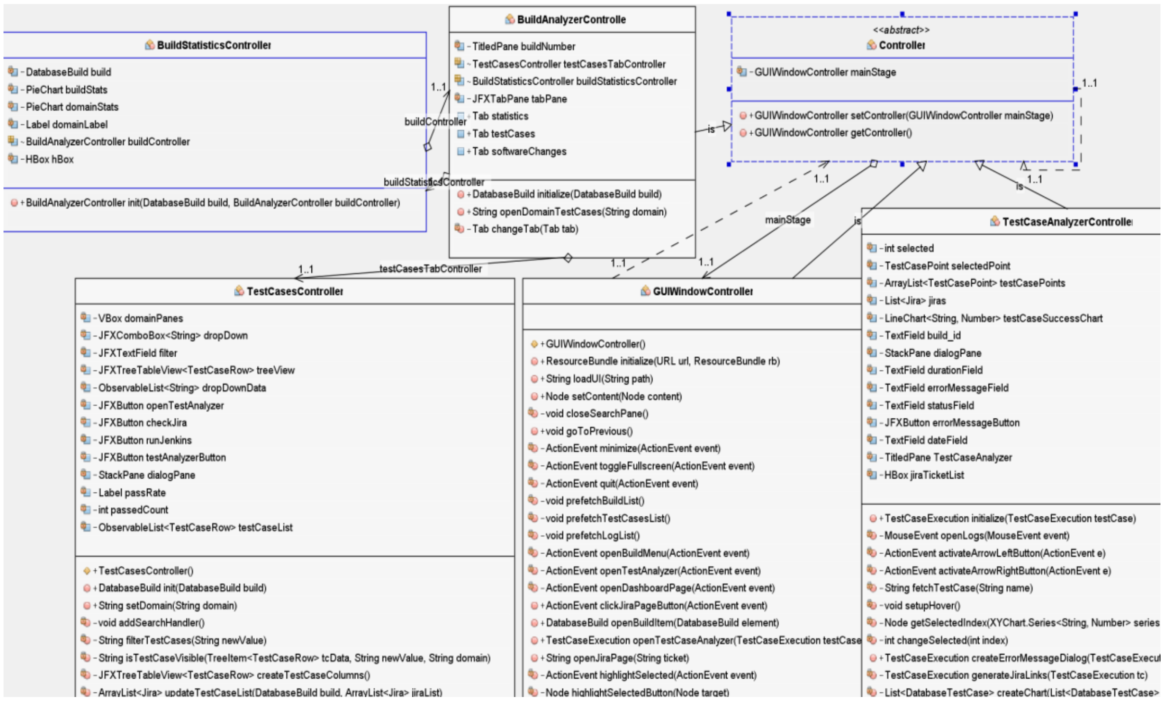

BuildStatisticsController
- DatabaseBuild build
- PieChart buildStats
- PieChart domainStats
- Label domainLabel
~ BuildAnalyzerController buildController
- HBox hBox
+ BuildAnalyzerController init(DatabaseBuild build, BuildAnalyzerController buildController)
BuildAnalyzerControlle
- TitledPane buildNumber
~ TestCasesController testCasesTabController
~ BuildStatisticsController buildStatisticsController
- JFXTabPane tabPane
+ Tab statistics
+ Tab testCases
+ Tab softwareChanges
+ DatabaseBuild initialize(DatabaseBuild build)
+ String openDomainTestCases(String domain)
- Tab changeTab(Tab tab)
<<abstract>>
Controller
- GUIWindowController mainStage
+ GUIWindowController setController(GUIWindowController mainStage)
+ GUIWindowController getController()
1..1
buildController
buildStatisticsController
is
1..1
mainStage
is
1..1
is
1..1
testCasesTabController
1..1
1..1
TestCasesController
- VBox domainPanes
- JFXComboBox<String> dropDown
- JFXTextField filter
- JFXTreeTableView<TestCaseRow> treeView
- ObservableList<String> dropDownData
- JFXButton openTestAnalyzer
- JFXButton checkJira
- JFXButton runJenkins
- JFXButton testAnalyzerButton
- StackPane dialogPane
- Label passRate
- int passedCount
- ObservableList<TestCaseRow> testCaseList
+ TestCasesController()
+ DatabaseBuild init(DatabaseBuild build)
+ String setDomain(String domain)
- void addSearchHandler()
- String filterTestCases(String newValue)
- String isTestCaseVisible(TreeItem<TestCaseRow> tcData, String newValue, String domain)
- JFXTreeTableView<TestCaseRow> createTestCaseColumns()
- ArrayList<Jira> updateTestCaseList(DatabaseBuild build, ArrayList<Jira> jiraList)
GUIWindowController
+ GUIWindowController()
+ ResourceBundle initialize(URL url, ResourceBundle rb)
+ String loadUI(String path)
+ Node setContent(Node content)
- void closeSearchPane()
+ void goToPrevious()
- ActionEvent minimize(ActionEvent event)
- ActionEvent toggleFullscreen(ActionEvent event)
- ActionEvent quit(ActionEvent event)
- void prefetchBuildList()
- void prefetchTestCasesList()
- void prefetchLogList()
- ActionEvent openBuildMenu(ActionEvent event)
- ActionEvent openTestAnalyzer(ActionEvent event)
- ActionEvent openDashboardPage(ActionEvent event)
+ ActionEvent clickJiraPageButton(ActionEvent event)
+ DatabaseBuild openBuildItem(DatabaseBuild element)
+ TestCaseExecution openTestCaseAnalyzer(TestCaseExecution testCase
+ String openJiraPage(String ticket)
- ActionEvent highlightSelected(ActionEvent event)
- Node highlightSelectedButton(Node target)
TestCaseAnalyzerControlle
- int selected
- TestCasePoint selectedPoint
- ArrayList<TestCasePoint> testCasePoints
- List<Jira> jiras
- LineChart<String, Number> testCaseSuccessChart
- TextField build_id
- StackPane dialogPane
- TextField durationField
- TextField errorMessageField
- TextField statusField
- JFXButton errorMessageButton
- TextField dateField
- TitledPane TestCaseAnalyzer
- HBox jiraTicketList
+ TestCaseExecution initialize(TestCaseExecution testCase)
- MouseEvent openLogs(MouseEvent event)
- ActionEvent activateArrowLeftButton(ActionEvent e)
- ActionEvent activateArrowRightButton(ActionEvent e)
- String fetchTestCase(String name)
- void setupHover()
- Node getSelectedIndex(XYChart.Series<String, Number> series
- int changeSelected(int index)
+ TestCaseExecution createErrorMessageDialog(TestCaseExecut
- TestCaseExecution generateJiraLinks(TestCaseExecution tc)
- List<DatabaseTestCase> createChart(List<DatabaseTestCase>

Figure 22. All controllers used for visualizing data.

When loading FXML file, the *loadUI* method will give the main controller to the newly created controller object (see listing 6). *LoadUI* method takes the path of FXML file as argument, sets the content and returns the controller object.

```
public void openBuildItem(DatabaseBuild element) {
    BuildAnalyzerController buildController = (BuildAnalyzerController)
loadUI("builds/BuildAnalyzer.fxml");
    buildController.initialize(element);
}

public Controller loadUI(String path) {
    Controller controller = null;
    closeSearchPane();
    Parent root;
    path = "com/nokia/ourproject/" + path;
    try {
        if (null == path) {
            root = FXMLLoader.load(getClass().getResource(path));
        } else {
            FXMLLoader loader = new FXMLLoader(getClass().getResource(path));
            root = (Parent) loader.load();
            controller = loader.getController();
            controller.setController(this);
        }
    }
    catch (IOException ex) {
Logger.getLogger(GUIWindowController.class.getName()).log
       (Level.SEVERE, null, ex);
        System.exit(0);
        return null;
    }
    setContent(root);
    return controller;
}
```

Listing 6. Loading *BuildAnalyzer.fxml* and its controller class

4.3.1 Visualization of build results

Main functionalities in *BuildAnalyzer* is to show overall status of test case results in one build. End-users may observe pass rate summary of one build and pass rates in each product domain. Also, they can view a list of executed test cases and details about them. *BuildAnalyzer.fxml* contains *TitledPane* as a parent node and a *TabPane* as its child. *BuildAnalyzerController* sets text to *TitledPane* for showing the selected version and selected build. *TabPane* contains two tabs, statistics, and test cases. They have controllers, *BuildStatisticsController* and *TestCasesController*. Both of controllers initialize method is called in *BuildAnalyzerController* and they are given the build object. *BuildStatisticsController* creates two pie charts from build data. Figure 23 shows how pie charts visualize overall status of one software build test results. Pie charts are created in a new thread which executes a new runnable class. This is because creating the pie charts with the data from build object takes a few seconds and therefore it would freeze the GUI if it was done in GUI thread.

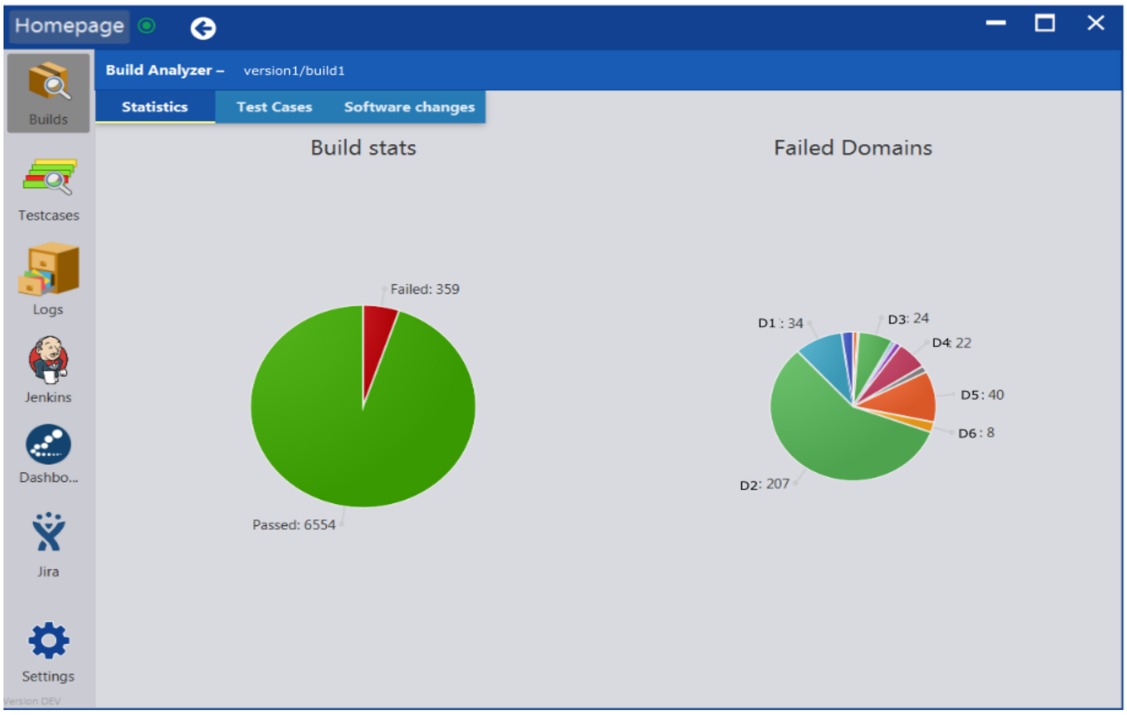
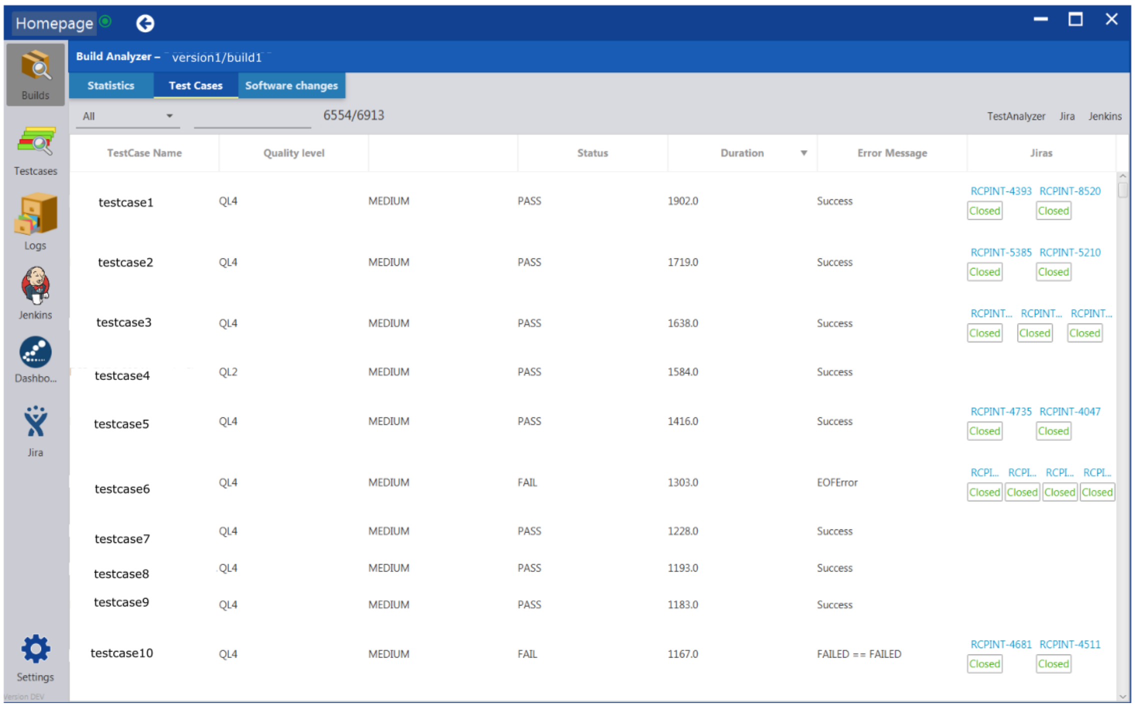


Figure 23. Pie charts visualizing build overall status. Slices are highlighted, and percentage of slice is shown when cursor is hover over it.

4.3.2 Visualization of test case results

Test cases tab shows all the test cases of one build in a *TreeTableView*. Like *BuildStatisticsController*, data is updated to the *TreeTableView* in another thread since there is lots of data. Also, new thread makes a request for all the Jira tickets. *TreeTableView* shows Jira ticket, if there has been Jira ticket made for the test case. Jira tickets shows the ticket ID in a hyperlink and status of the Jira. Hyperlink leads to opening the Jira website in the application (Jira icon on left side) and showing the ticket which was clicked. This enables testers to know whether there has been made a Jira ticket for the test case failure before.

Test cases can be filtered by searching for items in the text field or clicking on columns. Also, dropdown menu filters the list by domain. For example, figure 24 displays duration column filtered which enables testers to see which test cases took the longest time to execute. With this test case list, testers can easily see test results of one build. This is a key feature of this tool since it gives a good overview of how the tests have gone. If testers need to check more details of one test case, they can select a test case by selecting a row and clicking *TestAnalyzer*. *TestAnalyzer* button leads to showing the test case history of selected test case which is represented by testcases icon.



6554/6913

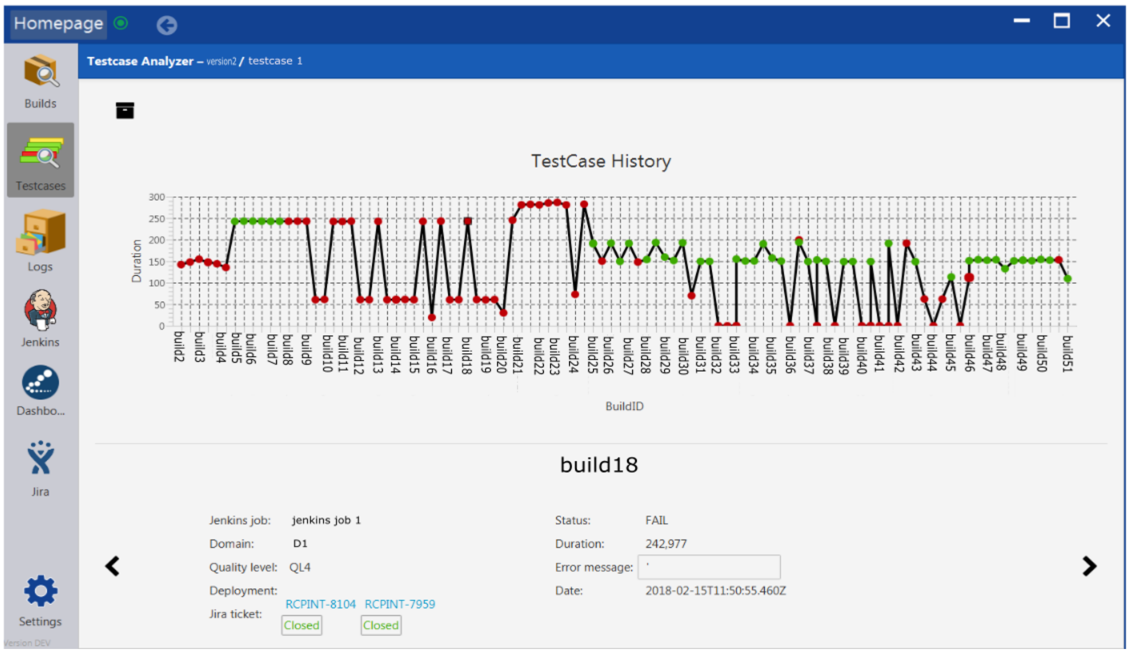
TestCase Name	Quality level		Status	Duration	Error Message	Jiras
testcase1	QL4	MEDIUM	PASS	1902.0	Success	RCPINT-4393 Closed RCPINT-8520 Closed
testcase2	QL4	MEDIUM	PASS	1719.0	Success	RCPINT-5385 Closed RCPINT-5210 Closed
testcase3	QL4	MEDIUM	PASS	1638.0	Success	RCPINT... Closed RCPINT... Closed RCPINT... Closed
testcase4	QL2	MEDIUM	PASS	1584.0	Success	
testcase5	QL4	MEDIUM	PASS	1416.0	Success	RCPINT-4735 Closed RCPINT-4047 Closed
testcase6	QL4	MEDIUM	FAIL	1303.0	EOFError	RCPI... Closed RCPI... Closed RCPI... Closed RCPI... Closed
testcase7	QL4	MEDIUM	PASS	1228.0	Success	
testcase8	QL4	MEDIUM	PASS	1193.0	Success	
testcase9	QL4	MEDIUM	PASS	1183.0	Success	
testcase10	QL4	MEDIUM	FAIL	1167.0	FAILED == FAILED	RCPINT-4681 Closed RCPINT-4511 Closed

Figure 24. TreeTableView visualizing list of all the test cases in one build. Test cases are filtered by duration showing the longest test cases.

4.3.3 Test case history

TestAnalyzer is a feature in this tool that shows how the test case has performed in last hundred builds. Testers can see how one test case has performed, details of it and the variance of its duration. Builds are sorted by date when they were executed. In figure 25, newest build is on the right side of the chart.

TestAnalyzer is shown by *TestCaseAnalyzer.fxml* and it functions by *TestAnalyzerController.java*. When *TestAnalyzerController* is initialized, a list of test case objects and Jira tickets are requested from database. After response is received, line chart is filled with the given data. Line chart needs data for x and y axes. It is given a string for x axis which represents the build id. Y axis gets a number for duration. After filling line chart with data, testers can select a build by clicking a dot on line chart or by using left and right key arrows. Dots are red if test case is failed and green if test case is passed. Details of test case execution in one build is shown under the line chart and they change every time a new build is selected. Details have been sorted to a GridPane. Each detail is an uneditable text field except error message. If the test case is failed, error message is shown in a dialog. Dialog appears in middle of the screen on top of everything else. This is because the error message can be a long text and is unable to fit in a text field fully.



build18

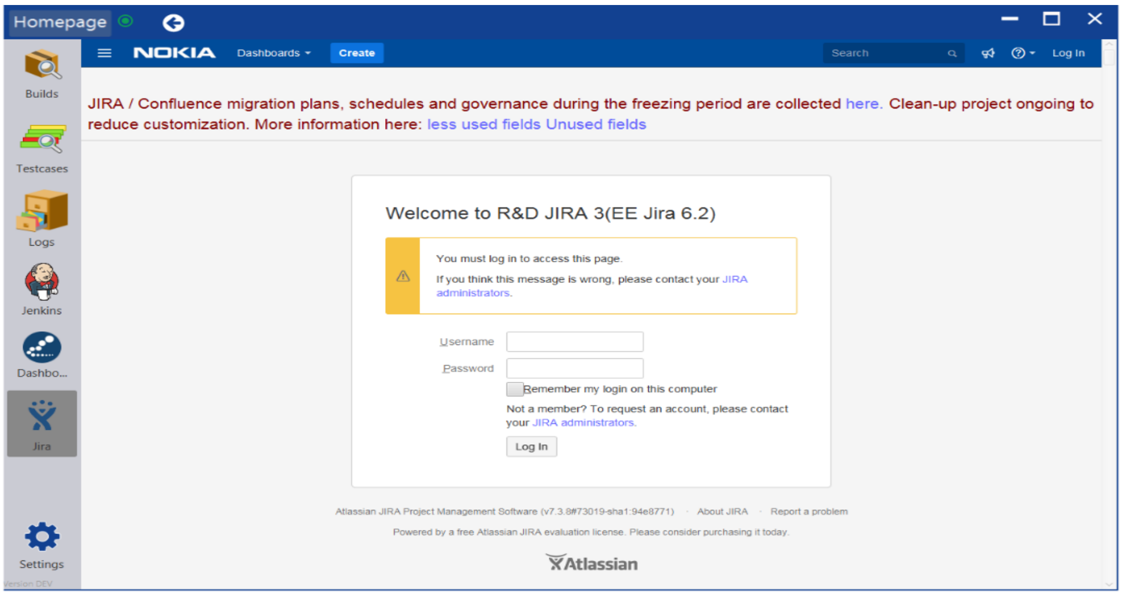
Jenkins job:	jenkins job 1	Status:	FAIL
Domain:	D1	Duration:	242,977
Quality level:	QL4	Error message:	'
Deployment:		Date:	2018-02-15T11:50:55.460Z
Jira ticket:	RCPINT-8104 Closed RCPINT-7959 Closed		

Figure 25. Line chart displaying test case history of one test case and details about it.

4.3.4 Support of other tools

Jenkins, Dashboard and Jira icons represent other tools used in testers' daily work. Testers do not necessarily need to open a browser since everything useful is implemented in this tool. Each of the icons is implemented almost the same way into this new analysis tool with *WebView*. *WebView* is one of JavaFX libraries and its main purpose is to be a node for *WebEngine* and therefore show its content. *WebEngine* acts like a browser inside an application. *WebEngine* loads web pages inside application window. It can show one web page at a time and it can also run JavaScript on web pages. This way, there is communication between application and JavaScript code of the page. Web page features work similarly inside the application as they would in a browser as seen in figure 26.

Jenkins, Dashboard and Jira *WebView* objects are created in their respective classes. Jenkins.java for Jenkins page *WebView*, Dashboard.java for Dashboard *WebView* and JiraPage.java for Jira *WebView*. Each of these classes are like each other, except for the URL path that the *WebEngine* takes as a parameter in class constructor. Each of these classes are called in their respective methods in *GUIWindowController.java*, when clicking on their icons. Upon calling these methods, new class object is made and set as parameter to previously mentioned *setContent* method.



JIRA / Confluence migration plans, schedules and governance during the freezing period are collected here. Clean-up project ongoing to reduce customization. More information here: less used fields Unused fields

Welcome to R&D JIRA 3(EE Jira 6.2)

You must log in to access this page.

If you think this message is wrong, please contact your JIRA administrators.

Username

Password

Remember my login on this computer

Not a member? To request an account, please contact your JIRA administrators.

Log In

Atlassian JIRA Project Management Software (v7.3.8#73019-sha1:94e8771) · About JIRA · Report a problem

Powered by a free Atlassian JIRA evaluation license. Please consider purchasing it today.

Atlassian

Figure 26. Showing a Jira page inside the application window with *WebView*.

5 Conclusion

The purpose of this thesis was to develop client software as part of new test analysis tool. The idea of the test analysis tool is to reduce the time of analysis and thus improve and facilitate the analysis work. The client software would have an easy-to-use graphical user interface, would be able to retrieve data from the server, visualize the test result data, and support other tools. In addition to these, other features described in user stories were implemented as well.

Client software was implemented with Java and JavaFX. Java was chosen as programming language for this project and JavaFX library for building graphical user interface. JavaFX utilizes a declarative markup language called FXML for defining the user interface components. Each FXML file has a controller.java file for containing the application logic code. This means that FXML files only show the components in graphical user interface, but controllers define how they work.

All the thesis objectives were successfully met and most of the features described in user stories was implemented. Test case history filtered by domains and tracking the changes in software repository was not implemented in the work of this thesis. Most of the end users using this tool are testers from product continuous integration team. They benefit greatly from this tool as all the information and functions required for test analysis work are combined within the tool. Testers can check overall test case pass rate, details of test cases in any software build or history performance of any test case. If the wanted build or test case name is not in *SearchPane*, they can write build id or test case name into search field and press enter for fetching the desired data.

There are still some challenges and development targets in the client software for future development. To further quicken analysis time, the tool could help tester make a Jira ticket by pre-filling data from failed test case into the ticket. Another improvement idea is to allow the tool to communicate with Jenkins server for running the failed test cases again. Running the failed test cases again helps to see if there was a connection timeout during test case execution instead of a real bug. These features are somewhat already implemented, but not finished.

References

1. Kraftic Inc. 2018. Web-page. *Continuous integration and delivery solutions*. <https://www.kraftic.com/services/Continuous-Integration-and-Delivery-Solutions>. Fetched 16.7.2018.

2. Prince, Suzie. 2016. *The Product Managers' Guide to Continuous Deliver and DevOps*. Web-article. Mind the product. <https://www.mindtheproduct.com/2016/02/what-the-hell-are-ci-cd-and-devops-a-cheatsheet-for-the-rest-of-us/>. Fetched 17.7.2018.

3. Pittet, Sten. 2018. *Continuous integration vs. continuous delivery vs. continuous deployment*. Web-article. Atlassian. <https://www.atlassian.com/continuous-delivery/ci-vs-ci-vs-cd>. Fetched 24.7.2018.

4. Jenkins. 2018. Web-page. Jenkins. <https://jenkins.io/>. Fetched 25.7.2018.

5. Robot Framework. 2018. Web-page. <http://robotframework.org/>. Fetched 16.7.2018.

6. Piironen, Janne. 2011. *Robot Framework Plugin*. Web-page. Wiki.jenkins. <https://wiki.jenkins.io/display/JENKINS/Robot+Framework+Plugin>. Fetched 17.7.2018.

7. Oracle. 2017*. About the Java Technology*. Web-page. <https://docs.oracle.com/javase/tutorial/getStarted/intro/definition.html>. Fetched 21.8.2018.

8. Smith, Roger. 2016. *Why Java is the most popular programming language*. Web-article. Theserverside. <https://www.theserverside.com/feature/Why-Java-is-the-most-popular-programming-language>. Fetched 5.5.2018.

9. Net-informations. 2018. *What is Java virtual machine*. Web-page. Net-informations. <http://net-informations.com/java/intro/jvm.htm>. Fetched 6.6.2018.

10. Programiz. *Java tutorial*. Web-page. <https://www.programiz.com/java-programming/jvm-jre-jdk>. Fetched 7.6.2018.

11. Wikipedia. 2018. *Rich Internet application*. Web-page. <https://en.wikipedia.org/wiki/Rich_Internet_application>.Last edited 16.5.2018. Fetched 21.5.2018.

12. Wikipedia. 2018. *JavaFX*. Web-page. <https://en.wikipedia.org/wiki/JavaFX>. Last edited 7.5.2018. Fetched 21.5.2018.

13. Leahy, Paul. 2017. *What Is JavaFX*. Web-article. ThoughtCo. <https://www.thoughtco.com/what-is-javafx-2034192>. Fetched 23.5.2018.

14. Tutorialspoint. 2018*. JavaFX – Architecture*. Web-page. <https://www.tutori-alspoint.com/javafx/javafx_architecture.htm>. Fetched 24.5.2018.

15. Baskirt, Onur. 2016. *Getting Started with JavaFX*. Web-page. Swtestacademy. <https://www.swtestacademy.com/getting-started-with-javafx/>. Fetched 25.5.2018.

16. Hommel, Scott. 2013. *Working with the JavaFX Scene Graph*. Web-page. Ora-cle. <https://docs.oracle.com/javafx/2/scenegraph/jfxpub-scenegraph.htm>. Fetched 25.5.2018.

17. W3schools. 2015. *Introduction to XML*. Web-page. <https://www.w3schools.com/xml/xml_whatis.asp>. Fetched 28.5.2018

18. Oracle. 2017. *Enum Types*. Web-page. Docs.oracle.<https://docs.ora-cle.com/javase/tutorial/java/javaOO/enum.html> Fetched 7.8.2018.

19. Ingalls, Robert. *Sockets Tutorial*. Web-page. Cs.rpi.edu. <http://www.cs.rpi.edu/~moorthy/Courses/os98/Pgms/socket.html> Fetched 8.8.2018.