jamk.fi

Model-based acceptance testing as a part of continuous delivery, Case: Contriboard

Petri Matilainen

Bachelor's Thesis
December 2014

Degree Programme in Information Technology
Technology, Communication and Transport

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

Description

Author Matilainen, Petri	Type of publication Bachelor's thesis	Date 12.12.2014
		Language English
	Pages 65 + 17	Permission for web publication (X)

Title
Model-based acceptance testing as a part of continuous delivery
Case: Contriboard

Degree programme
Software Engineering

Tutor
Mieskolainen, Matti

Assigned by
N4S@JAMK
Rintamäki, Marko

Abstract

The thesis was assigned by N4S@JAMK project within JAMK University of Applied Sciences. The documentation was published as one contribution from JAMK to Digile's Need for Speed research program.

The thesis focused on exploring how model-based acceptance testing could be implemented into a continuous delivery chain Corolla v1.1 and used in its reference product Contriboard. Both Corolla v1.1 and Contriboard are developed by N4S@JAMK team.

The tests were made and executed with Ixonos Visual Test tool and made to build from Jenkins. As the product for various reasons did not have a currently working developer build, the production build was used as a system under test to prove that the implementation works.

As a result, the product had a working testing environment and tests which could be launched from the Jenkins. Later when development build is back online, the tester can be easily changed to use it as a system under test. The test model was intentionally made lightweight because of the incoming major changes; however, it can be very easily expanded in the future.

Keywords
Model-based testing, Acceptance testing, Contriboard, N4S@JAMK

Miscellaneous

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

Kuvailulehti

Tekijä Matilainen, Petri	Julkisun laji Opinnäytetyö	Päivämäärä 12.12.2014
		Julkaisun kieli Englanti
	Number of pages 65+17	Verkkojulkaisulupa myönnetty (X)

Työn nimi
Mallipohjainen hyväksyntätestaus osana jatkuvaa julkaisua
Case: Contriboard

Koulutusohjelma
Ohjelmistotekniikan koulutusohjelma

Työn ohjaaja
Mieskolainen, Matti

Toimeksiantajat
N4S@JAMK
Rintamäki, Marko

Tiivistelmä

Opinnäytetyön toimeksiantaja oli N4S@JAMK -projekti Jyväskylän ammattikorkeakoulusta. Dokumentaatio julkaistiin JAMK:n tuloksena Digilen Need for Speed -tutkimusohjelmassa.

Työn tavoitteena oli tutkia, kuinka mallipohjainen hyväksyntätestausprosessi saataisiin integroitua Corolla v1.1 jatkuvan julkaisun ketjuun sekä toteutettua Contriboard referenssituotteen osalta. Corolla v1.1 sekä Contriboard ovat molemmat N4S@JAMK-tiimin tuotoksia.

Testit toteutettiin sekä ajettiin Ixonosin Visual Test -työkalulla ja liitettiin osaksi Jenkins tehtävienhallintaa. Koska tuotteesta ei sattuneista syistä ollut toimivaa kehitysversiota saatavilla, käytettiin testauksen todentamisessa sen hetkistä tuotantoversiota.

Työn tuloksena tuotteelle saatiin toimiva testausympäristö sekä testit jotka pystyttiin ajamaan Jenkinsin kautta. Myöhemmin kun kehitysversio on jälleen saatavilla, testaus saadaan helposti kohdennettua siihen. Testimalli tehtiin tarkoituksella kevyeksi, koska tuotteeseen on tulossa suuria muutoksia. Mallin laajennus onnistuu kuitenkin näiden jälkeen helposti.

Avainsanat (asiasanat)
Mallipohjainen testaus, Hyväksyntätestaus, Contriboard, N4S@JAMK

Muut tiedot

Table of Contents

Figures

Acronyms

BDD	Behavior Driven Development
CDel	Continuous Delivery
CDep	Continuous Deployment
CI	Continuous Integration
GUI	Graphical User Interface
ISTQB	International Softare Testing Qualifications Board
IVT	Ixonos Visual Test
JAMK	JAMK University of Applied Sciences
N4S	Need for Speed -program
OAT	Operational Acceptance Testing
SaaS	Software as a Service
SSH	Secure Shell
SUT	System Under Test
UAT	User Acceptance Testing
UI	User Interface
VM	Virtual Machine

1 Introduction

1.1 JAMK as a part of N4S research program

N4S@JAMK is a project started by JAMK University of Applied Sciences at Jyväskylä, Finland. It is a part of Need for Speed (N4S) which is a four year long program hosted by Digile and funded by Tekes. Long-term goal of the program is to provide tools and methods that speed up the work process for other software and ICT companies. (N4S-program 2014.)

Program currently has 11 large industrial organizations, 15 small and medium-sized enterprises and 10 research institutes and universities as partners working on three major areas of focus (N4S-program 2014.):

Delivering value in real-time: The Finnish software industry has taken a new direction in business by making the organizations lean towards more value-driven and adaptive real-time value model. Transformation is supported by required assets and technical infrastructure built especially for the needs.

Deep customer insight (increased profit): Assets and new technical infrastructure with various sources of information and collected data is utilized to gain and apply deeper insight to what customers need and how they behave. With this knowledge it is possible to improve sales and significantly increase the returns on investments in the development of both products and services.

Mercury Business (finding new income): New approach to growing business by behaving like liquid mercury. Constantly flowing to new trends and aggressively finding business opportunities in new markets with as little effort as possible. Key for success is to adapt to new conditions and actively change and expand focus.

1.2 Objectives of Thesis

Objectives of the thesis were to get familiar with model-based testing paradigm, how to use it in acceptance testing and how to implement both of them to continuous delivery chain of an existing product.

N4S@JAMK is developing a production environment called Corolla v1.1 and a web-based application named Contriboard as a reference product to it for the N4S program and was in a need of testing solutions for the already existing continuous delivery chain. Main focus for the thesis was to implement lightweight model for an acceptance testing and prove that it can be integrated to the Corolla v1.1 development chain.

Alongside with the actual implementation, other objectives were to explore general consensus of software testing, and also do collaboration with Ixonos by using their currently in development testing software named Ixonos Visual Test to create the actual tests for the implementation.

2 Contriboard

2.1 In general

Contriboard is a web-based service where user can create boards, populate them with tickets and share them with external URLs for other users. Users are divided into two types (Contriboard 2014.):

Power users: User has an account with free trial or valid payment plan. User has a workspace which contains all his or her boards with options for editing or removing them and creating new ones (see Figure 1 for GUI example). In specific board user can create, edit, remove and move tickets. User also has an option to open a specific board to public and share the given URL to guest users. Currently the creation of power user account is free since the payment model has not been implemented yet.

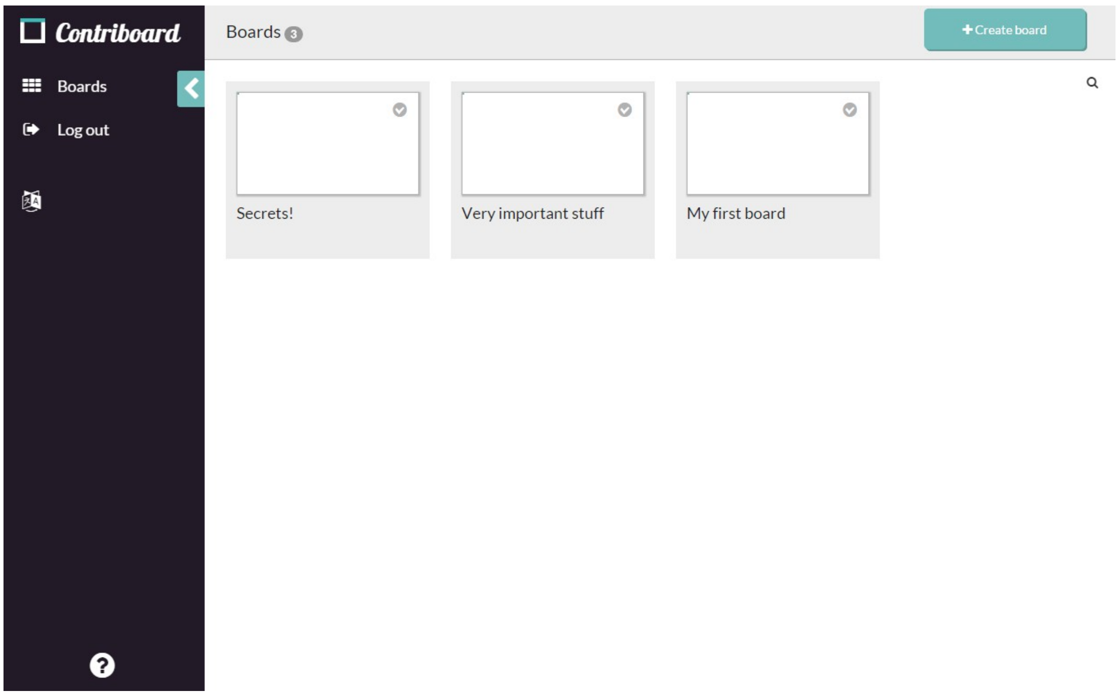


Figure 1: Contriboard UI - Workspace

Guests: User does not have an account, instead he or she gets a URL from power user and is able to use the specific board as long as it is open for public. When user goes to the board, he or she is prompted to give a name or an alias which is used to log the performed actions. User can then create, edit, remove and move tickets in the shared board (see Figure 2 for GUI example).

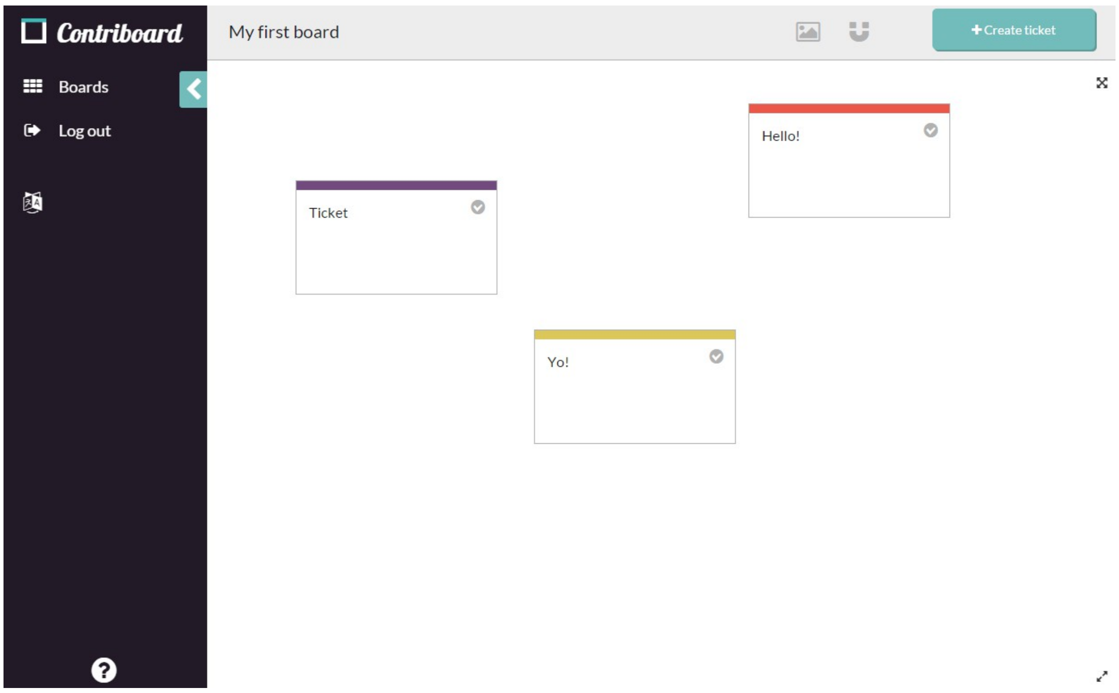


Figure 2: Contriboard UI - Board

2.2 Architecture

Contriboard's architecture contains following components (see Figure 3, note that repositories in GitHub are with the old Teamboard name):

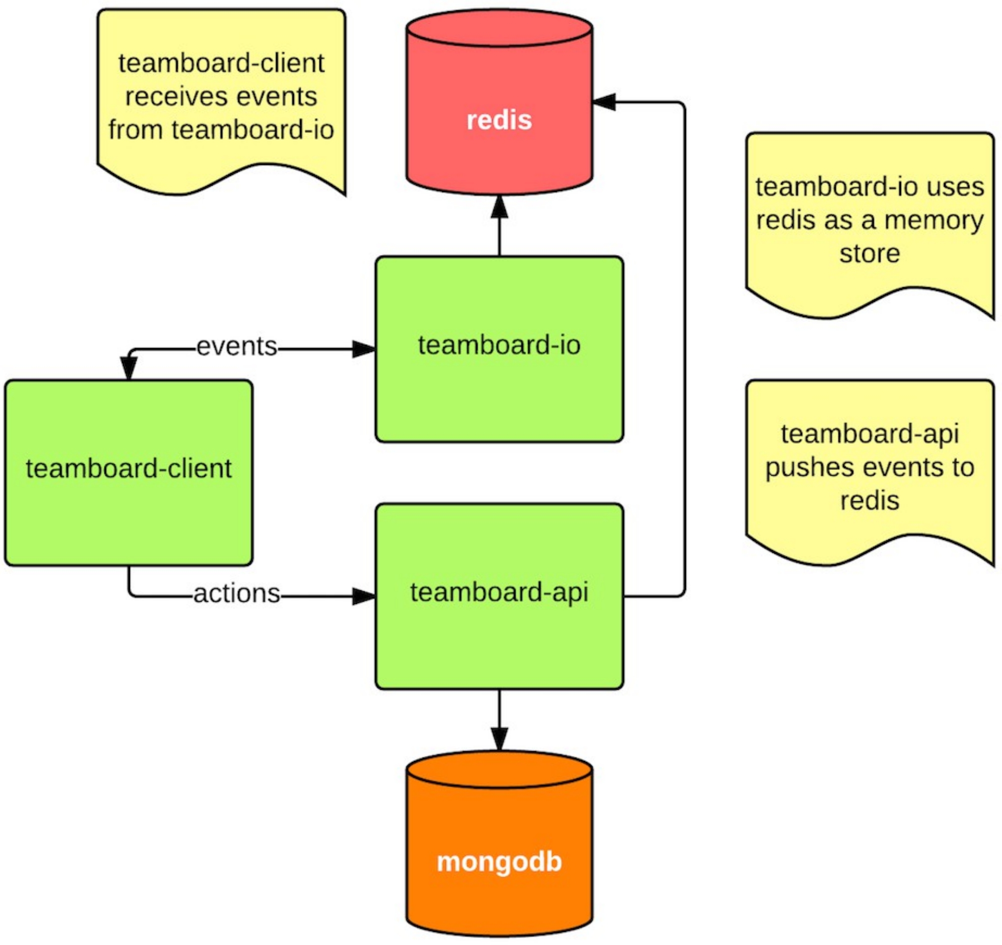

teamboard-client receives events from teamboard-io
redis
teamboard-io uses redis as a memory store
events
teamboard-io
teamboard-client
teamboard-api pushes events to redis
actions
teamboard-api
mongodb

Figure 3: Contriboard architecture

(Contriboard 2014)

Client: Used for user interactions. Application is made with AngularJS and runs on open-source Nginx server. Client connects to API and Socket.IO through HAProxy, which handles the load balancing accordingly.

API: Application Programming Interface (API) handles the authentication and resources. API is built on ExpressJS 4 framework service with Mongoose for MongoDB abstraction. Uses bcrypt to hash and compare passwords.

Socket.IO: Transmits events from API to client. Socket.IO takes an access-token in a query and compares it to the token generated for the user when logging in through

API.

Redis: Used by Socket.IO as a MemoryStore for scaling and API event handling.

MongoDB: NoSQL classified document-oriented database. Stores all user, board and ticket data and is queried by the API.

3 Continuous Delivery

Continuous Delivery (CDel) is a software development discipline. Main idea is to build the software so that it can be released at any given time. This can be achieved by continuously integrating the developed software, building executables from it and running automated tests against it to find issues. Afterwards the executables should be deployed in a production-like environment to make sure that it works before shipping it to a customer. (Fowler 2013.)

3.1 Continuous Integration

In Continuous Integration (CI) developers integrate their code frequently. Usually this happens at least once a day which leads to multiple integrations daily. Integration is immediately verified by automated build and tests to detect issues and complications. (Fowler 2013.)

Benefits of the Continuous Integration is to rapidly find bugs and issues and to make it easier to fix them in early stages. It also removes the so called "blind spot" where there are different branches of code but no idea how easy and fast it is to integrate them for the final product. (Fowler 2013.)

Continuous Integration is very much required to build Continuous Delivery or Deployment. (Fowler 2013).

3.2 Continuous Deployment

Continuous Delivery is sometimes mixed with Continuous Deployment (CDep). In

Continuous Deployment every change goes through the pipeline and is pushed to production automatically. Instead Continuous Delivery means that it is possible to do these frequent deployments, but it is not necessary and completely under the developer's control. (Fowler 2013.)

Continuous Delivery is required when implementing Continuous Deployment; however, not the other way around. (Fowler 2013.)

4 Introduction to software testing

Purpose of software testing is to add value to the product, which means raising the quality and reliability of the program in question. To do that, tester should not try to prove that the program works, but instead assume that it is filled with errors and try to find as many of them as possible. (Myers, Sandler & Badgett 2012, Chapter 2.)

While programmers value a working code as a successful accomplishment, testers could think it upside down and consider test case that finds an issue (more critical the better) as a success. It is virtually impossible to achieve in making a completely error-free program, so testing should not be considered as process to proof that the program works. (Myers et al. 2012, Chapter 2.)

4.1 ISTQB

International Software Testing Qualifications Board (ISTQB) is a non-profit association founded in Belgium in November 2002. Organization is based on volunteer work of hundreds of testing experts around the world. (International Software Testing Qualifications Board 2014.)

ISTQB issues multiple levels of certifications for software testing competences (see Figure 4 for the levels). In a nutshell these levels are (International Software Testing Qualifications Board 2014.):

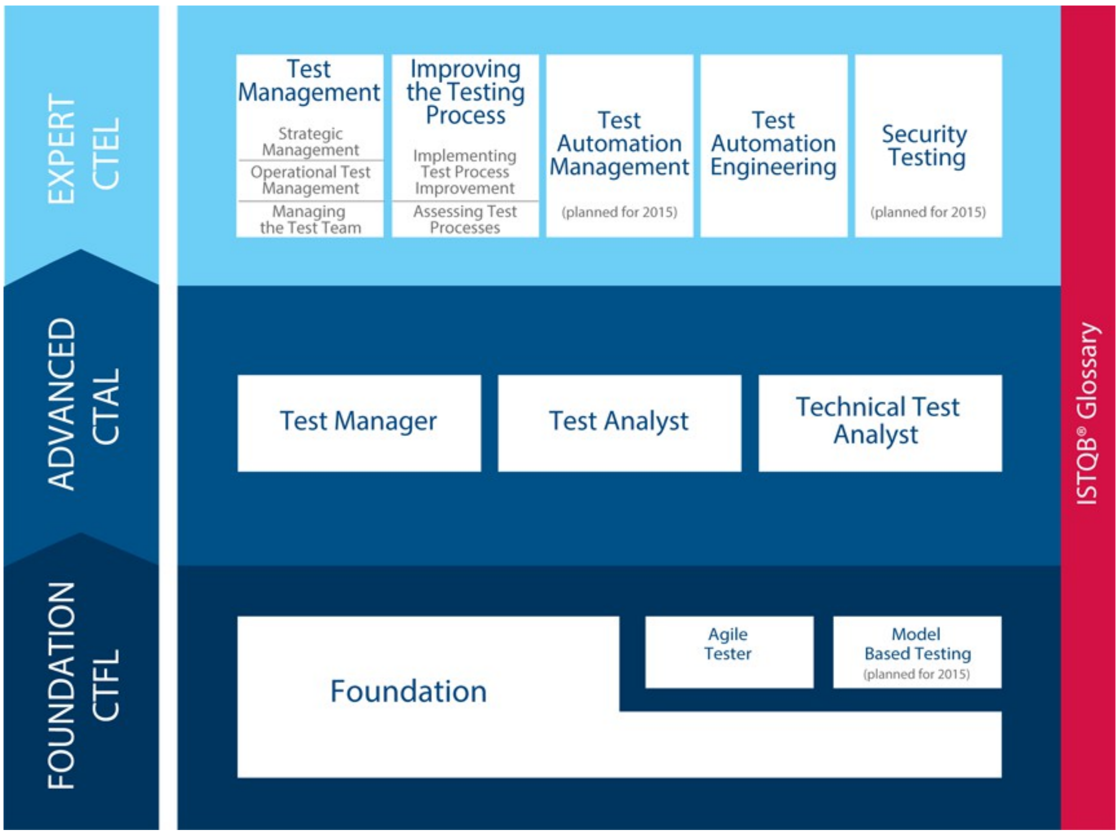


Figure 4: ISTQB exam levels

(International Software Testing Qualifications Board 2014)

Foundation: The foundation level qualification is for professionals who want to demonstrate their practical knowledge of the basic concepts of software testing. Or anyone who needs the basic understanding of the process.

Advanced: The advanced level qualification is for professionals who have achieved an advanced point in their career. Or anyone who needs deeper insight of software testing in general. Candidates who want the certificate must have completed the foundation level qualifications and have sufficient practical experience in the field.

Expert: The expert level qualification is for professionals that need in-depth and practically-oriented knowledge of wide range of different testing subjects. Candidates must have completed the advanced level qualification, have at least five years of practical testing experience and at least two years of industry experience in the specific expert level topic.

4.2 Box testing methods

Software testing is usually done with one of the box approach methods. They basically describe the point of view when designing the test cases. These methods are usually divided between black and white box testing, however, also sometimes as gray box testing.

Black Box testing: Testing strategy using only requirements and specifications. Tester does not need to know anything about the structure or implementations of the program. (Koirala & Sheikh 2008, 2.)

White Box testing: Testing strategy using code structure and implementation of the program. Tester should have sufficient programming skills for creating the tests. (Koirala & Sheikh 2008, 2.)

Gray Box testing: Testing strategy where tester looks inside the code just enough to understand how features work and then implement the tests using black box methods. (Koirala & Sheikh 2008, 2).

4.3 Testing process

Testing is a multiphase process which includes planning the test, designing the proper test cases, implementing and executing them, evaluating and reporting the outcome and finally taking the closing actions. (International Software Testing Qualifications Board 2014).

4.3.1 Planning the test

Planning a test has multiple major tasks that should be taken in consideration through the process (International Software Testing Qualifications Board 2014.):

Scope and risks: Determining which parts of the software are tested and what kind of risks they possibly include if issues are found. Objectives need to be identified for the test in question.

Test approach: Determining which testing methods need to be used. Which one is needed, black box or white box approach? Are the tests for continuously running regressive use or for example verifying requirement specification through acceptance testing?

Test policy: Test policy or test strategy basically is an outline describing the portion of software that is affected by the testing process. It gives information to testers and developers alike about some key issues that the testing process in the software developing cycle.

Resources: Determining the needed resources. How many people are needed to create the test cases and possibly executing them? What kind of environment is needed? Are physical machines required or can the tests be done with virtual machines?

Schedule: Determining the schedule for designing tasks, implementation, execution and evaluation of test in question.

Exit criteria: Determining criteria to see when the test is successfully completed. For example a coverage criterion defines the percentage of statements in the software that need to be executed during the testing process.

4.3.2 Designing the test

Designing a test have multiple major tasks that should be taken in consideration through the process (International Software Testing Qualifications Board 2014.):

Review test basis: Test basis is the needed information for creating the test cases.

Usually it is a documentation including requirements, design specifications, risk analysis and such. Basically a document offering understanding how the system should work after building it.

Identify test conditions: Defining what should happen in the test when some specific is done. For example username in sign-up form should always start with an alphabet and ignore numbers and special characters.

Design tests: Designing test cases for features and defining how they need to be executed step-by-step from the beginning to the expected outcome. Actual execution might be done manually or scripted for automated testing.

Evaluate testability: Testability of requirements and system under test (SUT) need to be evaluated to make a more realistic picture of whole process for creating the test cases.

Design test environment: Determining an environment and needed tools and infrastructure to create and execute the test cases.

4.3.3 Implementing the test

Implementation phase is the place to actually start working on test cases and possible automation scripts. Major tasks for implementation are (International Software Testing Qualifications Board 2014.):

Develop test cases: Creating and prioritizing test cases. Creating test data is also usually needed for testing certain features, like those that need some kind of input from the user, for example login.

Write test procedures: Test procedures are basically instructions how the tests should be executed.

Create test suites: Test suites are collections of test cases. They are used to test a specified set of behaviours in the software. Test suite includes information and instruction for the set of test cases. Basically test suites are used to group up same kind of test cases.

Implement environment: Building up the environment for the test suites and test cases.

4.3.4 Executing the test

When test cases are implemented, they naturally need to be executed. Major tasks for excution are (International Software Testing Qualifications Board 2014.):

Execute tests: Executing of individual test cases and test suites. Tests are carried out the way they are instructed in the test procedures.

Re-execute tests: Also known as re-testing or confirmation testing. Running the failed tests again to confirm a possible fix to the issue.

Write logs: Executed tests need to be logged so tester can see what happened during the run. Test log basically includes which test cases were executed and in what order, who executed them and did it pass or fail.

Compare results: Compare results from logs to those that were expected. If there was differences between them, report discrepancies.

4.3.5 Evaluating exit criteria

This is where exit criteria is set to determine where the limits are for when enough testing has been done. These are usually set based on the project's risk assessment plan or similar documentation. (International Software Testing Qualifications Board 2014.)

Common exit criteria are (International Software Testing Qualifications Board 2014.):

- Maximum amount of test cases has been executed and preset pass percentage is fulfilled.
- Amount of bugs is below preset treshold.
- Preset deadlines are met.

Major tasks for evaluating the exit criteria are (International Software Testing Qualifications Board 2014.):

Compare to test logs: Exit criteria is compared to test log and evaluated if all of them are met.

Additional tests: If criterion is not met, there need to be evaluation if additional tests are needed. Or maybe the exit criterion should be changed. This depends on the state of the software and recent changes to it.

Write summary: After evaluation, it is important to write a test summary report. It will be a valuable asset to developers to see how features and systems are currently working and it also may be required by the stakeholders involved in the project.

4.3.6 Closure activities

These activities are usually done at the moment of software delivery to the end user, but there are also few other cases when they could be needed. These could be for example (International Software Testing Qualifications Board 2014.):

- All the needed information has been gathered from the executed tests.
- Project has been suddenly canceled.
- Some major milestone has been achieved.
- An update has just been released.

Major tasks for the closure activities are (International Software Testing Qualifications Board 2014.):

Check status: Verify that everything that was planned has been delivered and discrepancies have been solved.

Archive test ware: All the used scripts, environments, testing software and other testing related stuff should be archived for later use.

Handover test ware: The aforementioned test ware should be delivered to the maintenance crew that will take over the future testing and maintaining of the software. Usually this a division inside the customer's organization.

Evaluate process: Finally the whole testing process should be evaluated. What went right, what went wrong and what could be learned from the journey for the next time.

4.4 Testing levels

Testing levels are used to make sure that every phase in development life cycle is tested and that they do not overlap with each other. Software development life cycle consists of phases like requirement gathering, design, coding and implementation. Each phase must be tested with various levels (ISTQB Exam Certification 2014.):

Unit testing: Unit testing is done by the developers to verify that their code is working as intended and corresponds the user specifications. They basically test their classes, functions and interfaces written in the code.

Component testing: Also known as module testing. Component testing differs from unit testing in manner of testing the whole component instead of just a piece of it is code.

Integration testing: Integration testing is done when two modules are integrated, so it can be verified that they work together as intended. Component integration test is performed when two components are integrated together. System integration test is performed when whole systems are integrated to work in same environment together.

System testing: System testing is done when compatibility of software needs to be verified with certain system.

Acceptance testing: Acceptance testing is done to make sure that the specification requirements are filled.

Alpha testing: Alpha testing is done at the end of the development cycle, usually at the developer's site or in other kind of closed environment.

Beta testing: Beta testing is done just before the launch at the end user's site or local environment.

4.5 Functional and non-functional testing

Testing can be categorized in functional and non-functional testing. (Software Testing Class 2012).

4.5.1 Functional testing

Functional tests derivate from business requirements. They evaluate if the software is functioning like specified or not, or if it is doing something completely unintended. Functional test consists of testing an action or function in code and verifying the outcome. (Software Testing Class 2012.)

Regression testing as an example

Re-testing process of the software that has gone through some modifications. It basically must be fully automated and aims to find flaws between components every time that any of them has changed even slightly. It would be counterproductive to run every test after each change, so usually regression tests are run for example at night or outside of office hours, and reviewed next morning. Alternatively regression tests could include only essential test cases to make the process faster, but then they would not cover all aspects of the software. (Ammann & Offutt 2008, 215.)

Regression tests need constant maintenance to cover all the new changes in the software. If test fails, it needs to be reviewed if the change is faulty or is the test case outdated. (Ammann & Offutt 2008, 215.)

Other examples of functional testing:

- Integration testing
- Interface testing
- System testing
- Unit testing
- Acceptance testing

4.5.2 Non-functional testing

Non-functional tests derivate from non-functional requirements, which basically specify the quality of the product. They evaluate aspects of the software that are not directly functional or user interactable. For example, how long does it take to respond to user actions under heavy load or how secure the database is. (Software Testing Class 2012.)

Performance testing as an example

Performance in software generally means that the applications performs well and responds to user interactions and given tasks without considerable delay or irritation to the user. (Molyneayx 2009, 2).

Measurement of performance happens with service-oriented and efficiency-oriented indicators. Service-oriented indicators are availability and response time, which measure how well the software provides services to user. Efficiency-oriented indicators are throughput and utilization, which measure how well the software makes use of the provided resources. Briefly defined these terms are (Molyneayx 2009, 3.):

Availability: The amount of time the software is available for the user. Meaning the time user is able to access and make use of the services. Lack of availability may be crucial to the end user in the terms of business cost.

Response time: The amount of time the software takes to respond to a user action. Usually response time is measured in system response time. System response time is the time between request action and response message measured client-side.

Throughput: The rate of occurring software-oriented events. For example, number of hits on a web page within a predefined period of time.

Utilization: The percentage of the used theoretical capacity of resource. For example, how much network bandwidth is being used while heavy traffic or how much memory the software uses server-side while high amount of users are active concurrently.

Other examples of non-functional testing:

- Compatibility testing
- Recovery testing
- Scalability testing
- Security testing
- Stress testing

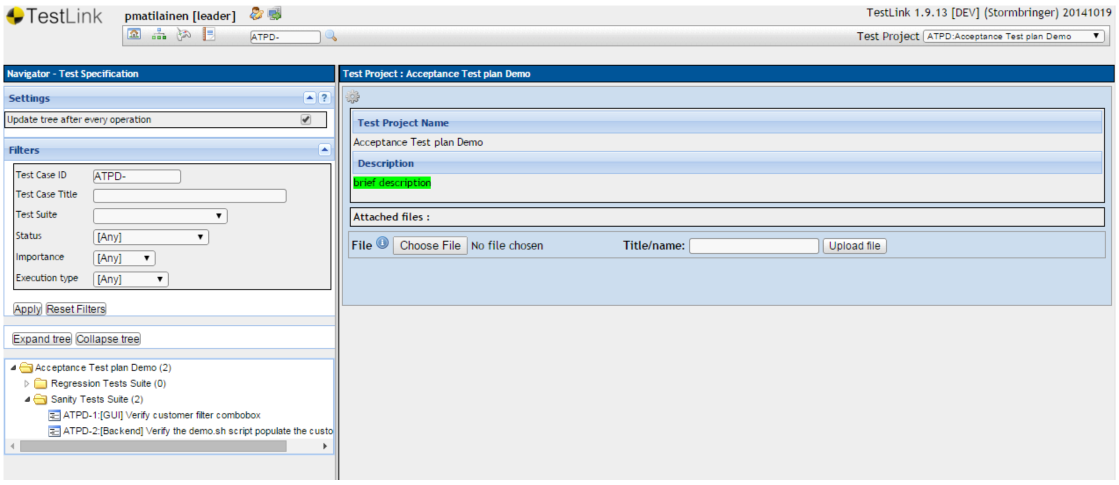
4.6 Manual testing

Method requires user to run tests manually without using any automated tools or predefined scripts. Tester behaves as an end user and tests the software while trying to find any unexpected behavior or other issues. (Software Testing Types 2014.)

In manual testing the person working as a tester is responsible for the correct execution of the tests. Tester usually uses a predefined test plan with different test cases or scenarios to systematically test the software. Another way is to go for exploratory testing route where tester explores the software without any specific plan identifying errors as they appear. (Software Testing Types 2014.)

Test plans are created with test management tool. One commonly used is open source project TestLink which provides ways to create test specifications, test plans and requirements specifications, execution of the plans, reporting support and collaboration with various bug trackers. (Havlat 2005.)

Test specification view of the TestLink can be seen in the Figure 5 below.



TestLink 1.9.13 [DEV] (Stormbringer) 20141019

pmatilainen [leader]

Test Project ATPD:Acceptance Test plan Demo

Update tree after every operation ✓

Test Case ID: ATPD-
Test Case Title:
Test Suite:
Status: [Any]
Importance: [Any]
Execution type: [Any]

Acceptance Test plan Demo (2)
Regression Tests Suite (0)
Sanity Tests Suite (2)
ATPD-1:[GUI] Verify customer filter combobox
ATPD-2:[Backend] Verify the demo.sh script populate the custo

Test Project : Acceptance Test plan Demo

Test Project Name: Acceptance Test plan Demo

Description: brief description

No file chosen   Title/name:

Figure 5: TestLink test specification view

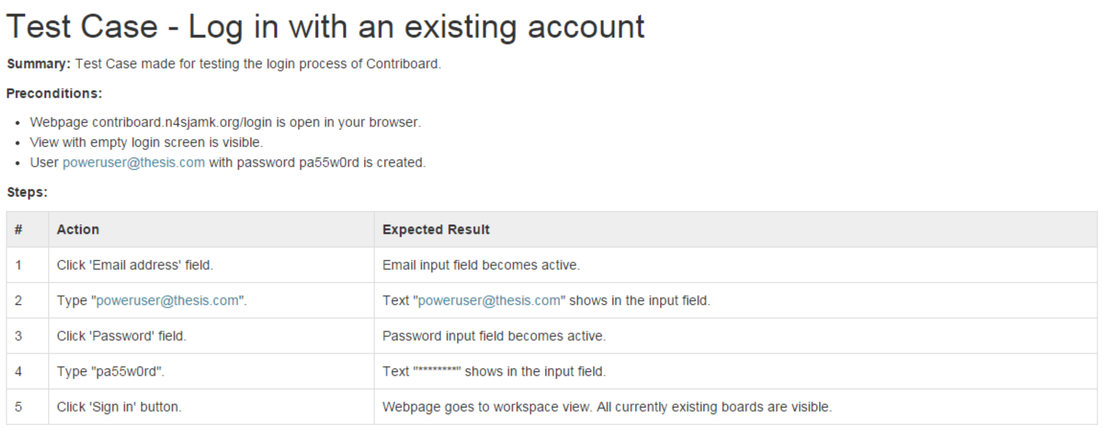
Most of the testing should be automated but there are still few examples where manual testing is recommended (Koirala & Sheikh 2008, 127.):

Unstable software: Software that is under heavy development and receives constant big changes. Automation testing scripts fall behind and it may be more efficient to do the testing manually.

"Once in a blue moon" test scripts: Testing scripts that are run once in a while should be executed manually because the software may have changed drastically from the last run testing script.

Code and document review: Automating code and document review is ill-advised and probably cause more harm than good.

Example test case for manual testing can be seen in Figure 6 below. It was used in early states of Contriboard testing at the summer as a part of Summer Challenge Factory 2014.



Test Case - Log in with an existing account

Summary: Test Case made for testing the login process of Contriboard.

Preconditions:

- Webpage contriboard.n4sjamk.org/login is open in your browser.
- View with empty login screen is visible.
- User poweruser@thesis.com with password pa55w0rd is created.

Steps:

#	Action	Expected Result
1	Click 'Email address' field.	Email input field becomes active.
2	Type "poweruser@thesis.com".	Text "poweruser@thesis.com" shows in the input field.
3	Click 'Password' field.	Password input field becomes active.
4	Type "pa55w0rd".	Text "********" shows in the input field.
5	Click 'Sign in' button.	Webpage goes to workspace view. All currently existing boards are visible.

Figure 6: Example test case for manual testing

4.7 Test automation

In this method tester writes scripts and uses preferred software to handle the testing process. It is not possible to automate everything, but most of the scenarios can and should be automated. (Software Testing Types 2014.)

Few examples for the use of test automation are (Software Testing Types 2014.):

Regression testing: Re-run test scenario quickly and repeatedly. Saves considerable amount of time and resources compared to manual testing.

Performance stress testing: Run test scenario with multiple concurrent actions. For example log in with 1000 users simultaneously to see if the software can handle the stress.

Validations and functionality: Test cases that for example test database connections, input validations and GUI element functionality.

Examples of automation test scripts can be examined in the chapter 6 later below.

5 Acceptance testing

Main objective of the acceptance testing is to confirm that the software meets the given business requirements. It also verifies that the software is working as intended before delivering it to the end user. (Watkins & Mills 2011.)

5.1 User Acceptance Testing

User acceptance testing (UAT) focuses more on the meeting the business requirements side of the acceptance testing. It is usually done with black box testing method where tester does not have any insight what happens under the hood of the software. (Watkins & Mills 2011.)

User acceptance tests should be performed by the user representatives with the help of the testing team and supervised by a testing team leader. Leader should make sure that appropriate amount of testing is done with right kind of testing technics. (Watkins & Mills 2011.)

User acceptance tests are planned beforehand by the testing team leader. They should be developed with reference to the overall development and testing plan for the software so that the developing of the product will not get interfered by the tests. Test contents should reflect the normal user's everyday use of the software. (Watkins & Mills 2011.)

5.2 Operational Acceptance Testing

Operational acceptance testing (OAT) is very much similar to user acceptance testing except it focuses more on meeting the operations requirements side of the acceptance testing. It is also done with the black box approach but instead of normal users, it should be performed by the operative’s representatives. There should also be testing team and their leader present to give support and supervise the testing process. (Watkins & Mills 2011.)

Operational acceptance tests are planned beforehand by the testing team leader. They should also be developed with reference to the overall development and testing plan for the software to minimize possible conflicts in the product development. Test content should include operations and administrative aspects like updating, backing up and restoring software systems and used data, registering new users and maintaining their privileges. (Watkins & Mills 2011.)

6 Examples of acceptance testing tools

Next are few examples of different acceptance testing tools. Couple were picked by popularity and few others were used earlier in Contriboard testing itself alongside the Ixonos Visual Test tool.

Note that these tools are only described very briefly since the main focus was on using the Ixonos Visual Test in actual implementation of the work.

6.1 Cucumber

Cucumber has not been used in Contriboard testing but here is a very brief description of the project.

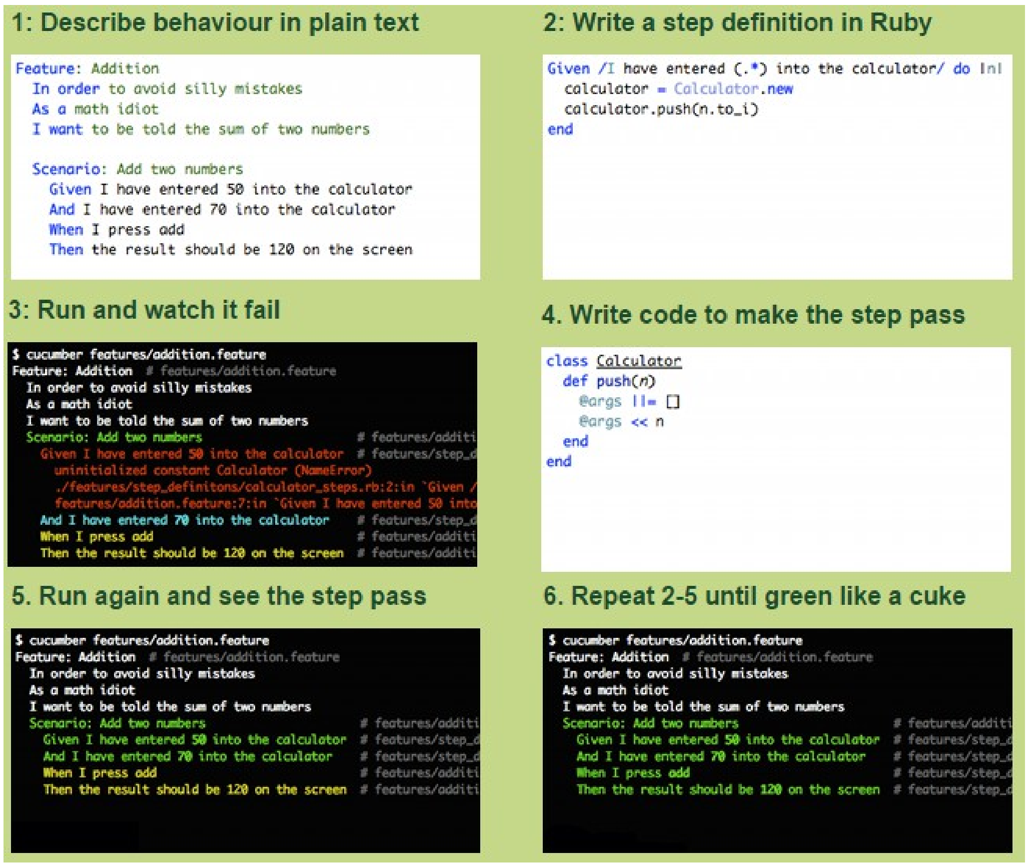
Cucumber is a Behavior Driven Development (BDD) tool, which means that the behavior is defined before the actual code implementation. Basically Cucumber lets developers describe how the software they are working on should work in plain text. The same text can then be used as a documentation for implementation and testing. (Cucumber 2014.)

Cucumber works with multiple languages, for example Ruby, Java, .NET, or any commonly used web application languages. (Cucumber 2014).

Example workflow for Cucumber is as follows (Cucumber 2014.):

1. Describe behaviour in plain text
2. Write test definition with selected programming language
3. Write code to actually make the step pass
4. Run the test

Same flow can be seen in the Figure 7 below.



1: Describe behaviour in plain text

```
Feature: Addition
  In order to avoid silly mistakes
  As a math idiot
  I want to be told the sum of two numbers

  Scenario: Add two numbers
    Given I have entered 50 into the calculator
    And I have entered 70 into the calculator
    When I press add
    Then the result should be 120 on the screen
```

2: Write a step definition in Ruby

```
Given /I have entered (.*) into the calculator/ do |n|
  calculator = Calculator.new
  calculator.push(n.to_i)
end
```

3: Run and watch it fail

```
$ cucumber features/addition.feature
Feature: Addition  # features/addition.feature
  In order to avoid silly mistakes
  As a math idiot
  I want to be told the sum of two numbers
  Scenario: Add two numbers                      # features/additi
    Given I have entered 50 into the calculator  # features/step_d
      uninitialized constant Calculator (NameError)
      ./features/step_definitons/calculator_steps.rb:2:in `Given /
      features/addition.feature:7:in `Given I have entered 50 into
    And I have entered 70 into the calculator    # features/step_d
    When I press add                             # features/additi
    Then the result should be 120 on the screen  # features/additi
```

4. Write code to make the step pass

```
class Calculator
  def push(n)
    @args ||= []
    @args << n
  end
end
```

5. Run again and see the step pass

```
$ cucumber features/addition.feature
Feature: Addition  # features/addition.feature
  In order to avoid silly mistakes
  As a math idiot
  I want to be told the sum of two numbers
  Scenario: Add two numbers                      # features/additi
    Given I have entered 50 into the calculator  # features/step_d
    And I have entered 70 into the calculator    # features/step_d
    When I press add                             # features/additi
    Then the result should be 120 on the screen  # features/additi
```

6. Repeat 2-5 until green like a cuke

```
$ cucumber features/addition.feature
Feature: Addition  # features/addition.feature
  In order to avoid silly mistakes
  As a math idiot
  I want to be told the sum of two numbers
  Scenario: Add two numbers                      # features/additi
    Given I have entered 50 into the calculator  # features/step_d
    And I have entered 70 into the calculator    # features/step_d
    When I press add                             # features/step_d
    Then the result should be 120 on the screen  # features/step_d
```

Figure 7: Cucumber workflow

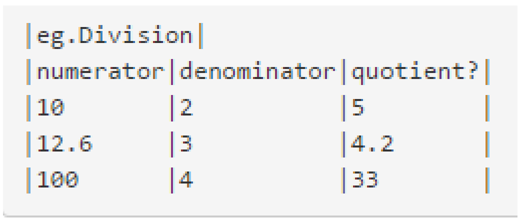
(Cucumber 2014)

6.2 FitNesse

Like Cucumber, FitNesse has not been used either in Contriboard testing but here is a very brief description of the project.

FitNesse is a software development tool for customers, testers and programmers to collaborate efficiently. It basically tells what the software should do and what it actually currently does. (FitNesse 2014.)

That being said, FitNesse also is a lightweight software testing tool for acceptance testing purposes. Tests are written in wiki markup language in the format of tables as seen in a Figure 8 below. (FitNesse 2014.)



```
|eg.Division|
|numerator|denominator|quotient?|
|10       |2          |5        |
|12.6     |3          |4.2      |
|100      |4          |33       |
```

Figure 8: FitNesse example.

(FitNesse 2014)

As seen in the example, there is a number 10 which will be divided with number 2 and the expected outcome should be number 5.

6.3 Selenium

Selenium alone has not been used in Contriboard testing, but it is used in fMBT (see chapter 6.5) and Ixonos Visual Test software. Here is a very brief description of the project.

Selenium is a set of software tools used in supporting test automation process. For example location UI elements, interacting with them and comparing expected results. (Selenium Project 2014.)

These days the Selenium is called Selenium 2 or Selenium WebDriver, because of the integration of WebDriver project to the original Selenium 1. It enables more varied testing methods that were not possible with original implementation because of JavaScript based engine. Selenium 2 basically supports the WebDriver API and technology used in it for better flexibility for porting tests. (Selenium Project 2014.) Selenium IDE is a prototyping tool for creating and running test scripts in browser. It is basically a plugin for Firefox. It can record user action in the browser and turn them in reusable scripts for automated tests in multiple different programming languages. (Selenium Project 2014.)

Selenium WebDriver has a support for many different broswers listed below (Selenium Project 2014.):

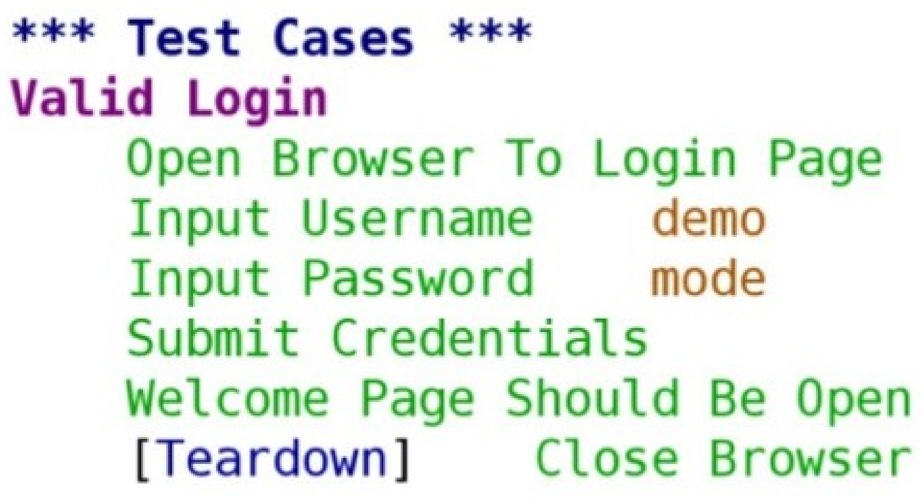
- Google Chrome
- Internet Explorer 6-10
- Mozilla Firefox
- Safari
- Opera
- HtmlUnit
- phantomjs

It also supports Android with Selendroid extension and iOS with ios-driver extension. Also both are supported with Appium extension alone. (Selenium Project 2014.)

6.4 Robot Framework

Robot Framework has been somewhat used in testing Contriboard. It is also a subject for another co-worker's thesis work but here is a very brief description of the Robot Framework.

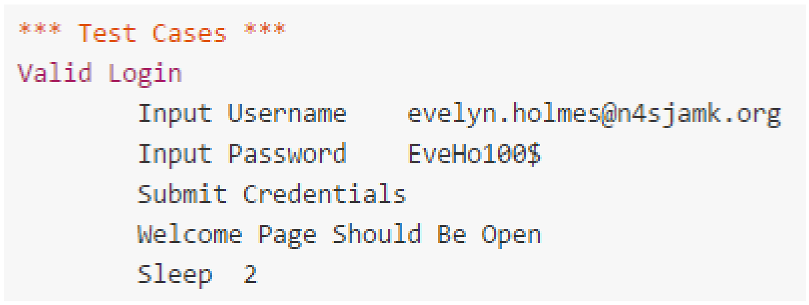
Robot Framework is a test automation framework that utilizes keyword-driven testing methods (see Figure 9 for syntax example). It is primarily made for acceptance level testing but can also be used in other automation testing. Its testing capabilities can be easily extended with libraries made by using Python or Java. (Klärck 2014.)



```
*** Test Cases ***
Valid Login
    Open Browser To Login Page
    Input Username    demo
    Input Password    mode
    Submit Credentials
    Welcome Page Should Be Open
    [Teardown]    Close Browser
```

Figure 9: Robot Framework keyword-driven syntax example

Here in Figure 10 below is a an example script for successful login attempt in Contriboard.



```
*** Test Cases ***
Valid Login
        Input Username    evelyn.holmes@n4sjamk.org
        Input Password    EveHo100$
        Submit Credentials
        Welcome Page Should Be Open
        Sleep   2
```

Figure 10: Robot Framework valid login script in Contriboard

6.5 fMBT

fMBT (free Model-Based Testing) tool was used earlier in Contriboard testing as a part of Summer Challenge Factory 2014. Here is a brief explanation of the fMBT project.

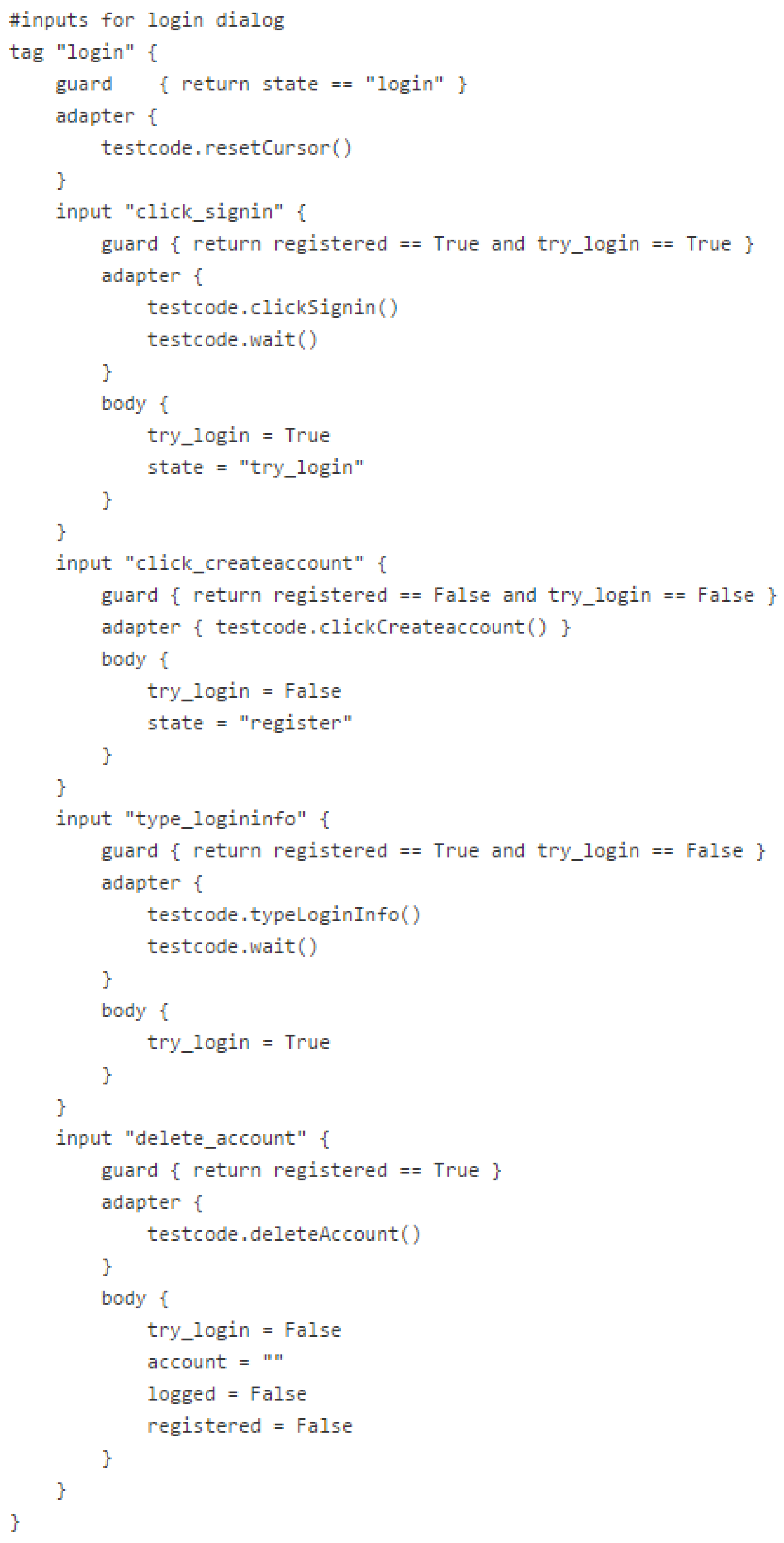
fMBT is for testing anything between individual classes to GUI applications in model-based fashion. fMBT consists of model editor, test generator, adapters for different interfaces and log analyzing tools. (Kervinen 2014.)

Test generator supports both online and offline model-based testing methods and scripts themself can be written in few different languages (Kervinen 2014.):

- Python
- C++
- JavaScript
- shell script

fMBT runs on Linux platform and it is easy to implement it to continuous delivery chain since both test generation and execution are run from the command line. (Kervinen 2014).

Example of login script in Contriboard model can be seen in Figure 11 below.

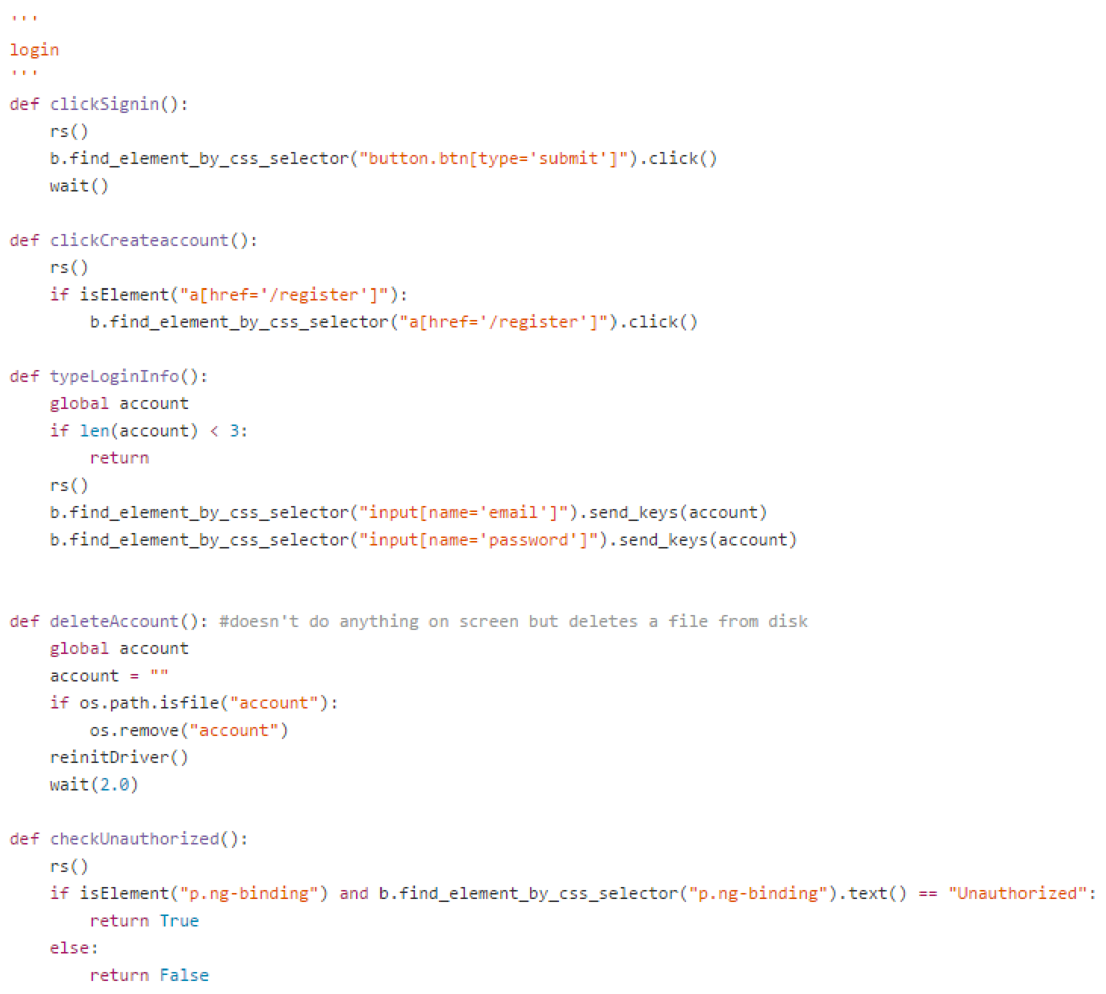


```
#inputs for login dialog
tag "login" {
    guard    { return state == "login" }
    adapter {
        testcode.resetCursor()
    }
    input "click_signin" {
        guard { return registered == True and try_login == True }
        adapter {
            testcode.clickSignin()
            testcode.wait()
        }
        body {
            try_login = True
            state = "try_login"
        }
    }
    input "click_createaccount" {
        guard { return registered == False and try_login == False }
        adapter { testcode.clickCreateaccount() }
        body {
            try_login = False
            state = "register"
        }
    }
    input "type_logininfo" {
        guard { return registered == True and try_login == False }
        adapter {
            testcode.typeLoginInfo()
            testcode.wait()
        }
        body {
            try_login = True
        }
    }
    input "delete_account" {
        guard { return registered == True }
        adapter {
            testcode.deleteAccount()
        }
        body {
            try_login = False
            account = ""
            logged = False
            registered = False
        }
    }
}
```

Figure 11: fMBT – Example of login model script

(GitHub teamboard-test -repository)

Example of login script in selenium adapter can be seen in the Figure 12 below.



```
'''
login
'''
def clickSignin():
    rs()
    b.find_element_by_css_selector("button.btn[type='submit']").click()
    wait()

def clickCreateaccount():
    rs()
    if isElement("a[href='/register']"):
        b.find_element_by_css_selector("a[href='/register']").click()

def typeLoginInfo():
    global account
    if len(account) < 3:
        return
    rs()
    b.find_element_by_css_selector("input[name='email']").send_keys(account)
    b.find_element_by_css_selector("input[name='password']").send_keys(account)


def deleteAccount(): #doesn't do anything on screen but deletes a file from disk
    global account
    account = ""
    if os.path.isfile("account"):
        os.remove("account")
    reinitDriver()
    wait(2.0)

def checkUnauthorized():
    rs()
    if isElement("p.ng-binding") and b.find_element_by_css_selector("p.ng-binding").text() == "Unauthorized":
        return True
    else:
        return False
```

Figure 12: fMBT – Example of login adapter script

(GitHub teamboard-test -repository)

7 Model-based testing

7.1 Definition

Model-based testing is practically automation of designing tests with black box approach. Which means that if tests are traditionally generated by basing them on the requirements documents, in model-based approach there is a model that is generated by basing it on the expectation how the SUT should behave. Test cases by themselves are then generated automatically from that model. (Utting & Legeard 2007, 8.)

7.2 Model

The model can be thought as an abstraction of the SUT. It defines the possible inputs and expected outcomes. Model should be as small as possible but also very detailed in the nature. (Utting & Legeard 2007, 9.)

The model should be focusing only to key aspects of the SUT and what needs to be tested to maintain the small size. It should also be completely error free, since these will become bigger issues later on if found. (Utting & Legeard 2007, 28.)

After the model is generally satisfactory, it is time to create a test suite with some free or commercial model-based testing tool (Utting & Legeard 2007, 9). The test suite is basically a collection of test cases discussed in the next chapter.

7.3 Test cases

First the abstract test cases are generated from the model. They are sequences of operations with values such as predefined inputs and expected outputs. (Utting & Legeard 2007, 9.)

In bigger models especially, it is important to define selection criteria for the test generation. Reason for this is that there usually are infinite amount of different routes in the model which the generator tries to turn into test cases. It is usually preferred to focus for example only on one part of the model for the generation process. (Utting & Legeard 2007, 28.)

Next these abstract test cases are concretized to executable scripts. This is usually also done with some model-based testing tool, but they can also be done manually with some scripting or programming language like Python or Java. (Utting & Legeard 2007, 9-10.)

Main advantage of dividing the tests into two different layers is that the abstract test cases are not tied to any specific programming language and can be later re-used in different environments by simply changing the adaptor code which is the link between test cases and actual SUT when executing them. (Utting & Legeard 2007, 29.)

These scripts can be then executed against the SUT and usually monitored with some testing tool to see what actually happens.

7.4 Testing process

Model-based testing process differs from normal testing process so, that instead of manually writing every test case, the test designer does the job. This reduces the overall design time drastically and every possible path in the model will be covered. (Utting & Legeard 2007, 26-27.)

Testing process is divided in five phases in Figure 13 below (Utting & Legeard 2007, 27.):

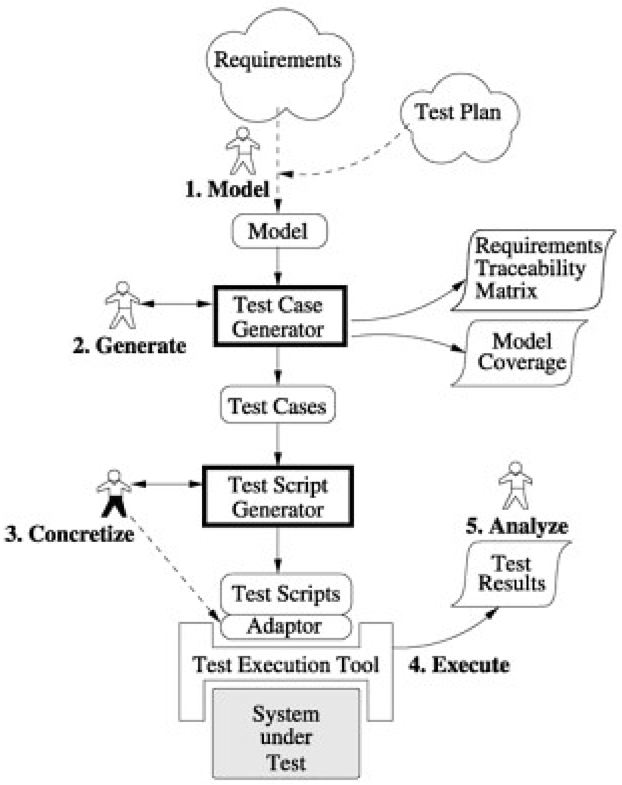


Figure 13: Generic model-based testing process (Utting & Legeard 2007, 27)

Model: Creation of model for the SUT and defining the restrictions for it.

Generate: Generation of abstract test cases from the model, which are automatically generated with a model-based testing tool.

Concretize: Turning the abstract test cases into executable scripts with the help of model-based testing tool or by manually writing them.

Execute: Executing the scripts against the SUT in either online or offline method with a model-based testing tool.

Analyze: Analyzing the outcome of the tests with the testing tool or possibly from the logs if no visualization is available.

7.5 Execution and analyzing of tests

There are two ways to execute the tests, online and offline. Analyzing model-based test run usually does not differ from analyzing more traditional test run.

Online execution

When using online method the tests are executed as they are produced. In the case the model-based testing tool executes the tests in order it sees fit and records the results. (Utting & Legeard 2007, 30.)

There are usually guards set to restrict some transactions in the model until certain criterion has been met. Also the paths taken are usually done with setting weights to them which simulates the probability of user taking them.

Offline execution

When using the offline method, there are preset concrete test case scripts generated and the testing tool executes them in predefined order and records the results. (Utting & Legeard 2007, 30). This method is encouraged to be used when there is need to execute the same tests regularly.

Analyzing test run

After execution phase, the results need to be analyzed. Model-based testing tool should be giving a detailed report for successful and failed test cases. Like stated earlier, analyzing the test report does not usually differ in model-based testing and traditional testing methods. (Utting & Legeard 2007, 30.)

Every failed test case need to be examined. It may be that the SUT has a flaw or that the test case is not up to date or contains an error. If it is the latter case, the adaptor code or model needs to be examined. It is common that first test runs contain high amount of failures because of small errors in the adaptor code. (Utting & Legeard 2007, 30.)

8 Ixonos Visual Test designer

Ixonos Visual Test (IVT) tool was used for implementing and running the test model in the thesis. Next is a brief description of the software and its features used for making the Contriboard model. Software has many other features that are not included here because of the limited use experience, for example online test runner used for performance testing.

Note that the software is currently developing rapidly and these features are described as they were at the moment of writing in the early December 2014.

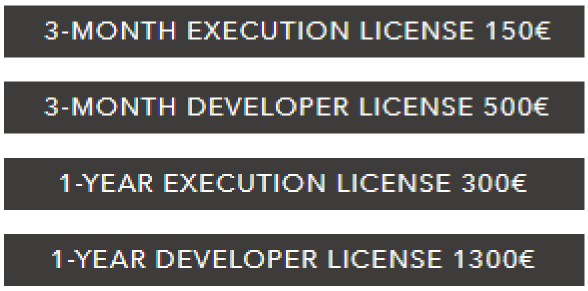
8.1 Introduction

Ixonos Visual Test is an automation framework for web and mobile software and it can be used on both Windows and Linux platforms. IVT is used for visually planning model-based tests for the target SUT. It uses earlier mentioned Selenium WebDriver for web page automation and Appium for Android testing. (Ixonos Visual Test 2014.)

Test execution can be integrated with Jenkins and TestLink and supported plaforms are (Ixonos Visual Test 2014.):

- Google Chrome
- Mozilla FireFox
- Internet Explorer
- Opera
- Android

Payment is subscription based and current prices for the different licenses can be seen in the Figure 14 below.



3-MONTH EXECUTION LICENSE 150€

3-MONTH DEVELOPER LICENSE 500€

1-YEAR EXECUTION LICENSE 300€

1-YEAR DEVELOPER LICENSE 1300€

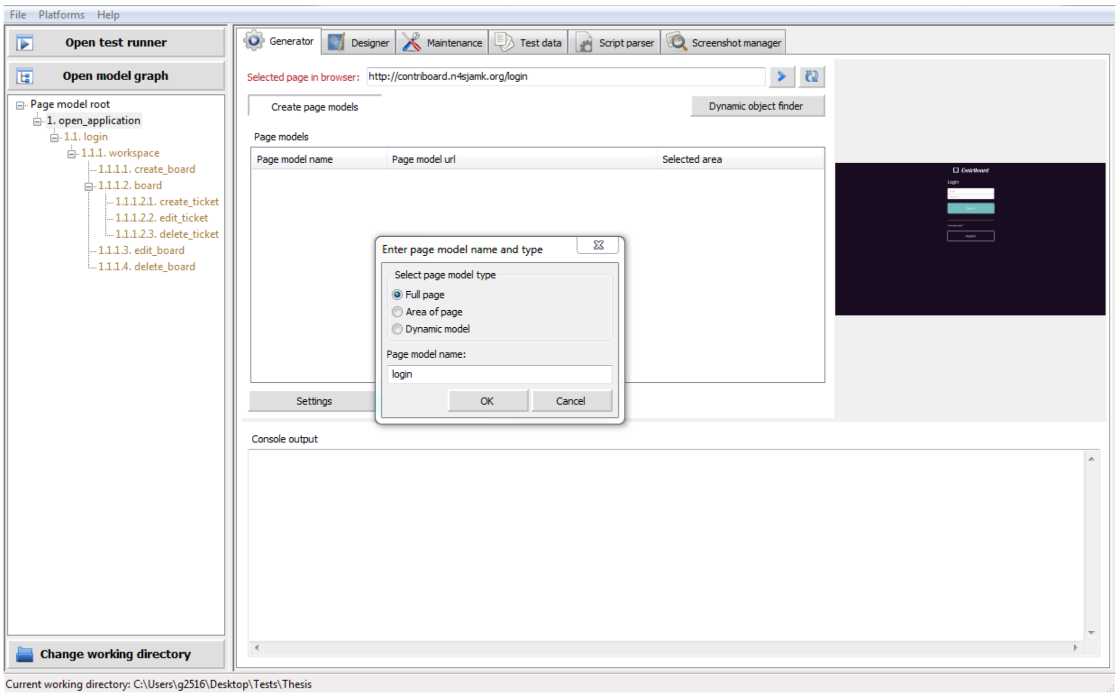
Figure 14: Ixonos Visual Test pricing

(Ixonos Visual Test 2014)

Alternatively there is a team license option with 1900€ price per 1-year development license to the license pool. (Ixonos Visual Test 2014).

8.2 Page models

IVT uses a concept of page models. Page model is created in the Generator-tool by opening the desired web page in the browser and clicking the create page models button. User can then decide if the page model is generated from the whole page, certain area of it or by manually selecting the wanted elements (see Figure 15 for an example).



Page model root
- 1. open_application
  - 1.1. login
    - 1.1.1. workspace
      - 1.1.1.1. create_board
      - 1.1.1.2. board
        - 1.1.1.2.1. create_ticket
        - 1.1.1.2.2. edit_ticket
        - 1.1.1.2.3. delete_ticket
      - 1.1.1.3. edit_board
      - 1.1.1.4. delete_board

Selected page in browser: http://contriboard.n4sjamk.org/login

Enter page model name and type

Select page model type
(● Full page) ○ Area of page ○ Dynamic model

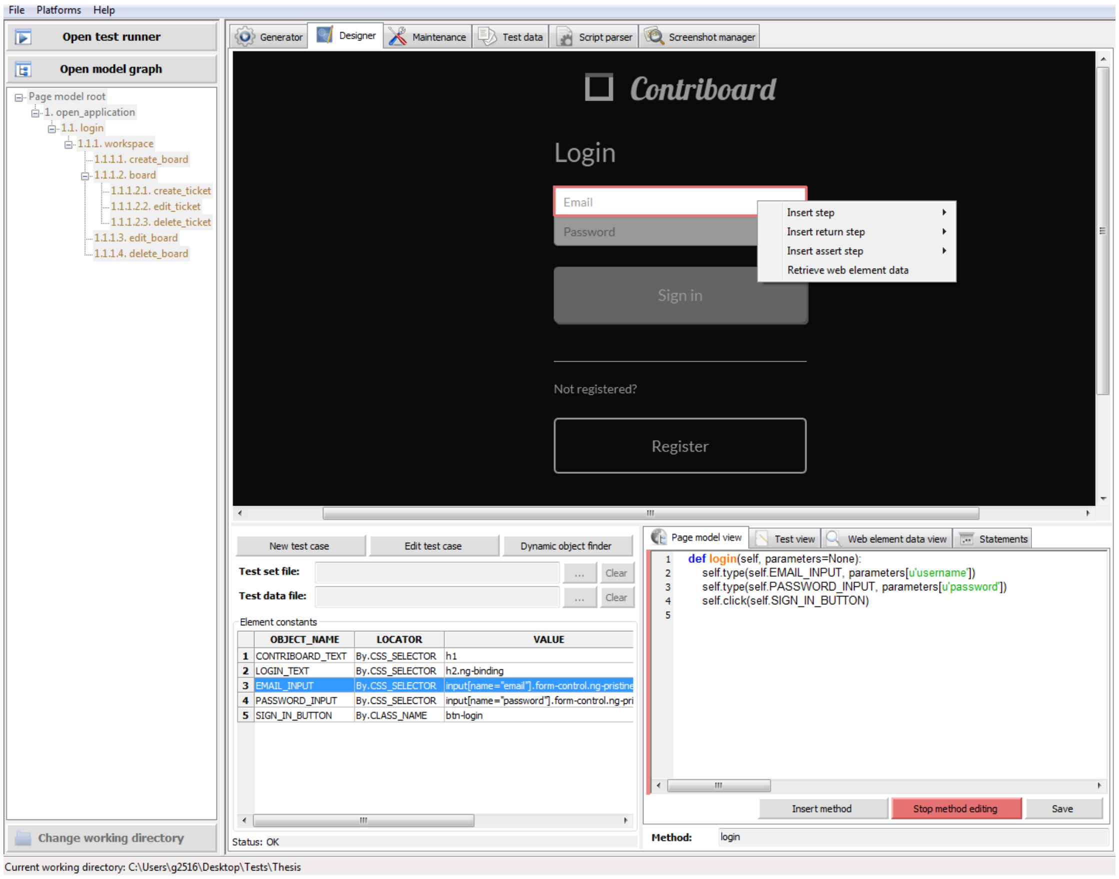
Page model name: login

Current working directory: C:\Users\g2516\Desktop\Tests\Thesis

Figure 15: IVT UI - Create page models

Page models contain elements of buttons, fields, images and other objects, which can later be used in interactions and verifications. Elements can be searched from the web page multiple ways. For example by id, class or xpath.

After page model is created, user can insert and edit methods in the Designer-tool by clicking the element and selecting the preferred action (see Figure 16 for an example). User can also write custom Python scripts to the methods.



Element constants

	OBJECT_NAME	LOCATOR	VALUE
1	CONTRIBOARD_TEXT	By.CSS_SELECTOR	h1
2	LOGIN_TEXT	By.CSS_SELECTOR	h2.ng-binding
3	EMAIL_INPUT	By.CSS_SELECTOR	input[name="email"].form-control.ng-pristine
4	PASSWORD_INPUT	By.CSS_SELECTOR	input[name="password"].form-control.ng-pri
5	SIGN_IN_BUTTON	By.CLASS_NAME	btn-login

Status: OK

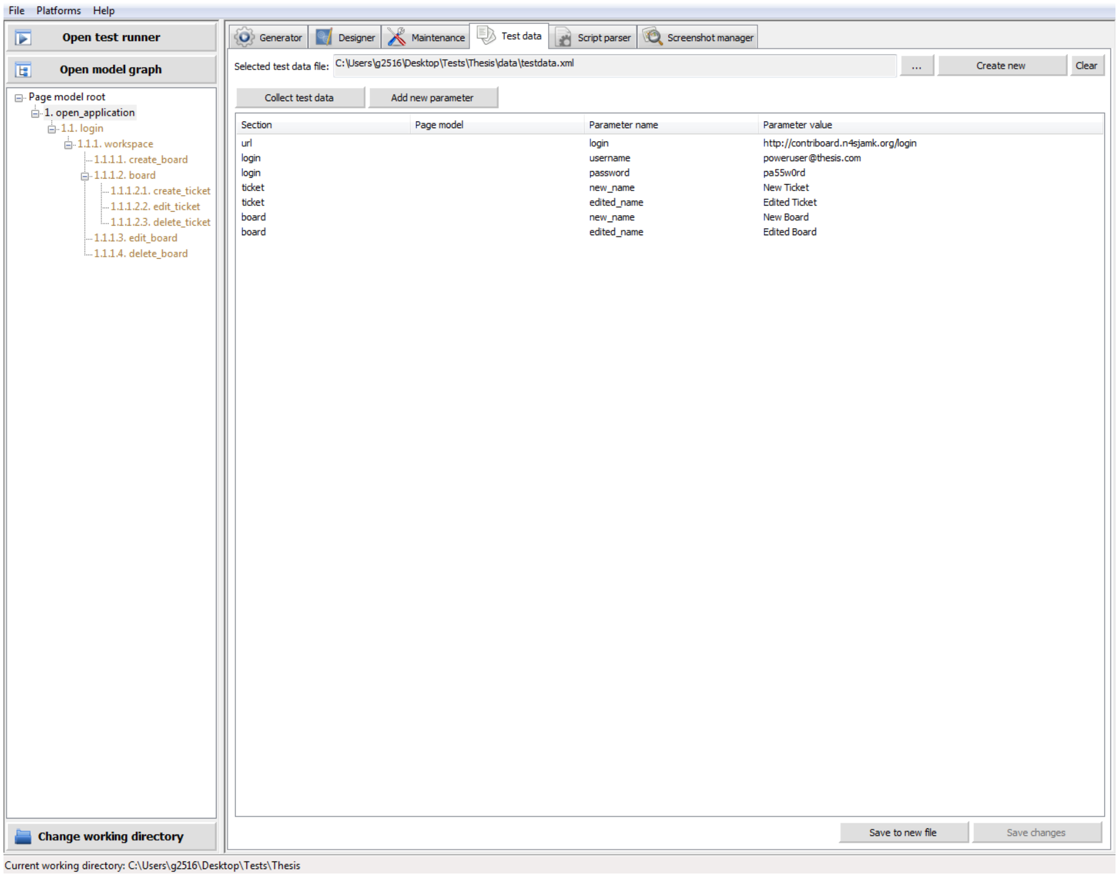
```
def login(self, parameters=None):
    self.type(self.EMAIL_INPUT, parameters[u'username'])
    self.type(self.PASSWORD_INPUT, parameters[u'password'])
    self.click(self.SIGN_IN_BUTTON)
```

Method: login

Current working directory: C:\Users\g2516\Desktop\Tests\Thesis

Figure 16: IVT UI - Edit method

The test data needed for running the tests can be stored to xml file in Test data -tool (see Figure 17 for an example). It can be used in Designer by selecting the file when creating methods.



Page model root
- 1. open_application
  - 1.1. login
    - 1.1.1. workspace
      - 1.1.1.1. create_board
      - 1.1.1.2. board
        - 1.1.1.2.1. create_ticket
        - 1.1.1.2.2. edit_ticket
        - 1.1.1.2.3. delete_ticket
      - 1.1.1.3. edit_board
      - 1.1.1.4. delete_board

Selected test data file: C:\Users\g2516\Desktop\Tests\Thesis\data\testdata.xml

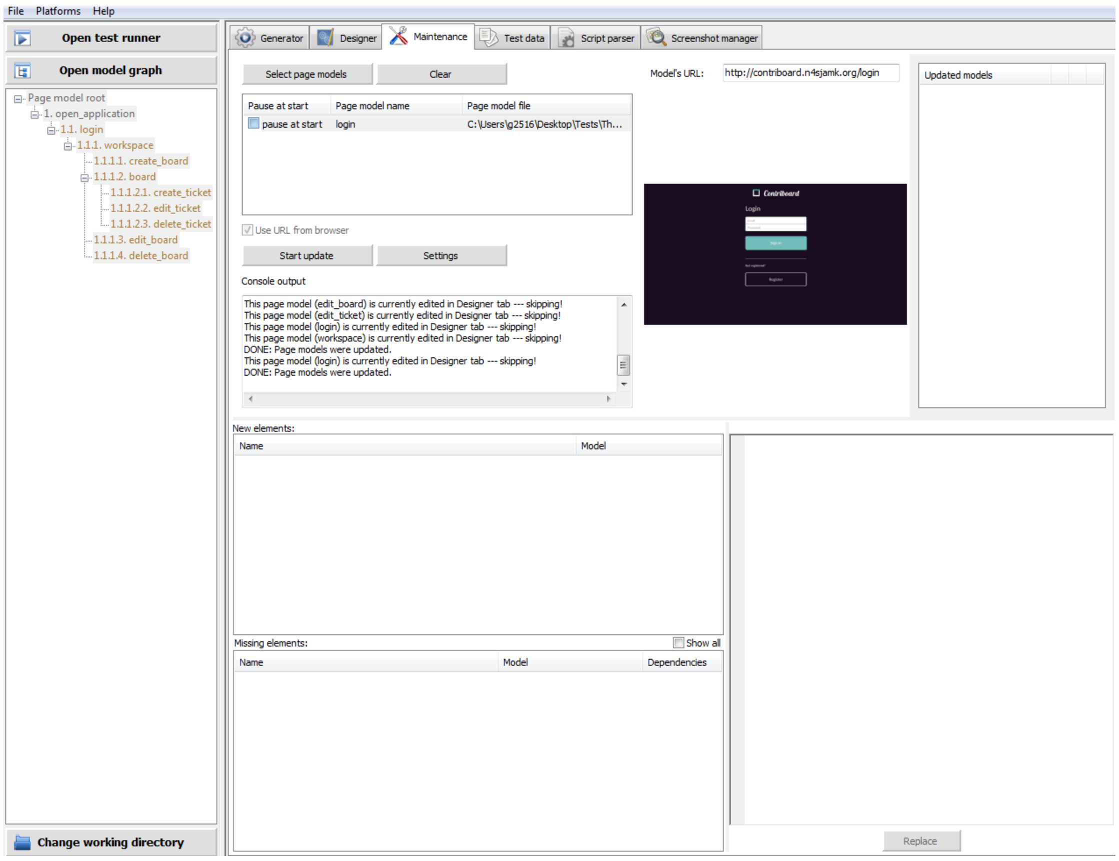
Section	Page model	Parameter name	Parameter value
url		login	http://contriboard.n4sjamk.org/login
login		username	poweruser@thesis.com
login		password	pa55w0rd
ticket		new_name	New Ticket
ticket		edited_name	Edited Ticket
board		new_name	New Board
board		edited_name	Edited Board

Current working directory: C:\Users\g2516\Desktop\Tests\Thesis

Figure 17: IVT UI - Test data tool

8.3 Maintenance

Maintenance-tool offers user an easy way of keeping the page models updated. If for example an element in login screen shown in Figure 16 changes, user can select the page model and start the update function (see Figure 18 for an example). It tries to find missing and new elements and suggest an automatic replace action for them.



Page model root
1. open_application
1.1. login
1.1.1. workspace
1.1.1.1. create_board
1.1.1.2. board
1.1.1.2.1. create_ticket
1.1.1.2.2. edit_ticket
1.1.1.2.3. delete_ticket
1.1.1.3. edit_board
1.1.1.4. delete_board

Model's URL: http://contriboard.n4sjamk.org/login

Pause at start	Page model name	Page model file
☐ pause at start	login	C:\Users\g2516\Desktop\Tests\Th...

☑ Use URL from browser

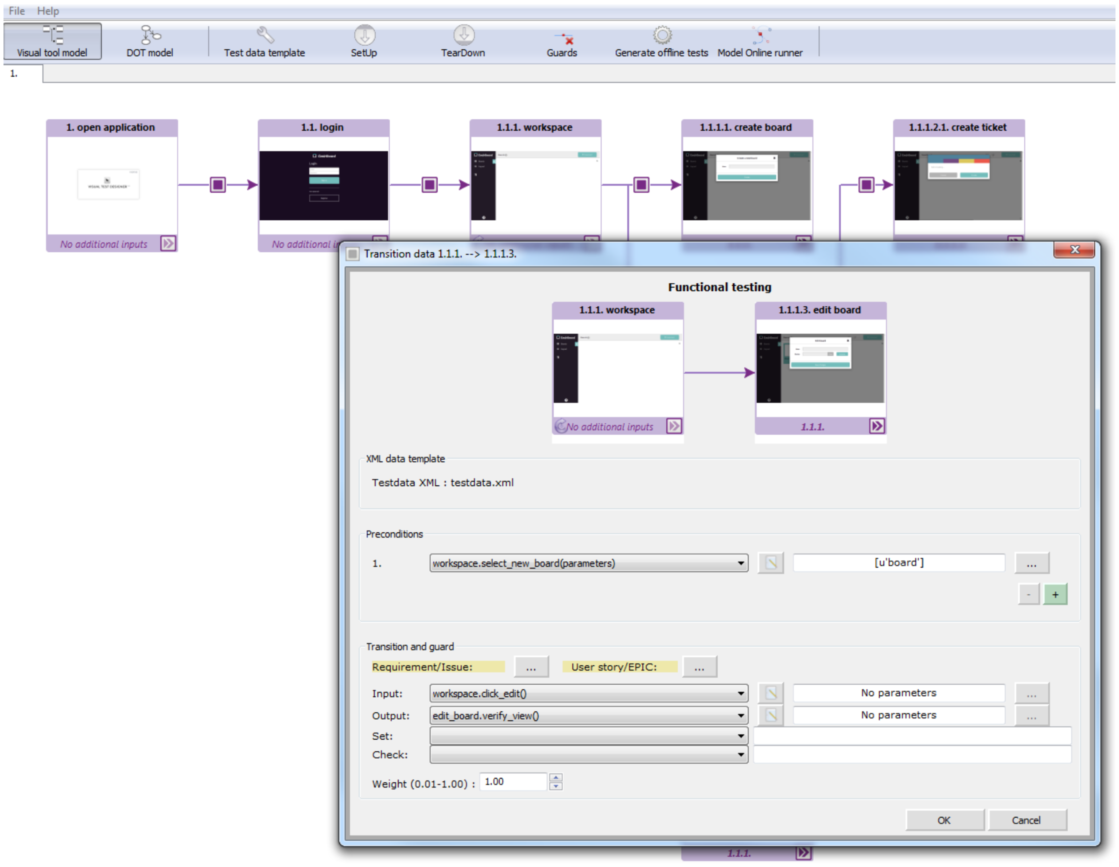
Console output

```
This page model (edit_board) is currently edited in Designer tab --- skipping!
This page model (edit_ticket) is currently edited in Designer tab --- skipping!
This page model (login) is currently edited in Designer tab --- skipping!
This page model (workspace) is currently edited in Designer tab --- skipping!
DONE: Page models were updated.
This page model (login) is currently edited in Designer tab --- skipping!
DONE: Page models were updated.
```

Figure 18: IVT UI - Maintenance tool

8.4 Model graph

Actual inputs and outputs for the page models are defined in edge nodes at the Model Graph -tool (see Figure 19 for an example).



XML data template

Testdata XML : testdata.xml

Preconditions

1. workspace.select_new_board(parameters) [u'board']

Transition and guard

Requirement/Issue: User story/EPIC:

Input: workspace.click_edit() No parameters

Output: edit_board.verify_view() No parameters

Set:

Check:

Weight (0.01-1.00) : 1.00

Figure 19: IVT UI - Edge node in model graph

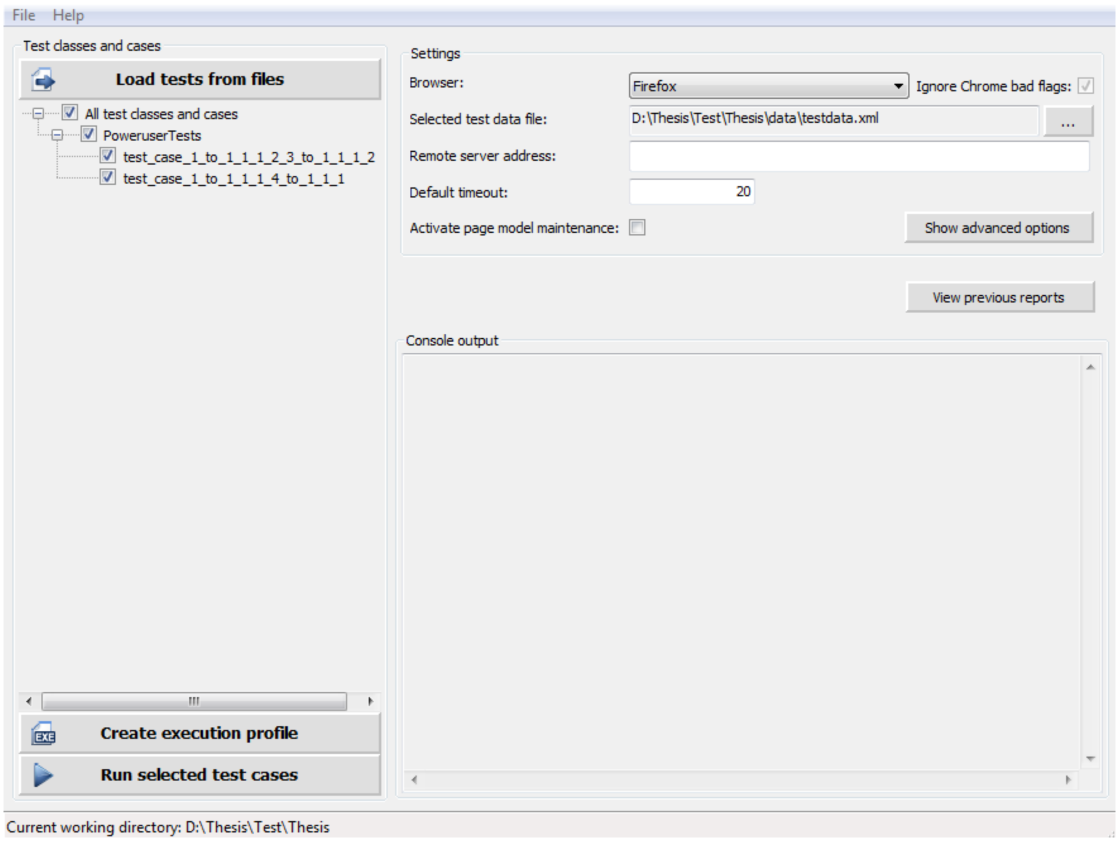
User can insert methods created in Designer-tool to inputs, outputs and preconditions. Usually input is the action that moves to new page model and output is the step for verifying it. Preconditions are always run before the input action.

If Figure 19 is examined, precondition and input methods are located in the page model where the transaction begins and output method is located in the page model the transaction ends.

When all transactions have been defined, test cases can be generated from the Model graph tool.

8.5 Test runner

Generated offline tests can be run from the Test runner -tool (see Figure 20 for an example). User can select the test file, choose test cases to be executed and select a test data.



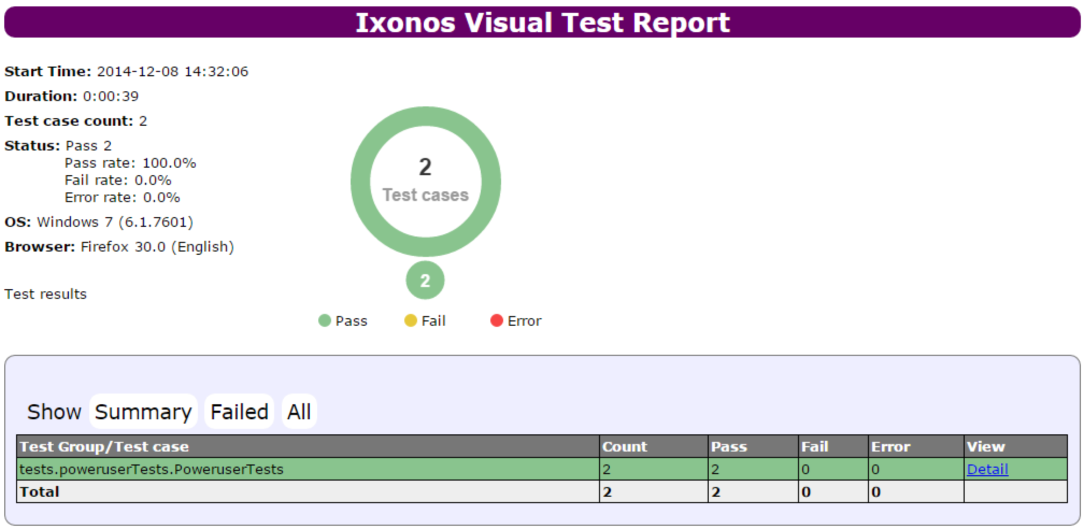
All test classes and cases
PowneruserTests
test_case_1_to_1_1_1_2_3_to_1_1_1_2
test_case_1_to_1_1_1_4_to_1_1_1

Browser: Firefox
Ignore Chrome bad flags: ✓
Selected test data file: D:\Thesis\Test\Thesis\data\testdata.xml
Remote server address:
Default timeout: 20
Activate page model maintenance: ☐

Current working directory: D:\Thesis\Test\Thesis

Figure 20: IVT UI - Test runner

The test execution is shown on the browser and user can see all the test cases in action. After the test run is concluded, IVT gives a report screen (see Figure 21 for an example).


Ixonos Visual Test Report

Start Time: 2014-12-08 14:32:06
Duration: 0:00:39
Test case count: 2
Status: Pass 2
Pass rate: 100.0%
Fail rate: 0.0%
Error rate: 0.0%
OS: Windows 7 (6.1.7601)
Browser: Firefox 30.0 (English)

Test results

Test Group/Test case	Count	Pass	Fail	Error	View
tests.poweruserTests.PoweruserTests	2	2	0	0	Detail
Total	**2**	**2**	**0**	**0**	

Figure 21: IVT UI - Test run results

Result are color coded as follows:

- Green means that the test case passed
- Yellow means that the test case failed
- Red means that there was an error in the test, usually a bug in the script

9 Implementing acceptance testing to Contriboard's development chain

9.1 Basis for the implementation

Currently Contriboard's production environment called Corolla v1.1 looks like as it is shown in Figure 22 below.

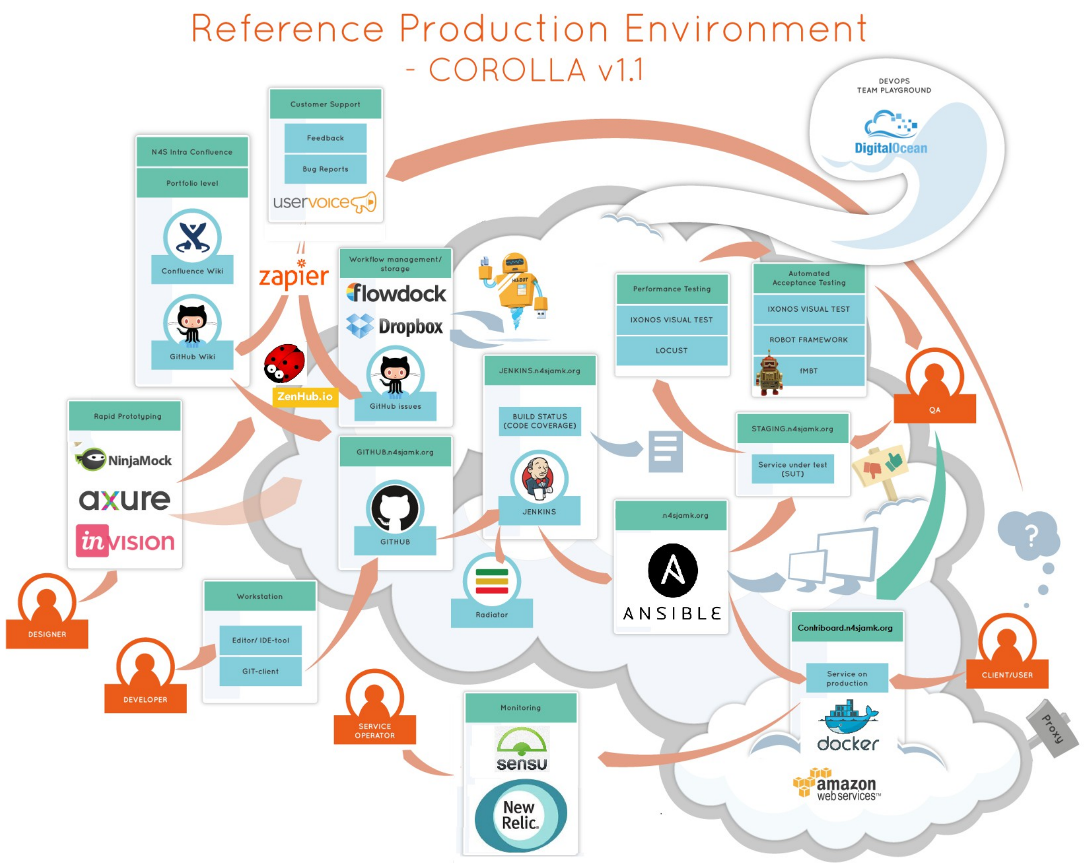


Figure 22: Corolla v1.1 - Contriboard production environment

Basically when new version is pushed to master branch in GitHub, Jenkins launches a job which tells Ansible to make a new build. After that the person responsible for quality assurance manually launches various tests with various tools.

What actually should be happening, is that when new version is pushed to GitHub, Jenkins should tell Ansible to build a development build, or possibly a testing build, depending on how the builds will be organized in the future. After that the Jenkins should start a new job that fires up the testers directly against the built SUT (see Figure 23 for sequence diagram of the flow).

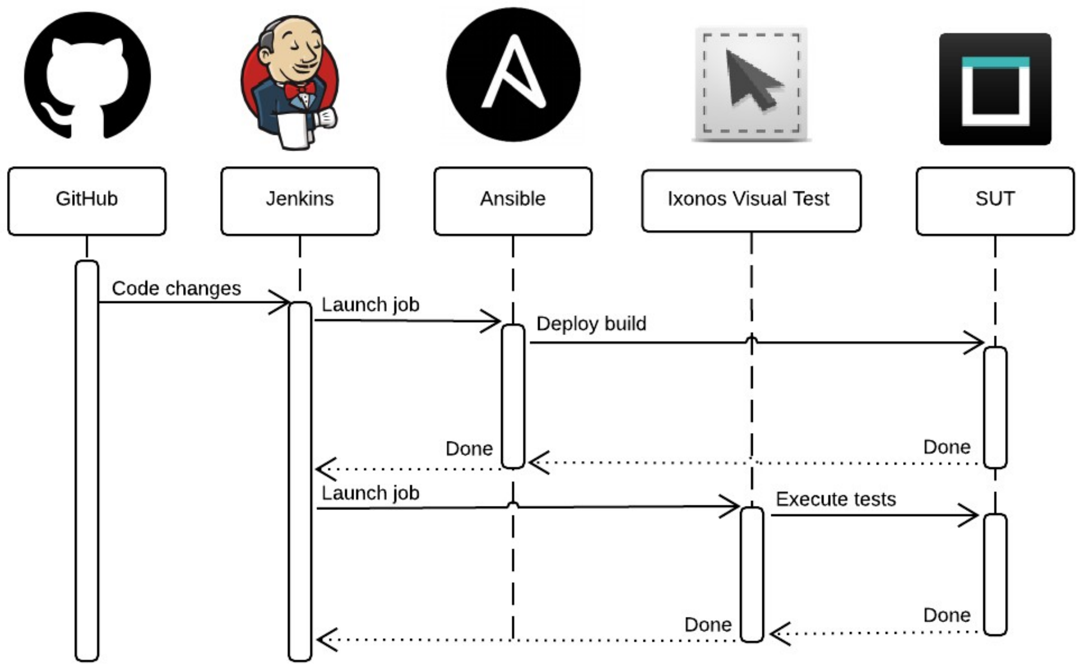


Figure 23: Sequence diagram of future test automation flow

Different kind of automated acceptance testing tools has been considered for the job as it can be seen in Figure 24 below.

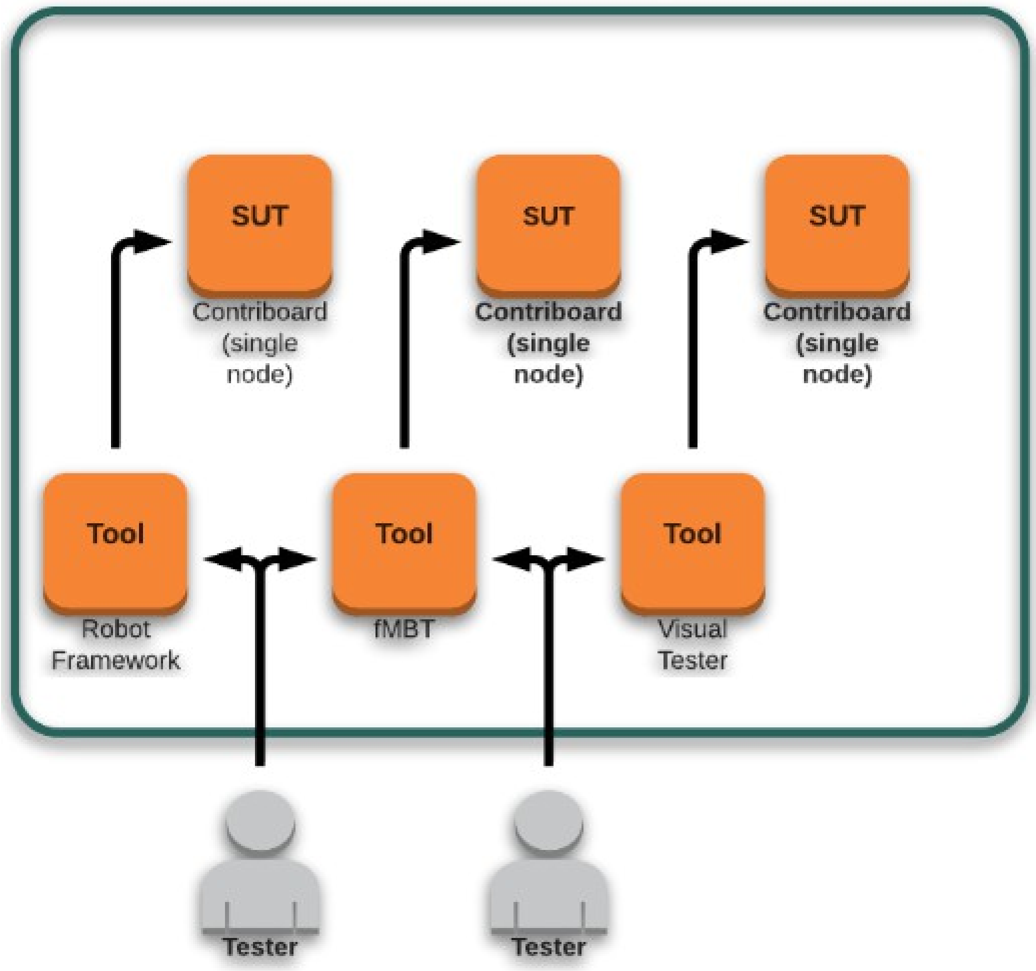

SUT
Contriboard (single node)
SUT
Contriboard (single node)
SUT
Contriboard (single node)
Tool
Robot Framework
Tool
fMBT
Tool
Visual Tester
Tester
Tester

Figure 24: Automated acceptance testing vision for Contriboard

In this thesis the main focus was to take the right side road where Ixonos Visual Test tool handles the automated acceptance testing and integrate it to the Corolla v1.1 development chain.

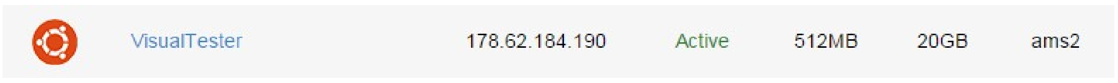
As there currently was no development build available, the implementation needed to be made against the production build. So in this case it was enough that testing environment was up and running, test cases were functional, test runner executed the created tests flawlessly and all this could be launched from the Jenkins manually.

9.2 Setting up the environment for Ixonos Visual Test

The first matter that needed to be set up was a Virtual Machine (VM) to host the Ixonos Visual Tester and other related packages and tools. VM was created to DigitalOcean's cloud, mainly because it has been used for other aspects of Contriboard's development too.

Specs for the VM were (see Figure 25):

- 64-bit Ubuntu 14.04 server
- 512MB memory
- 20GB disk space



VisualTester 178.62.184.190 Active 512MB 20GB ams2

Figure 25: Virtual Machine in DigitalOcean

After the VM was fired up, SSH connection was made through the Ubuntu's terminal at the local workstation to get inside the virtual machine. After that apt-get update and upgrade were run to make sure every software component was up-to-date.

Latest version (1.6.2 developer at the time) of the Ixonos Visual Test 64-bit Linux package was downloaded from Ixonos' cloud at visualtest.ixonos.com to the local workstation. Files were extracted and copied to the VM through scp command.

Next Ixonos Visual Test software itself and all the relevant packages for its functionality were installed following the instructions in the readme file included in the package (see Appendix 3 for detailed instructions). After that license key was activated by creating a licence.info file to /home/ixonos-visual-test -folder. License information provided by Ixonos personnel was then copied to the file to make it active.

Then it was essential to install Firefox for executing the tests in a browser and X virtual framebuffer (Xvfb) for running the tests in memory instead of showing them on the screen. Xvfb was started as a background process in the virtual machine. With these software installed, the environment was ready for executing the tests on the virtual machine using Ixonos Visual Test software headless.

See Appendix 2 for more detailed setup instructions.

9.3 Designing test model

The main objective for the thesis was to implement the model-based acceptance testing process to the continuous delivery chain. The test model was intentionally left as lightweight as possible. Reason for this is that the Contriboard is receiving big overhaul in the near future and model needs to be changed drastically. Also if needed, the model can be expanded very easily at its current state without affecting the continuous delivery chain itself.

Model was designed to test basic power user interactions which included:

- Signing in
- Creating a new board
- Editing the board
- Deleting the board
- Creating a new ticket inside the board
- Editing the ticket
- Deleting the ticket

The reasons for leaving out some other basic interactions are discussed more on the results chapter of the thesis.

Test flow

Pre-conditions: User has an active power user account for Contriboard and its workspace is empty.

1. User signs in to Contriboard with:
    - Username: poweruser@thesis.com
    - Password: pa55w0rd
2. User is directed to workspace view.
3. User creates a board and names it "New Board".
4. User enters the created board and is directed to specific board view.

5. User creates a red ticket and names it "New Ticket".
6. User edits the ticket:
   - Changes the color to purple
   - Changes the name to "Edited Ticket"
7. User deletes the ticket.
8. User navigates to workspace view.
   1. User edits the board:
      - Changes the name to "Edited Board"
9. User deletes the board

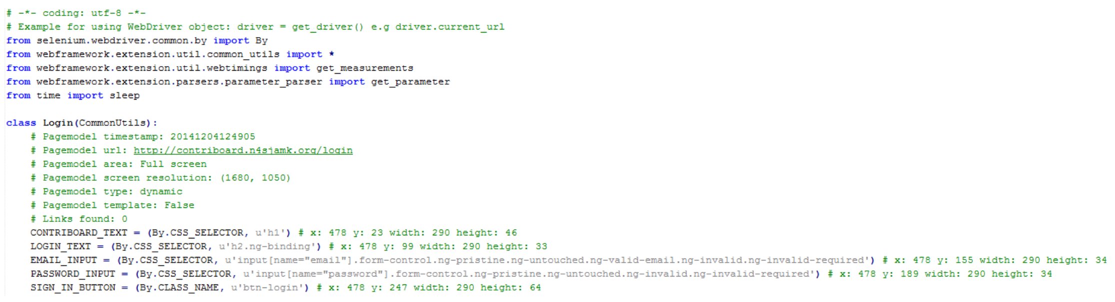
Post-conditions: User's workspace is empty.

See appendix 1 for more detailed and illustrated test flow.

9.3 Creating tests

9.3.1 Creating page models

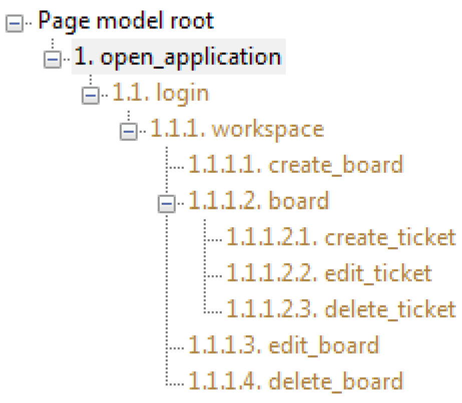
Actual work on the test model started with creating the page models. It was done by opening the desired page in the browser, for example login, and by clicking the Create Page Models button in Ixonos Visual Test. IVT found all the elements from the page and saved them automatically to the page model file. See Figure 26 below for example of automatically created login screen page model.



```
# -*- coding: utf-8 -*-
# Example for using WebDriver object: driver = get_driver() e.g driver.current_url
from selenium.webdriver.common.by import By
from webframework.extension.util.common_utils import *
from webframework.extension.util.webtimings import get_measurements
from webframework.extension.parsers.parameter_parser import get_parameter
from time import sleep

class Login(CommonUtils):
    # Pagemodel timestamp: 20141204124905
    # Pagemodel url: http://contriboard.n4sjamk.org/login
    # Pagemodel area: Full screen
    # Pagemodel screen resolution: (1680, 1050)
    # Pagemodel type: dynamic
    # Pagemodel template: False
    # Links found: 0
    CONTRIBOARD_TEXT = (By.CSS_SELECTOR, u'h1') # x: 478 y: 23 width: 290 height: 46
    LOGIN_TEXT = (By.CSS_SELECTOR, u'h2.ng-binding') # x: 478 y: 99 width: 290 height: 33
    EMAIL_INPUT = (By.CSS_SELECTOR, u'input[name="email"].form-control.ng-pristine.ng-untouched.ng-valid-email.ng-invalid.ng-invalid-required') # x: 478 y: 155 width: 290 height: 34
    PASSWORD_INPUT = (By.CSS_SELECTOR, u'input[name="password"].form-control.ng-pristine.ng-untouched.ng-invalid.ng-invalid-required') # x: 478 y: 189 width: 290 height: 34
    SIGN_IN_BUTTON = (By.CLASS_NAME, u'btn-login') # x: 478 y: 247 width: 290 height: 64
```

Figure 26: IVT login screen page model

Page models were created for total of nine views. They can be seen in Figure 27 below showing the page model tree hierarchy.

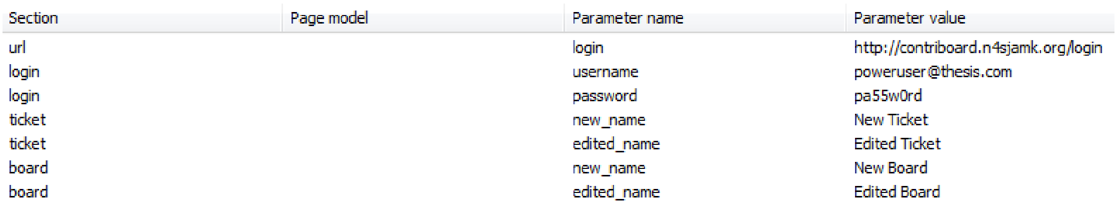


```
Page model root
  1. open_application
    1.1. login
      1.1.1. workspace
        1.1.1.1. create_board
        1.1.1.2. board
          1.1.1.2.1. create_ticket
          1.1.1.2.2. edit_ticket
          1.1.1.2.3. delete_ticket
        1.1.1.3. edit_board
        1.1.1.4. delete_board
```

Figure 27: Page model tree hierarchy

9.3.2 Creating test data

After page models were created, some test data was needed. That was done with IVT's testdata creation tool. For example user needed a username and password to sign in the Contriboard, so they were created under Login category. See Figure 28 below for all created test data.



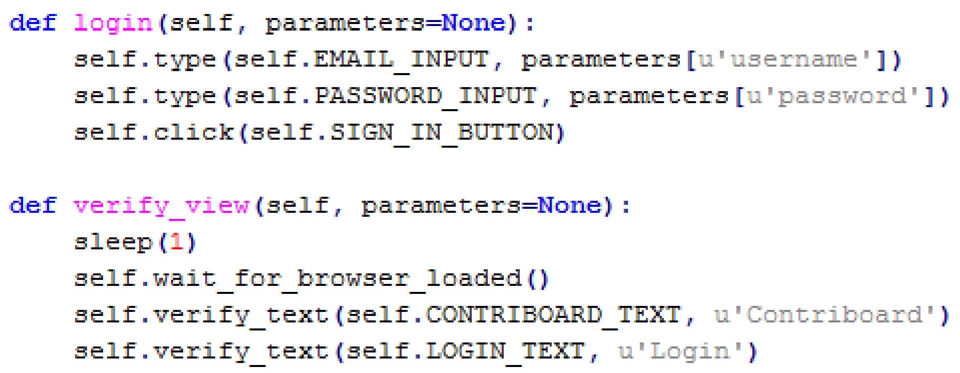
Section	Page model	Parameter name	Parameter value
url		login	http://contriboard.n4sjamk.org/login
login		username	poweruser@thesis.com
login		password	pa55w0rd
ticket		new_name	New Ticket
ticket		edited_name	Edited Ticket
board		new_name	New Board
board		edited_name	Edited Board

Figure 28: IVT test data

9.3.3 Creating methods to page models

Next in line was to implement functionality. This was done in the designer tool of IVT. By clicking the Add Method button, it was possible to create a new function by simply clicking the elements on the screen and choosing the functionality from the dropdown list. Some methods needed some manual Python scripting to make them work but most were done simply by choosing what was needed from IVT's selections.

See Figure 29 below for an example of login page model's functions.



```
def login(self, parameters=None):
    self.type(self.EMAIL_INPUT, parameters[u'username'])
    self.type(self.PASSWORD_INPUT, parameters[u'password'])
    self.click(self.SIGN_IN_BUTTON)

def verify_view(self, parameters=None):
    sleep(1)
    self.wait_for_browser_loaded()
    self.verify_text(self.CONTRIBOARD_TEXT, u'Contriboard')
    self.verify_text(self.LOGIN_TEXT, u'Login')
```

Figure 29: IVT login screen page model methods

The first method simply types the username and password to the right fields in browser and then clicks the sign in button.

The second method verifies that the login screen is visible at the moment.

9.3.4 Adding functionality to the model

The next step was to link all the methods to right transactions in the model. As it was seen earlier in Figure 27, the page models were created in certain tree hierarchy. Below, in Figure 30, the same hierarchy can be seen in visualized format.

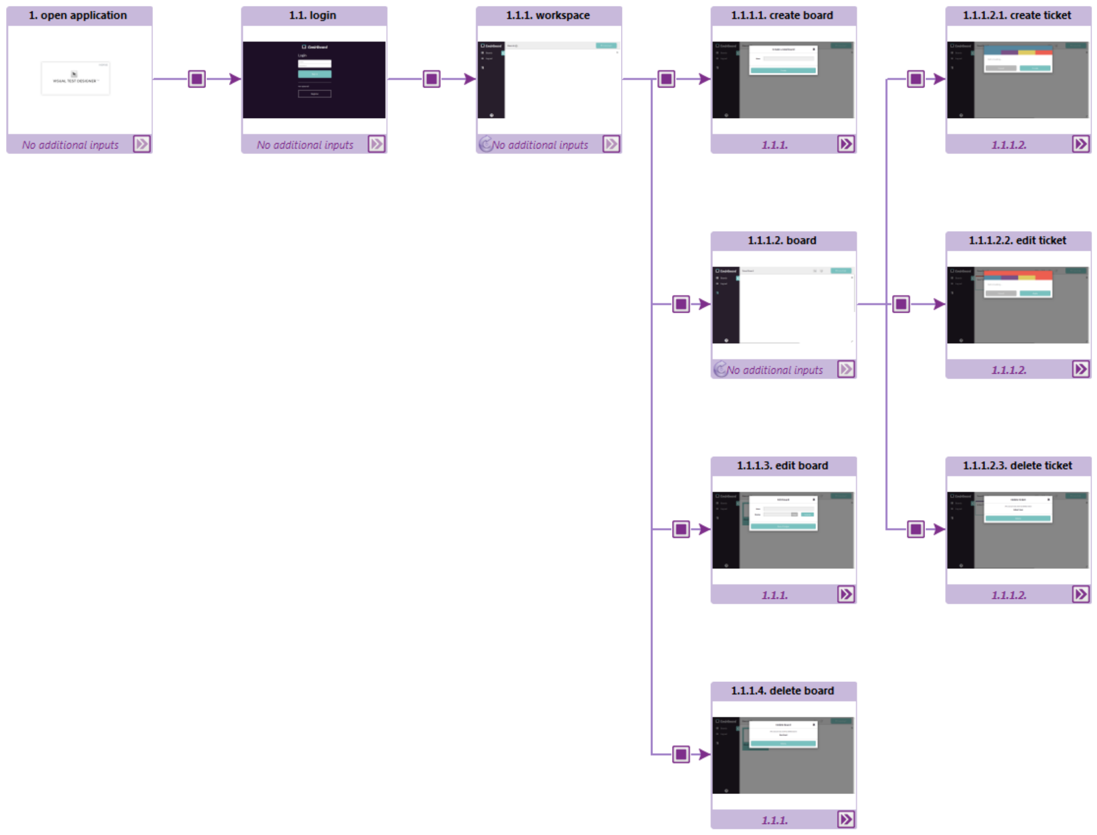
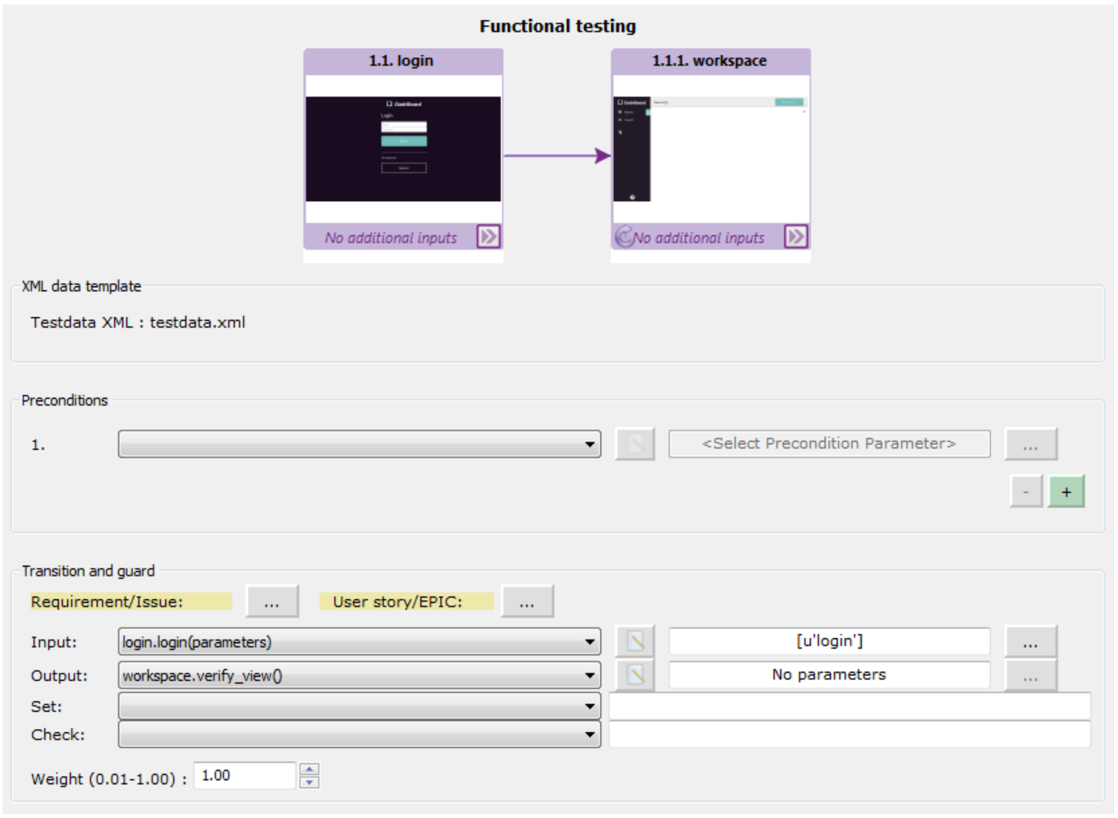


Figure 30: IVT visualized model

In the figure above, there are nodes between the different views. Those are the edges where the transactions take place. Login page model for example used those two methods: one for signing in and one for verifying the view. After login comes the workspace page model, which also has a method for verifying the view. Every edge has input and output (see Figure 31 for screenshot of edge between login and workspace) which need to be defined. When the test is moving from login page model to the workspace page model, the input method is called from the login and the output method is called from the workspace.



XML data template
Testdata XML : testdata.xml

Preconditions
1. <Select Precondition Parameter>

Transition and guard
Requirement/Issue: ...
User story/EPIC: ...
Input: login.login(parameters) [u'login']
Output: workspace.verify_view() No parameters
Set:
Check:
Weight (0.01-1.00) : 1.00

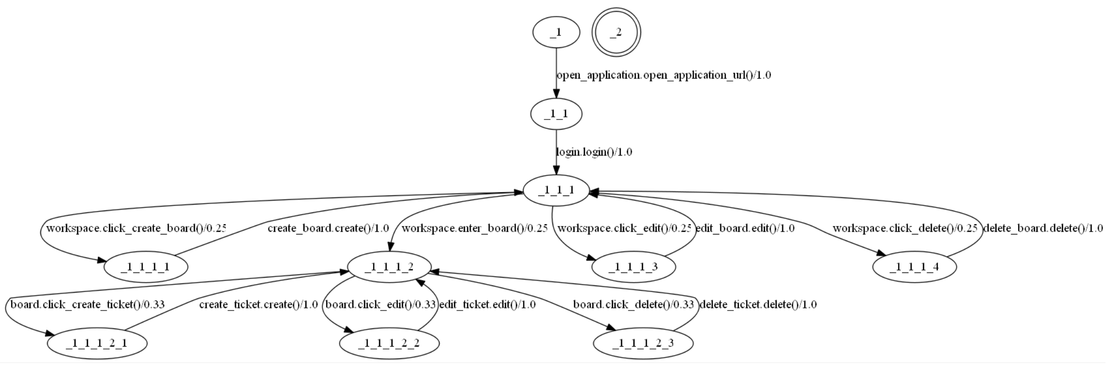
Figure 31: IVT edge node between login and workspace

In other words, using the above edge example, the user types in username and password and clicks the Sign in button. Input is completed and Contriboard moves to workspace view. Next the output method is called and verification of the workspace view begins. If it is succeeded the edge is covered. If for some reason either the input or output method fails, the test step is failed and seen that way in the report at the end.

Some of the page models needed references back to earlier page model when the test step was completed. For example when board was created, the reference was set back to workspace from the create board page model. References were made in the same way as the edges with input and output.

Additionally, some transactions needed a precondition. For example editing a board requires user to actually click the board active before clicking the edit button. These were simply their own methods in the same page model as the input method was and set in the same place as the input and output methods were (see Figure 31 for an example).

After all the edges and references were covered, the model could be viewed in DOT format as seen in the Figure 32 below.


_1
_2
open_application.open_application_url()/1.0
_1_1
login.login()/1.0
_1_1_1
workspace.click_create_board()/0.25
create_board.create()/1.0
workspace.enter_board()/0.25
workspace.click_edit()/0.25
edit_board.edit()/1.0
workspace.click_delete()/0.25
delete_board.delete()/1.0
_1_1_1_1
_1_1_1_2
_1_1_1_3
_1_1_1_4
board.click_create_ticket()/0.33
create_ticket.create()/1.0
board.click_edit()/0.33
edit_ticket.edit()/1.0
board.click_delete()/0.33
delete_ticket.delete()/1.0
_1_1_1_2_1
_1_1_1_2_2
_1_1_1_2_3
Figure 32: IVT model in DOT view

9.3.5 Generating test cases

Finally, the actual test file was built with clicking the create offline tests button. IVT automatically generated test cases based on possible ways the model can be used. In this case there was two test cases created as it can be seen in the Figure 33 below.

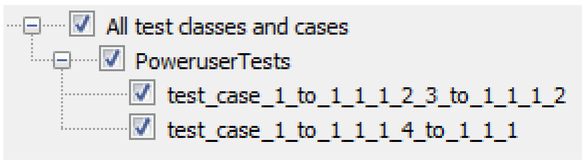

All test classes and cases
PoweruserTests
test_case_1_to_1_1_1_2_3_to_1_1_1_2
test_case_1_to_1_1_1_4_to_1_1_1

Figure 33: IVT generated offline test cases

The first test case flow:

- Browser goes to login screen
- User signs in
- User creates a new board
- User navigates to the board
- User creates a new ticket
- User edits the ticket
- User deletes the ticket

The second test case flow:

- Browser goes to login screen
- User edits the board
- User deletes the board

The test cases were both located in the same test file which could be run from the IVT test runner tool or from the command line as it was done in the Jenkins job.

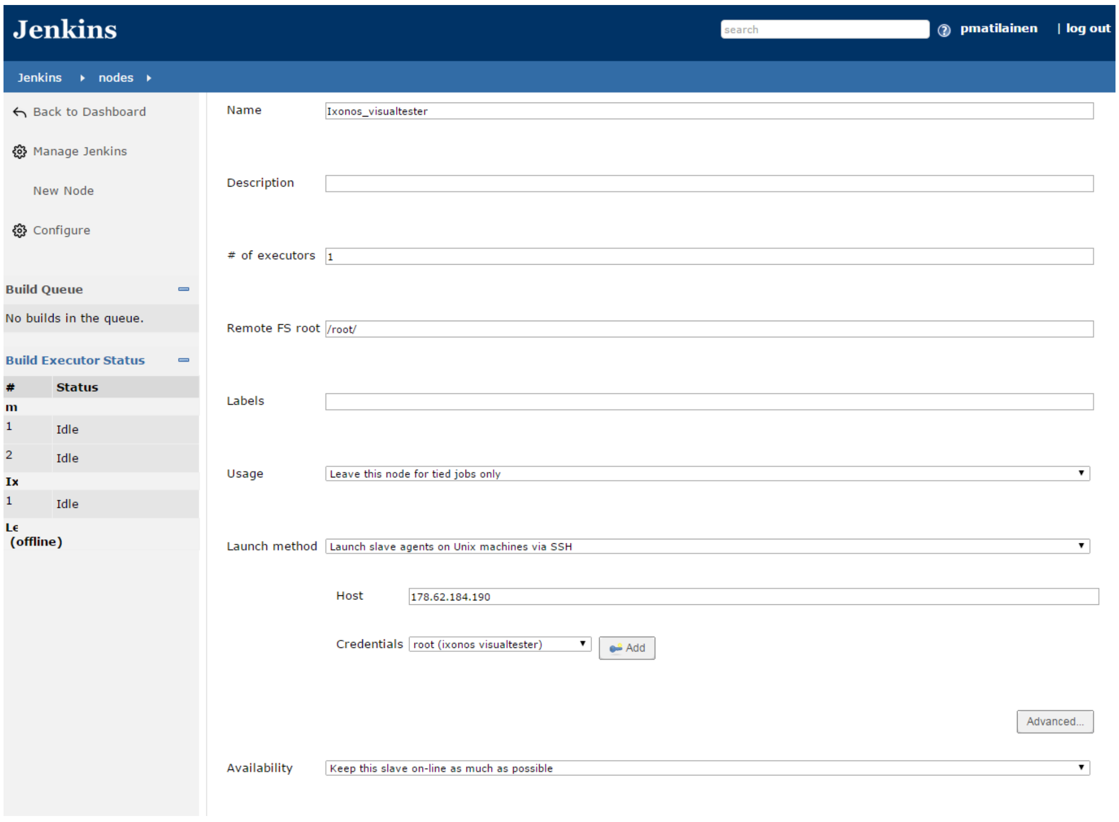
9.5 Jenkins integration

Before anything else, the test files created with Ixonos Visual Test tool was moved to the virtual machine.

9.5.1 Configuring Jenkins

To run Jenkins, Java was needed to be installed to the virtual machine and after that the root user needed SSH-key pair which was generated inside the VM. After generation process, public key was added to the authorized keys location in VM and private key to the Jenkins itself. New credentials were made in Jenkins and private key set to that user.

After SSH-keys were set, next in line was to create a slave node to Jenkins. Node was set up as shown in Figure 34 below.



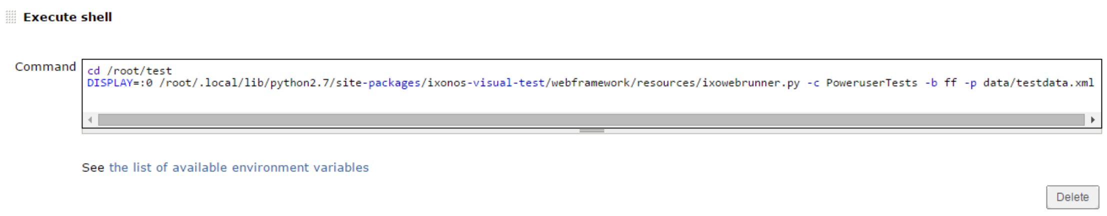
#	Status
m	
1	Idle
2	Idle
Ix	
1	Idle
Le (offline)	

Name	Ixonos_visualtester
Description	
# of executors	1
Remote FS root	/root/
Labels	
Usage	Leave this node for tied jobs only
Launch method	Launch slave agents on Unix machines via SSH
Host	178.62.184.190
Credentials	root (ixonos visualtester)
Availability	Keep this slave on-line as much as possible

Figure 34: Jenkins Node

After slave node was configured, it was time to create the actual job for executing the test run. Real case scenario would be that the job was executed every time there was new code pushed to the GitHub where the hooks resides, but in this case it was simplified that the job could be executed manually. Reasons for this decision are described in results chapter.

Script for running the test itself in the job is shown in the Figure 35 below.

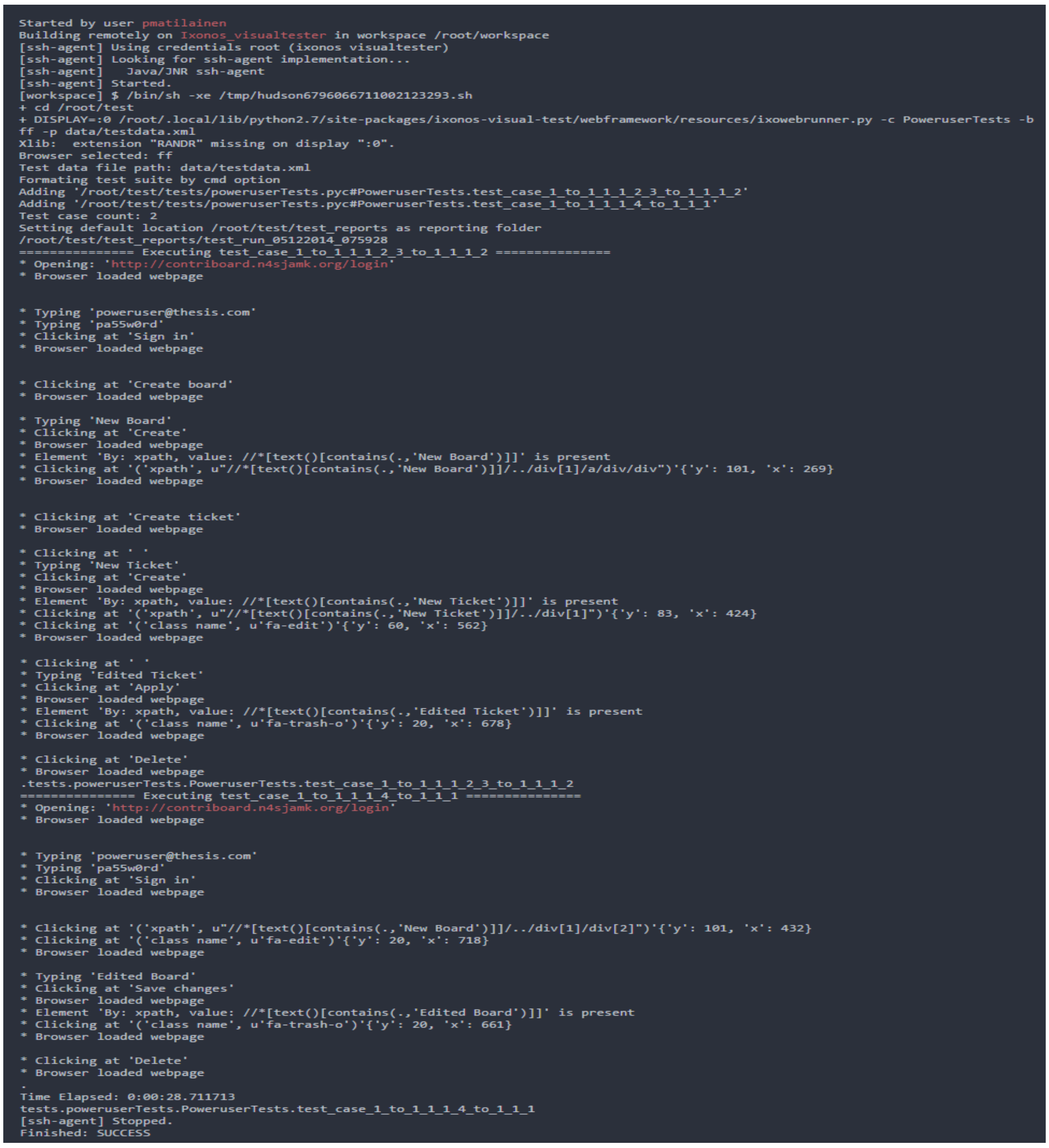


```
cd /root/test
DISPLAY=:0 /root/.local/lib/python2.7/site-packages/ixonos-visual-test/webframework/resources/ixowebrunner.py -c PoweruserTests -b ff -p data/testdata.xml
```

Figure 35: Test execution command

9.5.2 Running tests and output

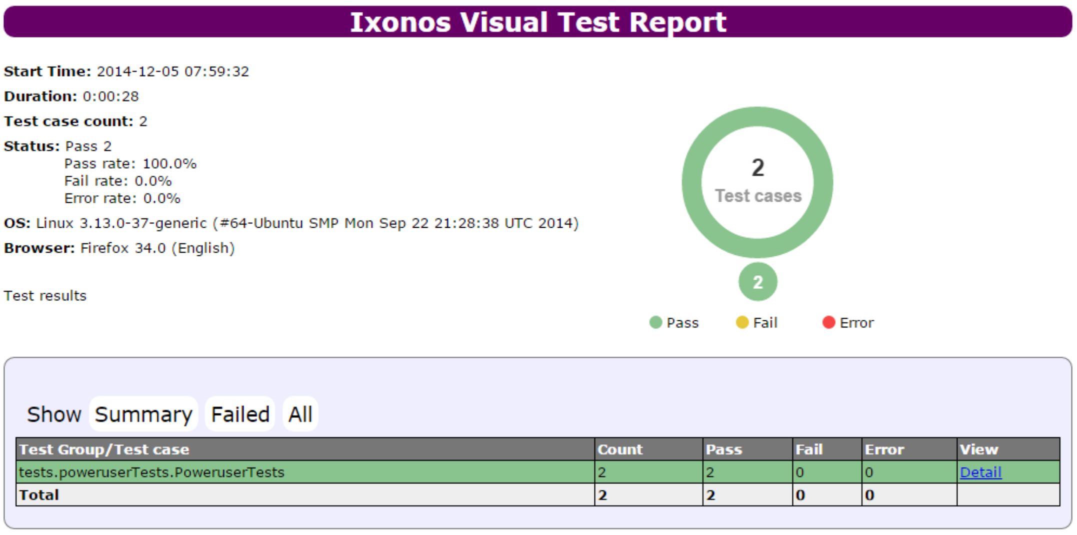
Like stated earlier, tests at this case were run manually by clicking the "build now" button in Jenkins. One build took somewhere around 30 seconds and after it was executed, the console output could be examined. Here in Figure 36 the output for successful build can be seen.



```
Started by user pmatilainen
Building remotely on Ixonos_visualtester in workspace /root/workspace
[ssh-agent] Using credentials root (ixonos visualtester)
[ssh-agent] Looking for ssh-agent implementation...
[ssh-agent]   Java/JNR ssh-agent
[ssh-agent] Started.
[workspace] $ /bin/sh -xe /tmp/hudson6796066711002123293.sh
+ cd /root/test
+ DISPLAY=:0 /root/.local/lib/python2.7/site-packages/ixonos-visual-test/webframework/resources/ixowebrunner.py -c PoweruserTests -b
ff -p data/testdata.xml
Xlib:  extension "RANDR" missing on display ":0".
Browser selected: ff
Test data file path: data/testdata.xml
Formating test suite by cmd option
Adding '/root/test/tests/poweruserTests.pyc#PoweruserTests.test_case_1_to_1_1_1_2_3_to_1_1_1_2'
Adding '/root/test/tests/poweruserTests.pyc#PoweruserTests.test_case_1_to_1_1_1_4_to_1_1_1'
Test case count: 2
Setting default location /root/test/test_reports as reporting folder
/root/test/test_reports/test_run_05122014_075928
=============== Executing test_case_1_to_1_1_1_2_3_to_1_1_1_2 ===============
* Opening: 'http://contriboard.n4sjamk.org/login'
* Browser loaded webpage


* Typing 'poweruser@thesis.com'
* Typing 'pa55w0rd'
* Clicking at 'Sign in'
* Browser loaded webpage


* Clicking at 'Create board'
* Browser loaded webpage

* Typing 'New Board'
* Clicking at 'Create'
* Browser loaded webpage
* Element 'By: xpath, value: //*[text()[contains(.,'New Board')]]' is present
* Clicking at '('xpath', u"//*[text()[contains(.,'New Board')]]/../div[1]/a/div/div")'{'y': 101, 'x': 269}
* Browser loaded webpage


* Clicking at 'Create ticket'
* Browser loaded webpage

* Clicking at ' '
* Typing 'New Ticket'
* Clicking at 'Create'
* Browser loaded webpage
* Element 'By: xpath, value: //*[text()[contains(.,'New Ticket')]]' is present
* Clicking at '('xpath', u"//*[text()[contains(.,'New Ticket')]]/../div[1]")'{'y': 83, 'x': 424}
* Clicking at '('class name', u'fa-edit')'{'y': 60, 'x': 562}
* Browser loaded webpage

* Clicking at ' '
* Typing 'Edited Ticket'
* Clicking at 'Apply'
* Browser loaded webpage
* Element 'By: xpath, value: //*[text()[contains(.,'Edited Ticket')]]' is present
* Clicking at '('class name', u'fa-trash-o')'{'y': 20, 'x': 678}
* Browser loaded webpage

* Clicking at 'Delete'
* Browser loaded webpage
.tests.poweruserTests.PoweruserTests.test_case_1_to_1_1_1_2_3_to_1_1_1_2
=============== Executing test_case_1_to_1_1_1_4_to_1_1_1 ===============
* Opening: 'http://contriboard.n4sjamk.org/login'
* Browser loaded webpage


* Typing 'poweruser@thesis.com'
* Typing 'pa55w0rd'
* Clicking at 'Sign in'
* Browser loaded webpage


* Clicking at '('xpath', u"//*[text()[contains(.,'New Board')]]/../div[1]/div[2]")'{'y': 101, 'x': 432}
* Clicking at '('class name', u'fa-edit')'{'y': 20, 'x': 718}
* Browser loaded webpage

* Typing 'Edited Board'
* Clicking at 'Save changes'
* Browser loaded webpage
* Element 'By: xpath, value: //*[text()[contains(.,'Edited Board')]]' is present
* Clicking at '('class name', u'fa-trash-o')'{'y': 20, 'x': 661}
* Browser loaded webpage

* Clicking at 'Delete'
* Browser loaded webpage
.
Time Elapsed: 0:00:28.711713
tests.poweruserTests.PoweruserTests.test_case_1_to_1_1_1_4_to_1_1_1
[ssh-agent] Stopped.
Finished: SUCCESS
```

Figure 36: Output of successful Jenkins job build

Since Jenkins only shows the console log of the test run, the implementation for showing the test report of Ixonos Visual Test tool's own format was made with using Apache2 in the VM and redirecting the results to website in address http://178.62.184.190/reports/. Here in Figure 37 the same successful test report can be seen in IVT's visualized format.


Ixonos Visual Test Report

Start Time: 2014-12-05 07:59:32
Duration: 0:00:28
Test case count: 2
Status: Pass 2
Pass rate: 100.0%
Fail rate: 0.0%
Error rate: 0.0%
OS: Linux 3.13.0-37-generic (#64-Ubuntu SMP Mon Sep 22 21:28:38 UTC 2014)
Browser: Firefox 34.0 (English)

Test results

Show Summary Failed All

Test Group/Test case	Count	Pass	Fail	Error	View
tests.poweruserTests.PoweruserTests	2	2	0	0	Detail
Total	**2**	**2**	**0**	**0**	

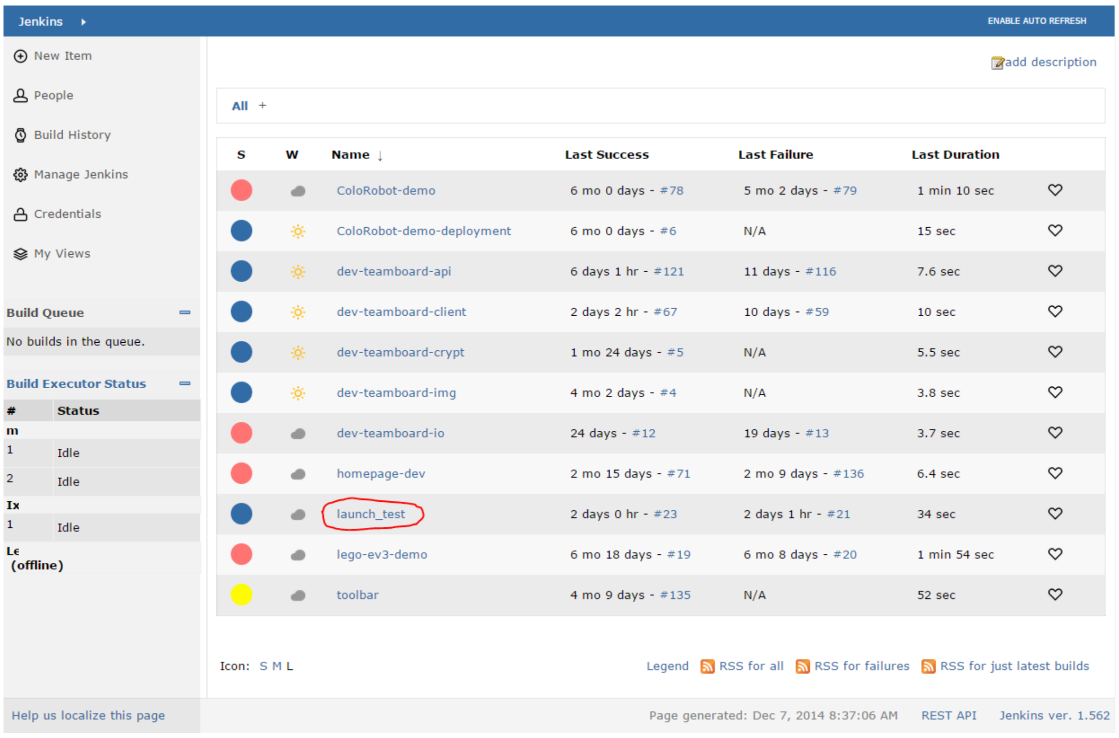
Figure 37: Ixonos Visual Test report of successful run

See Appendix 4 for detailed instructions.

10 Results and conclusions

10.1 Results

As a result, Contriboard has a working model-based acceptance testing solution that has been integrated to the Jenkins as it can be seen in the Figure 38 below.



Jenkins ▸ ENABLE AUTO REFRESH

New Item
People
Build History
Manage Jenkins
Credentials
My Views

Build Queue
No builds in the queue.

Build Executor Status

#	Status
m	
1	Idle
2	Idle
Ix	
1	Idle
Le (offline)	

add description

All +

S	W	Name ↓	Last Success	Last Failure	Last Duration	
🔴	☁	ColoRobot-demo	6 mo 0 days - #78	5 mo 2 days - #79	1 min 10 sec	♡
🔵	☀	ColoRobot-demo-deployment	6 mo 0 days - #6	N/A	15 sec	♡
🔵	☀	dev-teamboard-api	6 days 1 hr - #121	11 days - #116	7.6 sec	♡
🔵	☀	dev-teamboard-client	2 days 2 hr - #67	10 days - #59	10 sec	♡
🔵	☀	dev-teamboard-crypt	1 mo 24 days - #5	N/A	5.5 sec	♡
🔵	☀	dev-teamboard-img	4 mo 2 days - #4	N/A	3.8 sec	♡
🔴	☁	dev-teamboard-io	24 days - #12	19 days - #13	3.7 sec	♡
🔴	☁	homepage-dev	2 mo 15 days - #71	2 mo 9 days - #136	6.4 sec	♡
🔵	☁	launch_test	2 days 0 hr - #23	2 days 1 hr - #21	34 sec	♡
🔴	☁	lego-ev3-demo	6 mo 18 days - #19	6 mo 8 days - #20	1 min 54 sec	♡
🟡	☁	toolbar	4 mo 9 days - #135	N/A	52 sec	♡

Icon: S M L   Legend RSS for all RSS for failures RSS for just latest builds

Help us localize this page   Page generated: Dec 7, 2014 8:37:06 AM   REST API   Jenkins ver. 1.562

Figure 38: MBT acceptance test job in Jenkins

Because of the currently missing development build or testing build, the tests had to be made to run against the production build. This changed the actual implementation part of the thesis a little.

When development or testing build is brought online, the tests will use it as a SUT and Jenkins will be configured to launch the job every time new code is pushed to the GitHub's master branch. Now the implementation is temporarily so, that the testing process can be launched manually from the Jenkins.

Tests are run and the output can be viewed in two different format. First one is the Jenkin's own console log (see Figure 36 earlier for successful run) and Ixonos Visual Test tool's own visualized output in a web page (see Figure 37 earlier for successful run).

Tests were designed to be small and lightweight since there is a major overhaul coming to the Contriboard in the near future and most of the model will need to be changed. Tests currently simply demonstrate that everything works in the continuous delivery chain as it was defined in the objectives of the implementation part of the thesis. For example registering the user was not included in the test mainly because of the SUT being the production build. It would have been unwise to delete existing user from the database just for the sake of testing integration to the continuous delivery chain and same user cannot be registered twice.

10.2 Conclusions

Model-based testing in general feels very flexible way of testing systems which have different states and multiple ways of navigating to them. Creating the model from scratch might take some time but after it is done, the test generation happens automatically and saves a huge amount of time.

Like it was stated in the chapter about model-based testing, the amount of errors in fresh model can be a bit taunting at the beginning. But after the issues are solved, rest of the process feels very fluid.

Offline testing seems to work well with acceptance testing in mind. Every possible path is tested and even functionalities that could be forgotten are almost automatically brought in front of the tester's eyes when creating the model.

One big challenge in creating the tests time to time was using the Ixonos Visual Test. Software itself performs very well, but it is currently in constant development and sometimes when bigger changes happened, something broke down and actual test development of Contriboard went to a halt momentarily. The upside for this of course was the instant feedback opportunity. When there was a problem with the software, it was as easy as picking up the phone and calling to Anssi Pekkarinen at Ixonos and telling him about the issue. Usually it was solved and new version of the IVT software was available in a couple of days or earlier.

Biggest challenge in creating the tests and trying to maintain them through the autumn was definitely the testability of the Contriboard. Product was developed long time before any kind of proper testing solutions were implemented or even thought of, so there was some work to be done on that field. With this in mind, future testing implementations can be made a lot easier by better communication between testers and developers when designing new features and modifying the old ones.

Now that the implementation of model-based acceptance testing is done once and it has been linked to the Jenkins successfully, it can be re-used in other kind of testing processes, for example future solution for the performance testing.

Working on the thesis broadened my view of testing in general tremendously. Not to even mention the whole model-based testing concept, which was very much unknown when the work was started. Implementing the tests and integrating them to the Corolla v1.1 chain offered some challenges like mentioned earlier, but after actually solving the issues and finally getting everything to work, the practical knowledge had grown greatly.

References

Ammann, P., Offutt, J. 2008. Introduction to Software Testing. 1St Ed. New York City: Cambridge University press.

Cucumber. 2014. Project's web pages. Accessed on 6 December 2014. Retrieved from http://cukes.info/.

Contriboard. 2014. Project's GitHub wiki pages. Accessed on 20 October 2014. Retrieved from https://github.com/N4SJAMK/teamboard-meta/wiki.

FitNesse. 2014. What is FitNesse? Accessed on 6 December 2014. Retrieved from http://www.fitnesse.org/FitNesse.UserGuide.OneMinuteDescription.

Fowler, M. 2013. Continuous Delivery. Accessed on 18 November 2014. Retrieved from http://martinfowler.com/bliki/ContinuousDelivery.html.

International Software Testing Qualifications Board. 2014. Certification path. Accessed on 21 October 2014. Retrieved from http://www.istqb.org/certification-path-root.html.

ISTQB Exam Certification. 2014. What are Software Testing Levels? Accessed on 1 December 2014. Retrieved from http://istqbexamcertification.com/what-are-software-testing-levels/.

Ixonos Visual Test. 2014. Software's web pages. Accessed on 6 December 2014. Retrieved from http://www.ixonos.com/products/ixonos-visual-test.

Kervinen, A. 2014. fMBT – Overview. Accessed on 6 December 2014. Retrieved from https://01.org/fmbt/overview.

Klärck, P. 2009. Robot Framework Introduction. Accessed on 3 December 2014. Retrieved from http://www.slideshare.net/pekkaklarck/robot-framework-introduction.

Koirala, S., Sheikh, S. 2008. Software Testing: Interview Questions. 1St Ed. Hingham: Infinity Science Press.

Molyneayx, I. 2009. The Art of Application Performance Testing. 1St Ed. Sebastobol: O'Reilly Media, Inc.

Myers, G. J., Sandler, C. & Badgett, T. 2012. The Art of Software Testing. 3Rd Ed. Hoboken: John Wiley & Sons, Inc.

N4S-program. 2014. N4S-Program: Finnish Software Companies Speeding Digital Economy. Accessed on 20 October 2014. Retrieved from http://www.n4s.fi/en/.

Selenium Project. 2014. Introduction. Accessed on 6 December 2014. Retrieved from http://www.seleniumhq.org/docs/01_introducing_selenium.jsp.

Software Testing Class. 2012. Functional Testing vs Non-Functional Testing. Accessed on 19 November 2014. Retrieved from http://www.softwaretestingclass.com/functional-testing-vs-non-functional-testing/.

Havlat, M. 2014. TestLink Features. Accessed on 25 November 2014. Retrieved from http://testlink.sourceforge.net/docs/docs/features.php.

Utting, M., Legeard, B. 2007. Practical Model-Based Testing: A Tools Approach. 1St Ed. San Fransisco: Elsevier.

Tutorialspoint. 2014. Software Testing Types. Accessed on 25 November 2014. Retrieved from http://www.tutorialspoint.com/software_testing/testing_types.htm.

Watkins, J. & Mills, S. 2011. Testing IT: An Off-the-Shelf Software Testing Process. 2Nd Ed. New York City: Cambridge University Press.

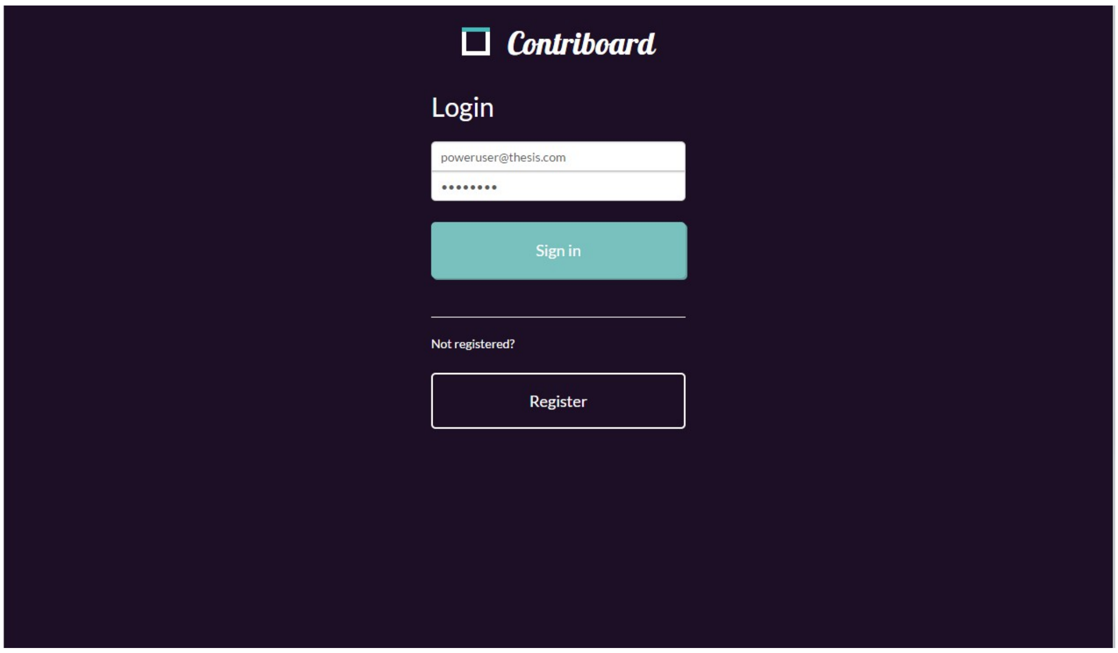
Appendices

Appendix 1: Illustrated test flow

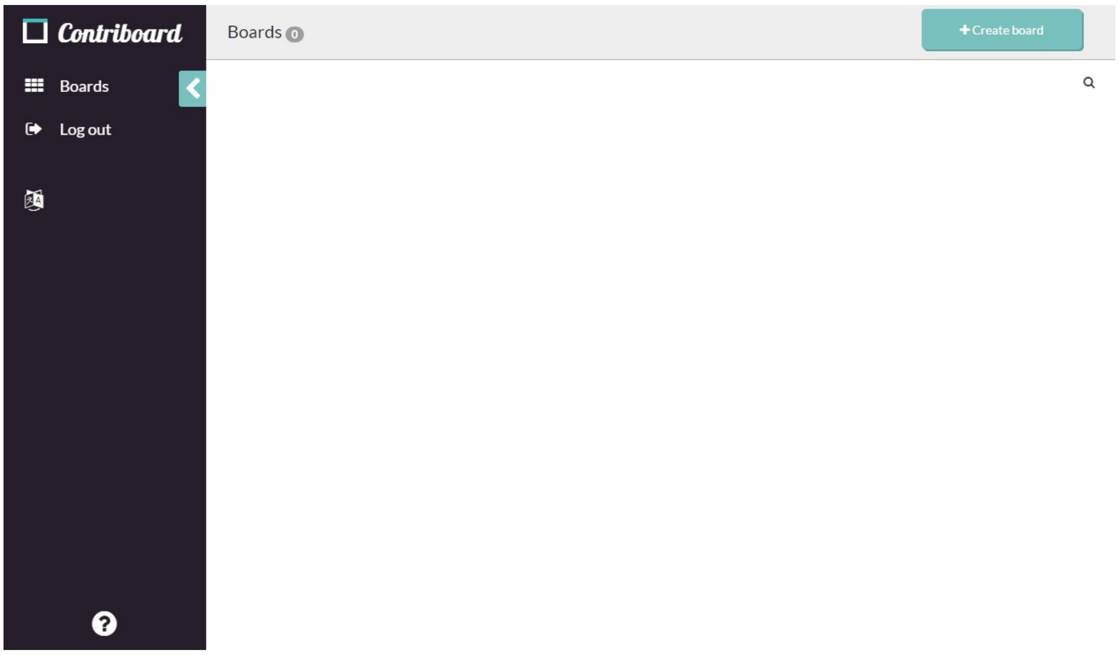
This process is automated with Ixonos Visual Test tool like stated earlier, but the tester here is referred as a user for better understanding of the process.

Test begins.

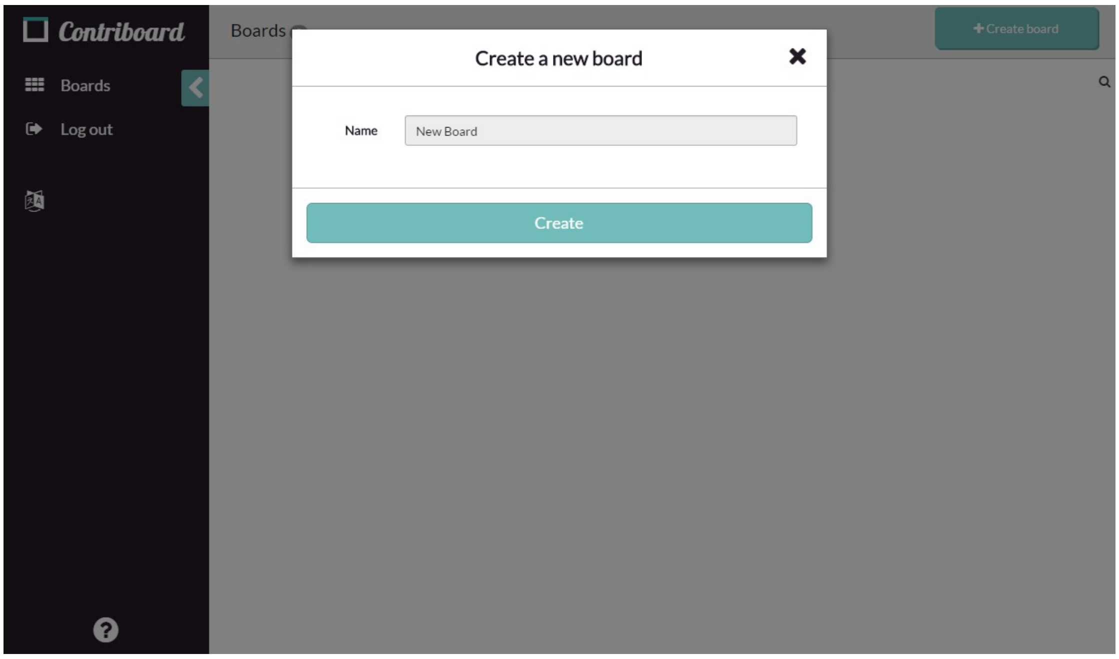
User opens the login screen of Contriboard and types username "poweruser@thesis.com" and password "pa55w0rd" to the fields.



User clicks Sign In button and is directed to empty workspace view.



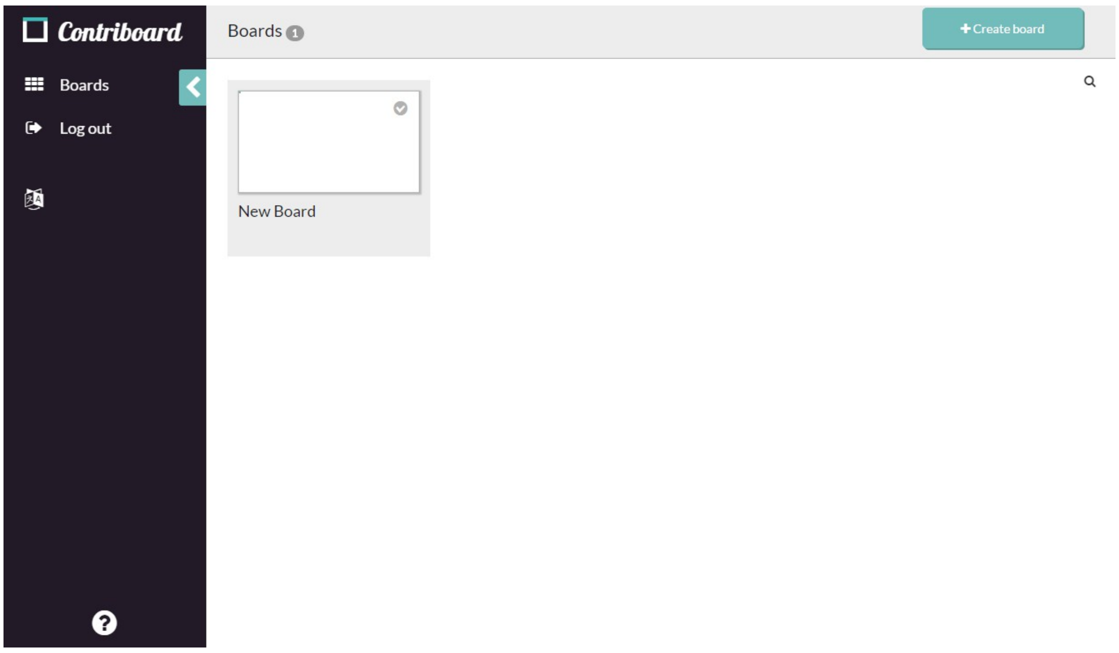
User clicks the Create Board button and enters name "New Board".



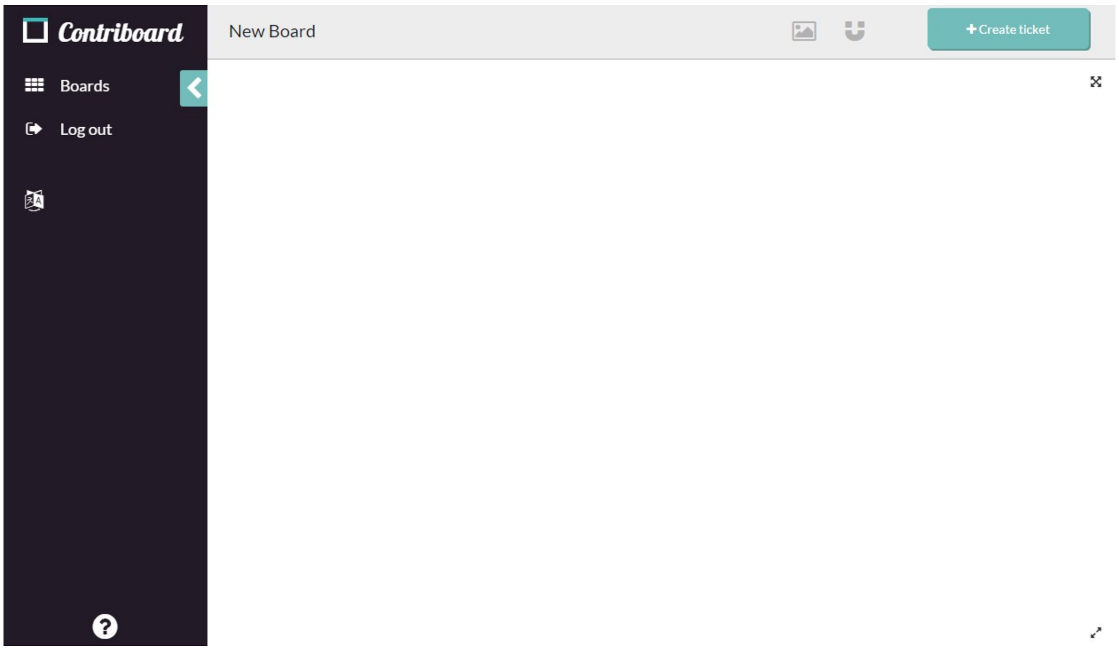
Create a new board

Name: New Board

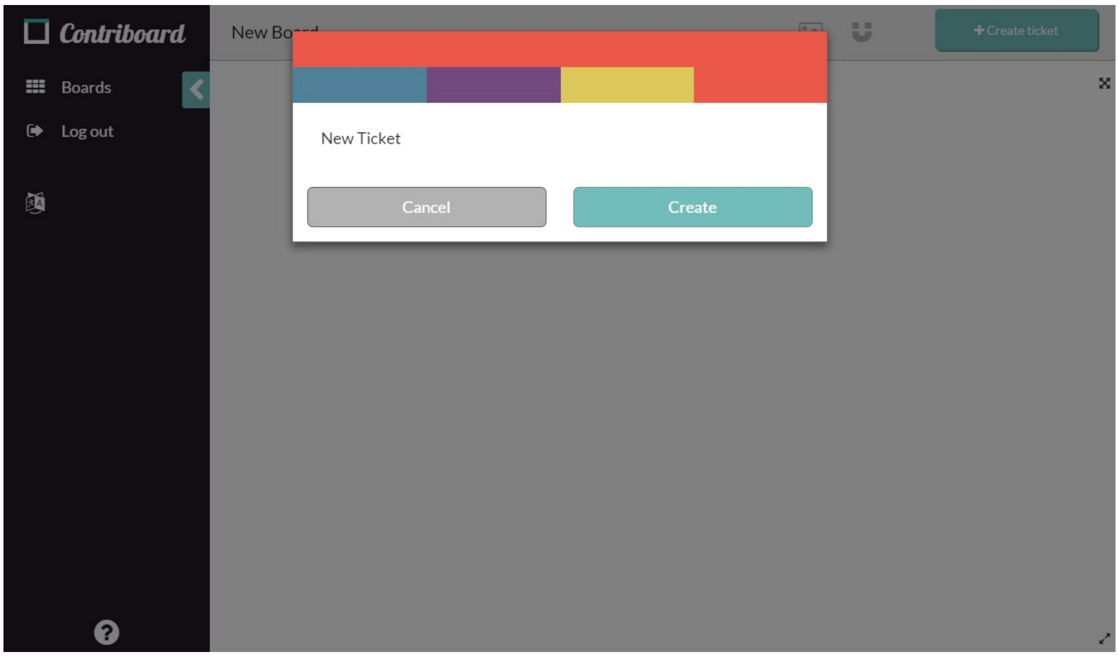
User clicks Create button in pop up and sees the board with name "New Board" on the workspace view.



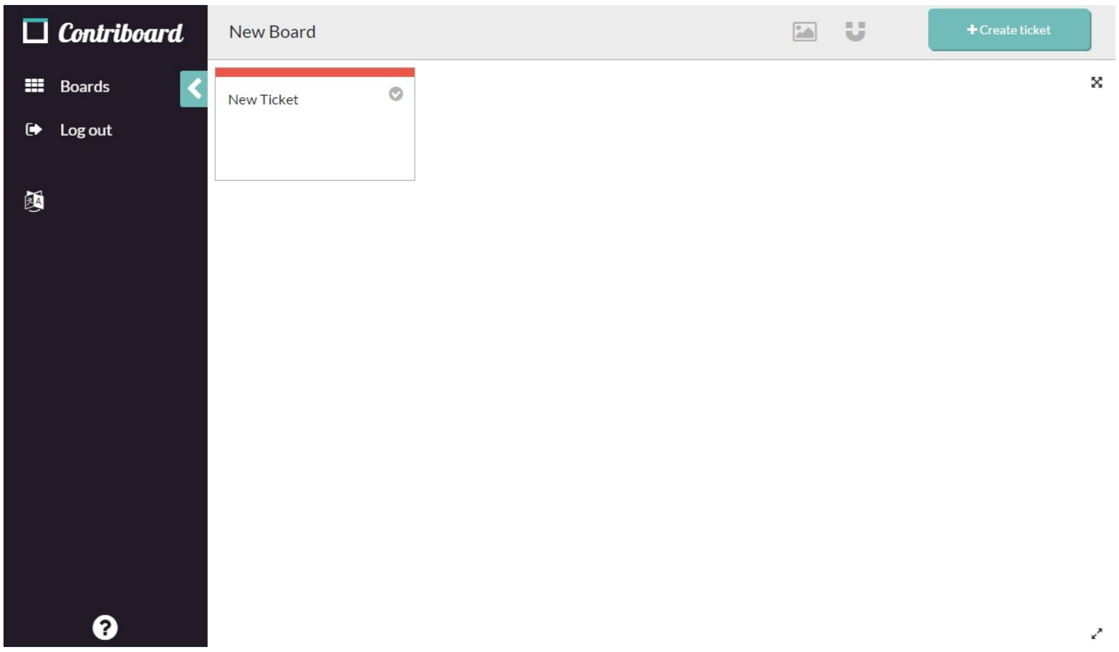
User clicks the board and is directed to the board specific view.



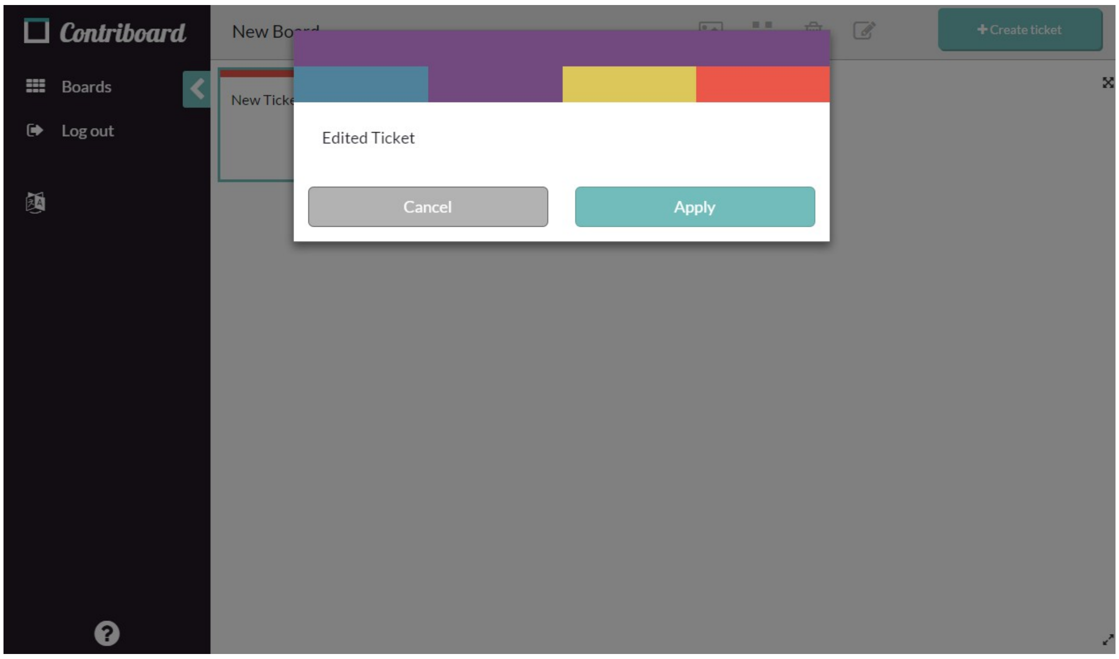
User clicks the Create Ticket button, selects red color and enters the name "New Ticket".



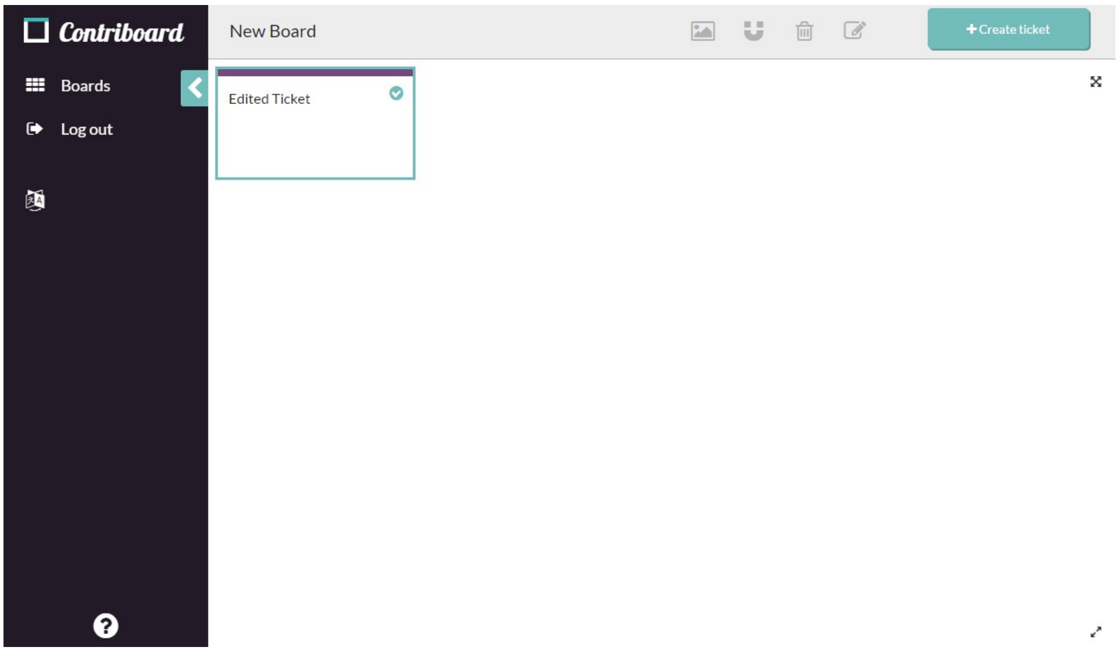
User clicks the Create button in pop up and sees the red ticket with name "New Ticket" in the board view.



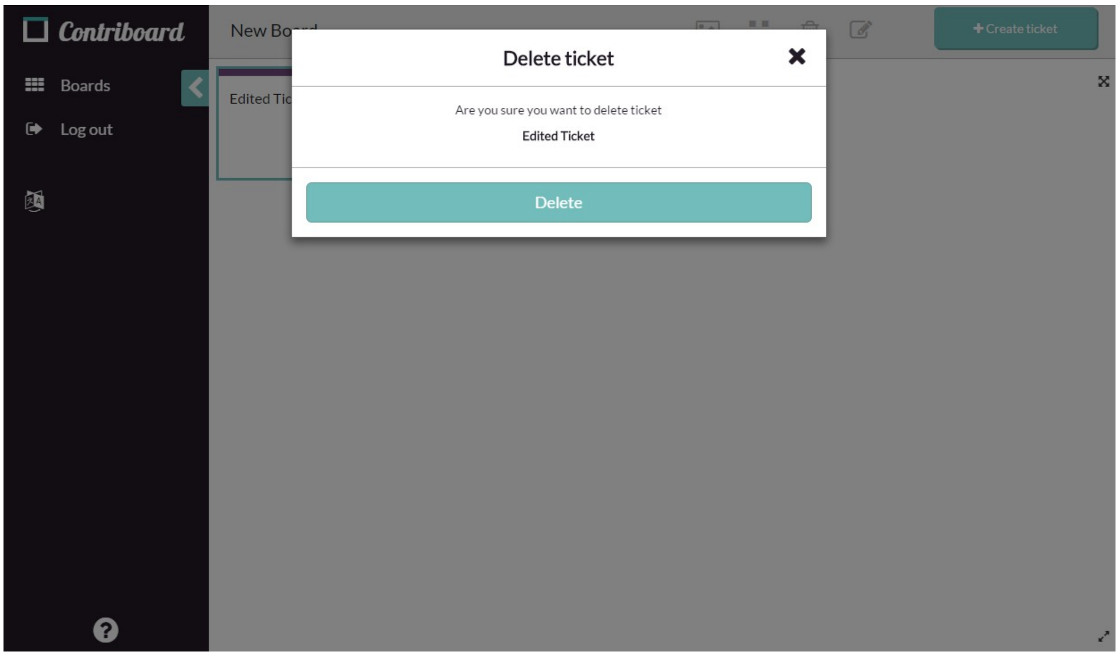
User selects the ticket, clicks the Edit icon, changes the color to purple and enters new name "Edited Ticket".



User clicks the Apply button in pop up and sees that the ticket has changed from red to purple and is now named "Edited Ticket".



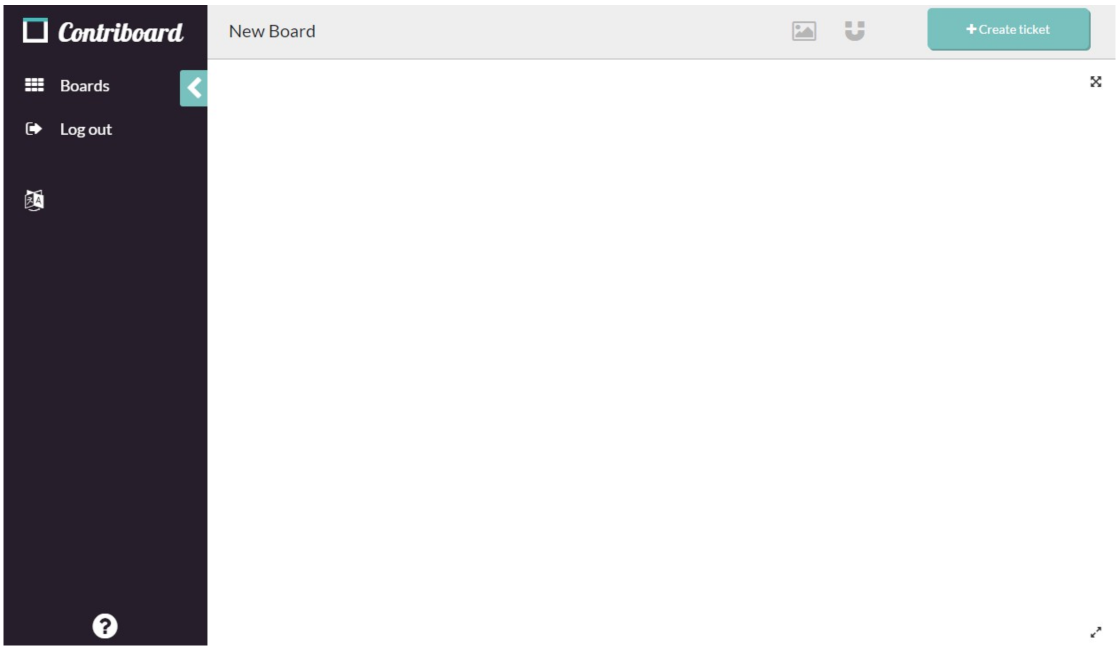
User leaves the ticket selected and clicks the Delete icon.



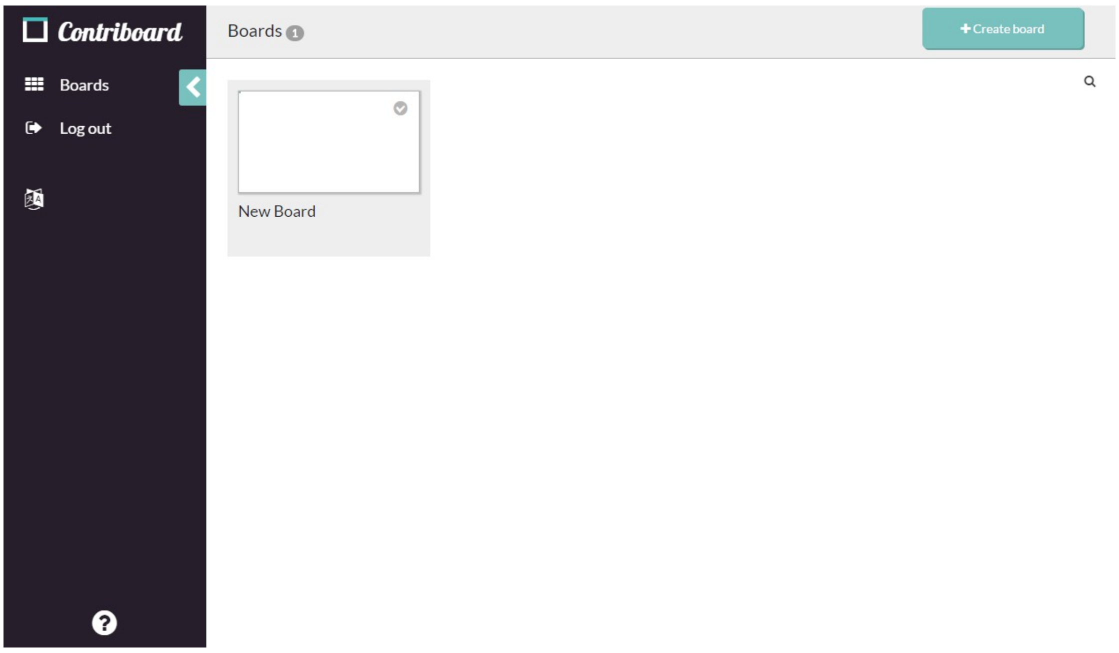
Are you sure you want to delete ticket

Edited Ticket

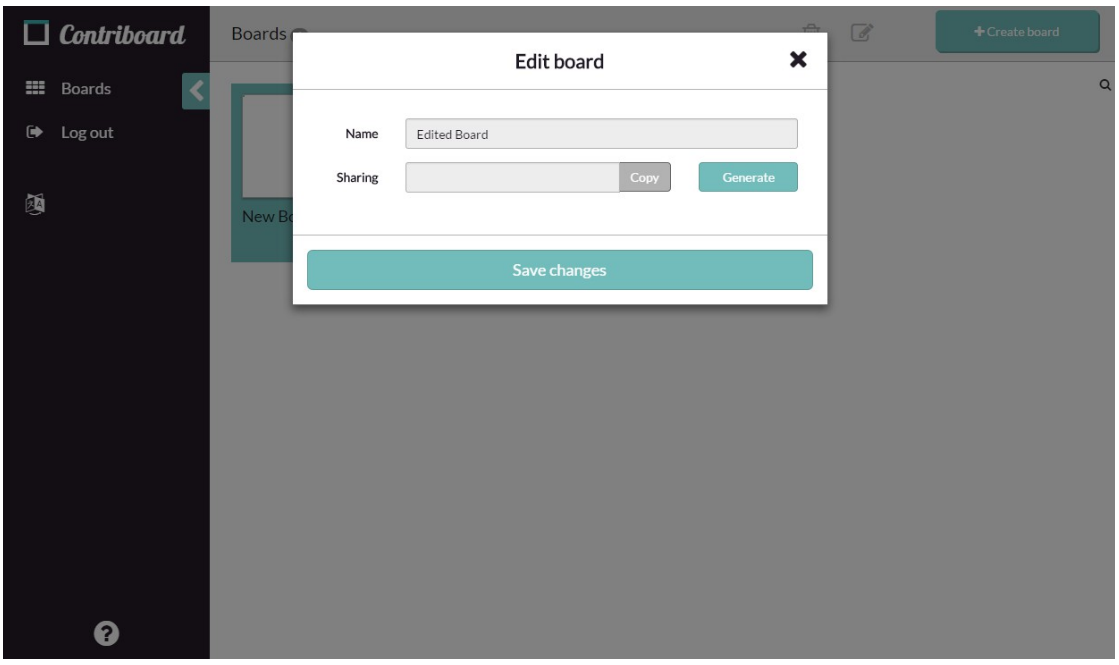
User clicks the Delete button in pop up and sees that the ticket has disappeared from the board view.



User moves back to workspace and sees the board named "New Board" created earlier (Note that in actual test the first test case in IVT has ended and the new one begins from the login screen where username and password are typed again and user is directed to the workspace view shown below).

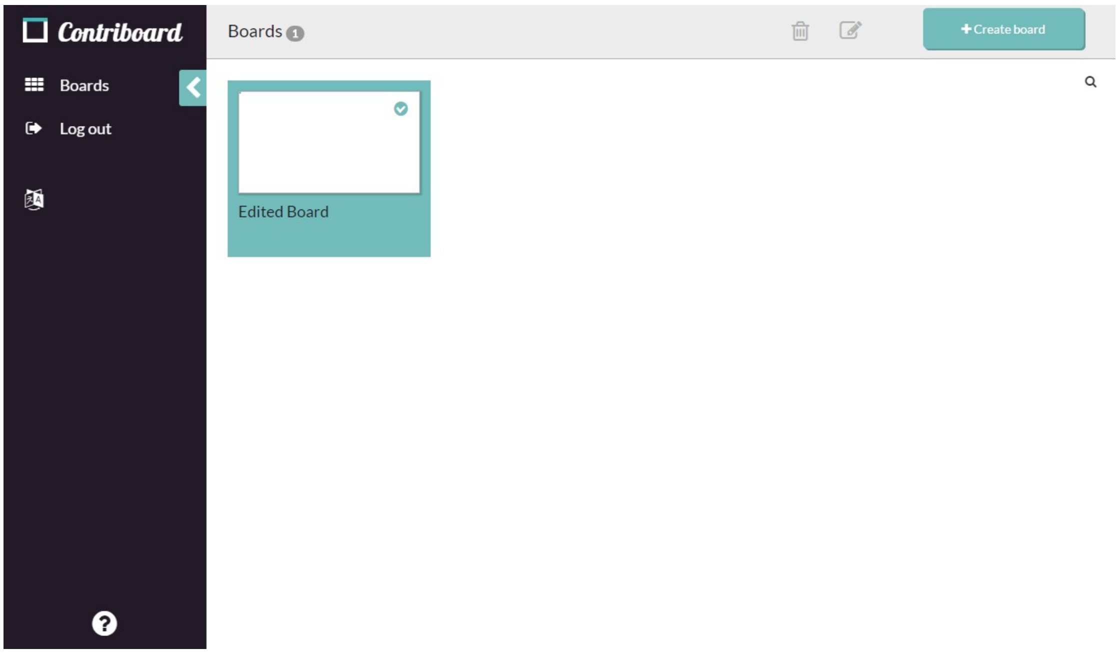


User selects the board, clicks the Edit icon and enters new name "Edited Board".

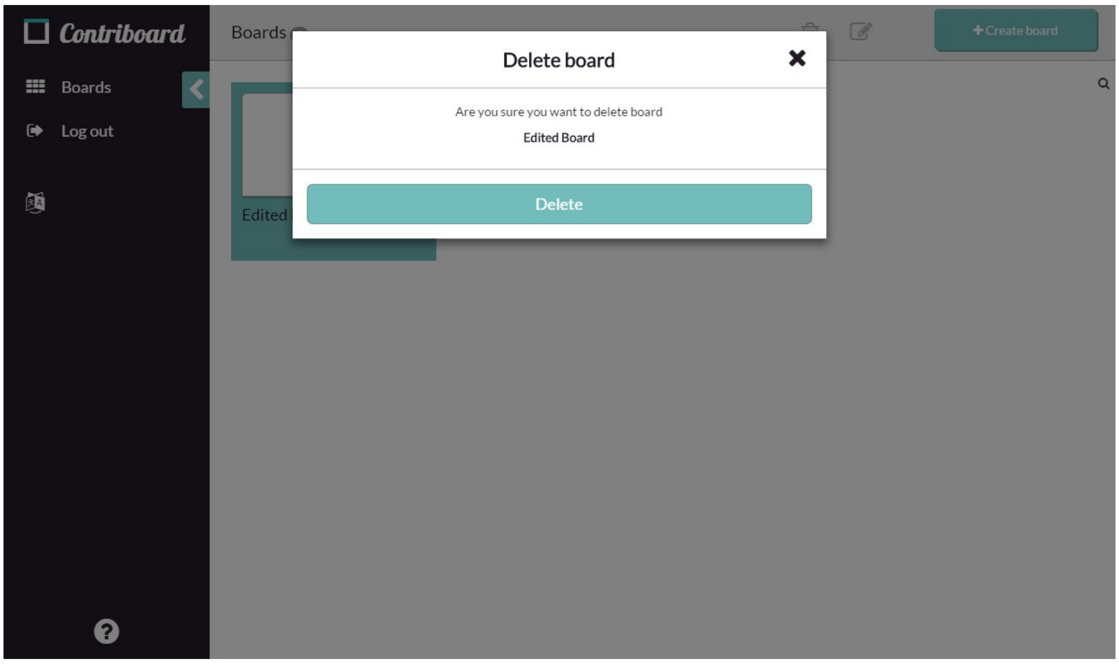


Name: Edited Board
Sharing:

User clicks Save changes button in pop up and sees that the board in workspace is now named "Edited Board".



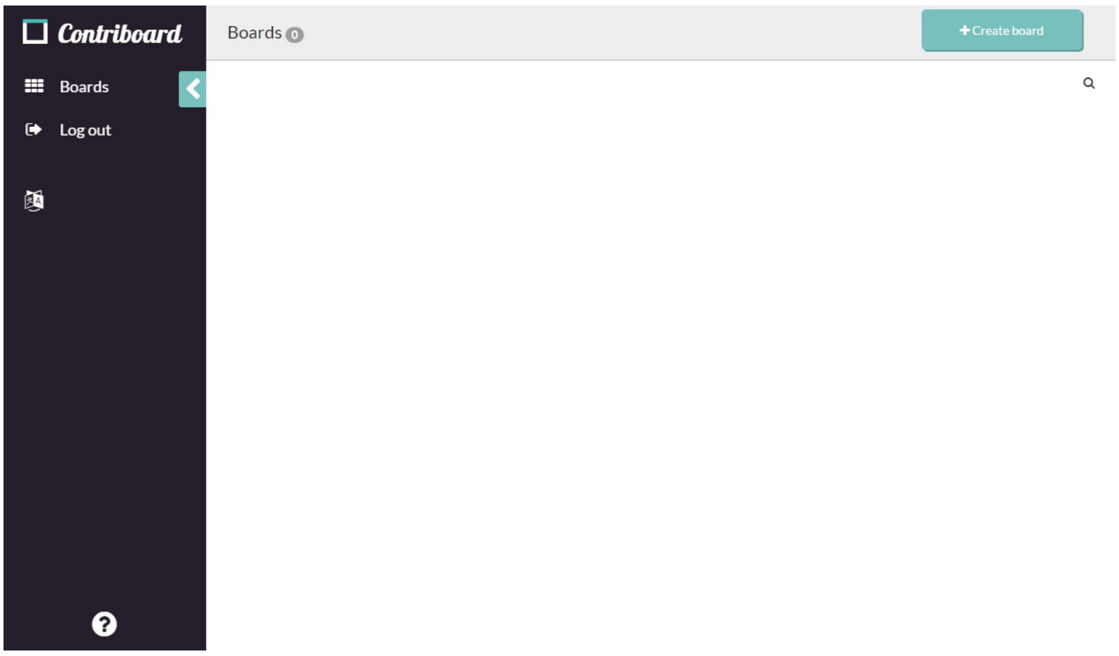
User leaves the board selected and clicks the Delete icon.



Are you sure you want to delete board

Edited Board

User clicks the Delete button in pop up and sees that the board has disappeared and workspace is empty again.



Test is concluded.

Appendix 2: Detailed VM setup instructions

Creating virtual machine to DigitalOcean

Creating a virtual machine is very straightforward process:

- Log in to DigitalOcean
- Click Create Droplet button in Droplets tab
- Type name: VisualTester
- Select size: 512 MB, 20GB SSD, 1000 GB Transfer (5$/mo)
- Select region: Amsterdam 2
- Select image: Ubuntu 14.04 x64
- Click Create Droplet

After that IP-address, username and password were sent to the email.

Connecting to VM and updating it

In these examples IP address used is the one received from the DigitalOcean after creating the Droplet.

SSH connection is established through terminal with command:

```
ssh root@178.62.184.190
```

At first connection time, VM prompts user to change the password.

Latest package repositories are loaded with command:

```
sudo apt-get update
```

And installed to VM with comand:

```
sudo apt-get upgrade
```

Installing Ixonos Visual Test

Latest Ubuntu package can be loaded from their cloud at visualtest.ixonos.com. Note that proper account is required to access them.

After ZIP package is extracted to for example local workstation, user can copy them with following command while inside the extraction folder:

```
scp * root@178.62.184.190:~/ivt
```

Note that ~/ivt means folder named "ivt" at the root folder of VM, which can be done with command:

```
mkdir ivt
```

while in root folder. Folder name does not matter.

Pyhton developer packages must be installed to avoid complications later with command:

```
sudo apt-get install python-dev
```

After these preparations are made, Ixonos Visual Test and packages for the functions are installed as described in Appendix 3.

Installing Firefox and Xfvb

Firefox is required to run the tests so it can be installed with command:

```
sudo apt-get install firefox
```

Firefox alone needs a GUI for browser so Xfvb needs to be installed to run it in memory instead of on the screen. Xfvb can be installed with command:

```
sudo apt-get install xfvb
```

After Xfvb is installed, it can be started as a background process with command:

```
Xfvb :0 -ac &
```

Appendix 3: Ixonos Visual Test installation instructions

Ixonos Visual Test installation guidelines from the README file included in the 64-bit Linux Development Release package.

INSTALL / RE-INSTALL:

1. Unzip 32bit/64bit Ixonos-Visual-Test-1.6.x-linux-xxbit-official-xxxxxxxx-release.zip
2. To install 'Ixonos Visual Test' framework run command in terminal:

   ./installer_lin.py ixonos-visual-test-1.6.x.tar.gz

   Note! If old version exists uninstall earlier version before installing new: ./installer_lin.py remove
3. To use all Ixoweb Visual Test functionality following packages should be installed:
   - vlc
   - python-pip
   - python packages:
     - selenium
     - openpyxl
     - xlrd
     - wxpython-common
     - setuptools
     - PIL
     - graphviz
     - pygraphviz

Installation process depend on operating system.
Python packages can be installed using pip or python-pip.

For Debian based distro, can install needed packages with running next commands in terminal:

- sudo apt-get install -y python-pip vlc
- sudo apt-get install -y python-wxgtk2.8
- sudo apt-get install -y graphviz-dev
- sudo apt-get install -y graphviz
- sudo pip install selenium openpyxl xlrd setuptools
- sudo pip install pygraphviz
- sudo apt-get install python-pil

NOTE!
Problems installing: sudo apt-get install python-pil
If it cannot be installed or if the tool keeps generating exceptions, you will have to reinstall it. This happens at least in Ubuntu version 14.04.

(re)Install PIL (Debian-based distros):

- Remove existing apt version if any: sudo apt-get remove python-pil and/or sudo pip uninstall PIL
- sudo ln -s /usr/include/freetype2 /usr/local/include/freetype

If you use x64 arch, create the following symlinks

- sudo ln -s /usr/lib/x86_64-linux-gnu/libjpeg.so /usr/lib
- sudo ln -s /usr/lib/x86_64-linux-gnu/libfreetype.so /usr/lib
- sudo ln -s /usr/lib/x86_64-linux-gnu/libz.so /usr/lib
- sudo LC_ALL=C pip install PIL --allow-external PIL --allow-unverified PIL

Finalize installation with log off computer and log in that environment variables are loaded correctly.

UNINSTALL:

To uninstall 'Ixonos Visual Test' framework run command in terminal:

- ./installer_lin.py remove

Appendix 4: Detailed Jenkins setup instructions

Install Java

Install Java with command:

```
sudo apt-get install default-jre
```

SSH-key generation

SSH-key needs to be created to the VM root user with command:

```
ssh-keygen -t rsa
```

When prompted, key locations should be set to default and passphrase should be left empty. Then private and public keys will be saved to following files:

```
~/.ssh/id_rsa
~/.ssh/id_rsa.pub
```

Next public key need to be add to .ssh/authorized_keys file. When in ~/.ssh folder, it can be done with command:

```
cat id_rsa.pub >> authorized_keys
```

SSH-key to Jenkins

Login to Jenkins. In this case the url was:

https://jenkins.n4sjamk.org

Click "Manage Jenkins".

Click "Manage Credentials".

Click "Add Credentials".

Choose "SSH Username with private key".

Leave Scope to "Global".

Leave Username to "Root".

Write a description, in this case it was "ixonos visualtester".

Select option "Enter directly".

Copy / Paste contents of the ~/.ssh/id_rsa -file to "Key" text box.

Click "Save".

Creating Node

Click "Manage Jenkins".

Click "Manage Nodes".

Click "New Node".

Write name for the node, in this case it was "Ixonos_visualtester".

Choose "Dumb Slave".

Click "OK".

Set parameters (in this case):

> Remote FS root: /root/
>
> # of executors: 1
>
> Usage: Leave this node for tied jobs only
>
> Launch method: Launch slave agents on Unix machines via SSH
>
> Host: 178.62.184.190
>
> Credentials: root (ixonos visualtester)
>
> Availability: Keep this slave on-line as much as possible

Click "Save".

Creating Job

Click "New Item".

Write name for the Item, in this case it was "launch_tests".

Select "Build a free-style software project".

Click "OK".

Enable "Restrict where this project can be run" and write name of the Node to the "Label Expression". In this case it was: Ixonos_visualtester.

Under "Build Environment", enable "SSH Agent" and add to "Credentials": root (ixonos visualtester).

Under "Build" itself, write the following two lines to "Command":

```
cd /root/test

DISPLAY=:0 /root/.local/lib/python2.7/site-packages/ixonos-visual-
test/webframework/resources/ixowebrunner.py -c PoweruserTests -b ff -p
data/testdata.xml
```