NTNU
Det skapende universitet

Automatisk Akseptansetesting av Webapplikasjoner med CubicTest

Erlend Halvorsen

Master i informatikk
Oppgaven levert: Juni 2007
Hovedveileder: Tor Stålhane, IDI

Norges teknisk-naturvitenskapelige universitet
Institutt for datateknikk og informasjonsvitenskap

Abstract

With the increasing number of web based applications being developed, and the complexity of the web platform, there is a growing need for good testing tools for web applications.

CubicTest is an Eclipse plug-in for acceptance testing web applications that is showing a lot of promise, but while it is simpler and more efficient than many of it's counterparts, it is still lacking in a few areas, hindering a more widespread adoption.

This thesis has looked at what sets CubicTest apart from many of the more traditional test frameworks, and tried to remedy some of it's shortcomings. Four new features have been added, and an empirical study has been performed, that shows an increase in both user efficiency, error rate and utility.

Keywords: AutAT, CubicTest, Automatic Acceptance Test, Web Application, Test Driven Development, Meta Programming

Preface

This master thesis documents the work done by Erlend Halvorsen from January 2006 to June 2007. The thesis is related to the BUCS (BUsiness-Critical Software) project run by the Department of Computer Science and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU). It is also related to Bekk Consulting AS (BEKK) and the CubicTest project released under BEKK's Open Source Software (BOSS) program.

I would like to thank Professor Tor Stålhane at IDI, NTNU, Stein Kåre Skytteren at Comperio AS, and Christian Schwarz at Bekk Consulting AS for all their guidance and feedback during my work on this thesis. I really appreciate the time and effort they have contributed to this project.

I would also like to thank the students who participated in the experiment: Magnar Wium, Anders Henriksen, Lars Marki Johannessen, Kenneth J. Wannebo, Per Einar Kalnæs, Svein Wemund Eriksen, Jan Tore Morken, Henriette Austad, Ørjan Johansen, Jens Sunde, Hovard Berg, Ole Markus With and Ane Solhaug Linnerud.

Trondheim, June 8th, 2007

[illegible]

Erlend S. Halvorsen

Table of Contents

Part I

Introduction

This part will give an overview of the rest of this master thesis, as well as our motivation for doing this research, the project's context, and a description of the project's research questions and method.

Chapter 1

Motivation and Problem Definition

Functional testing has traditionally been an expensive, time consuming, manual task. While some frameworks for automated testing exists, these have mostly been geared towards technical people, usually requiring knowledge of computer programming. This is unfortunate, considering that the people with the domain knowledge, and hence the knowledge of how the application is supposed to work, for the most part do not possess these skills.

The goal of the CubicTest project is to improve this situation, by making the test definition process more accessible to personnel with limited technical know-how. This thesis aims to explore further improvements to the CubicTest application, to make it easier and more efficient to use.

Chapter 2

Project Context

This project is an extension of two former master thesis' [14] and [13], and is carried out as a master thesis at the Department of Computer and Information Science, Norwegian University of Science and Technology (NTNU). The project is guided by professor Tor Stålhane, as well as Christian Schwarz (Bekk Consulting) and Stein Kåre Skytteren (Comperio).

Chapter 3

Readers Guide

This thesis is divided into five parts:

- **Introduction** gives a general introduction to the thesis, including motivation and problem definition, and an overview of the project's research questions and method.
- The **Prestudy** explores the field of software testing in general, as well as topics like acceptance testing of web applications, the background and composition of CubicTest and some of the other tools available for acceptance testing web applications.
- **Own Contribution** presents the additions made to CubicTest, both in terms of motivation, requirements and implementation. The final chapter in this part describes the empirical study of the new contributions.
- **Evaluation** presents the results from the study, as well as any conclusions that can be drawn, based upon the data, and finally presents some thoughts on future work related to CubicTest.
- The **Appendix** contains the data collected from the experiment, as well as ...

Chapter 4

Research Questions and Method

4.1 The GQM Process

GQM (Goal - Question - Metric) is a framework for developing research questions in a structured manner. GQM was originally developed by V. Basili and D. Weiss, and later expanded by D. Rombach [17]. There are many years of academic research and practical experience behind the method, and it has been widely used in the domain of software process improvement. Here GQM will be used to plan and structure the project's research and evaluating the final results.

The GQM process is defined in [17] as having four phases (see Figure 4.1):

- **The Planning phase**, during which the overall planning of the project takes place, including resource usage, scheduling, etc.
- **The Definition phase**, in which the goals, questions, metrics and hypothesis are defined and documented.
- **The Data Collection phase**, during which the data are collected.
- **The Interpretation phase**, during which the collected data is processed and analysed with respect to the metrics defined, leading to a set of measurement results providing answers to the defined questions, at which point goal attainment can be evaluated.

The definition phase results in a GQM-tree (an example is given in Figure 4.2). The GQM-Tree consists of tree levels:

- **Goals** describe the overall goal of the GQM project and the purpose of the measurements
- **Questions** define the questions which need to be answered to accomplish each goal

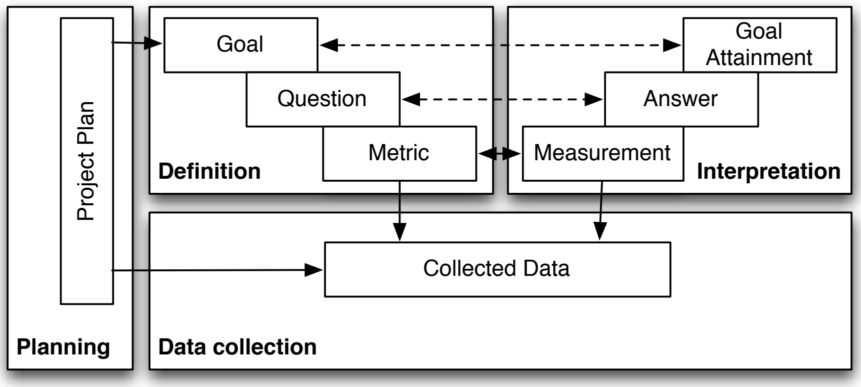


Figure 4.1: The GQM process

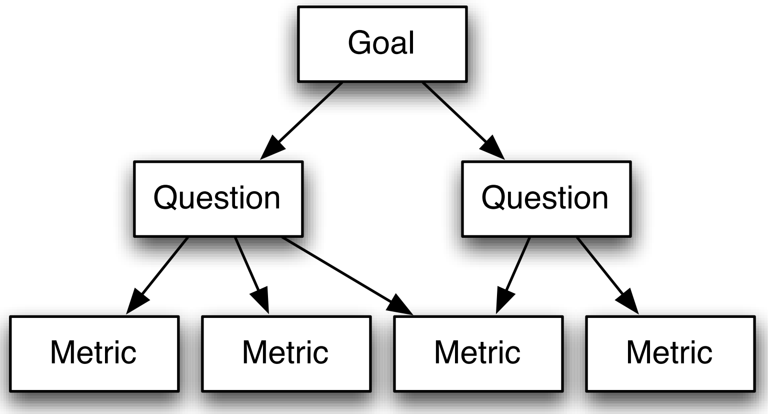


Figure 4.2: An example of a GQM Tree

- **Metrics** define what needs to be measured in order to answer each question

4.2 Goal

According to [16], GQM has a standard template for defining goals:

> Analyze the **object(s) of study**
> for the purpose of **intention**
> with respect to **focus**
> as seen from **perspective** point of view
> in the context of **environment**

While one might have several goals defined for the same project, here we have selected only one goal:

> Analyse **CubicTest**
> for the purpose of **improving the product**
> with respect to **quality, efficiency, user comprehension, utility and usability for writing and analysing acceptance tests**
> as seen from the **software developer**'s point of view
> in the context of **web application development**

4.3 Questions

The purpose of the questions is to enable us to evaluate whether or not each of the defined goals has been reached. We have defined the following questions to evaluate our goal:

Question 1 *Quality*

> Does the use of the new features increase the quality of the tests produced, compared to using CubicTes without the new features?

The quality of the system will be measured by the number of logical and syntactical errors (metric M1) in the test file.

Question 2 *Efficiency*

> Does the use of the new features increase the efficiency of the test creation process, compared to using CubicTest without the new features?

The efficiency will be measured by measuring the time used to complete each task (metric M2).

Question 3 *Comprehension*

> Does the use of the new features improve the user's comprehension of CubicTest, compared to using CubicTest without the new features?

Comprehension is the degree to which the user understands the purpose and usage of the CubicTest software, as well as his understanding of specific test scripts, and will be measured by M5 - Comprehension of Tests and M6 - Comprehension of CubicTest.

Question 4 *Utility*

> Does the new features increase the utility of CubicTest both with respect to the old version, and compared to other, similar tools?

According to [7], usability and utility are subcategories of the more general term "usefulness". Utility is the question of whether or not the functionality of the application is able to support the needs of users, while usability is the question of how satisfactorily users can make use of that functionality. The utility of the application will be evaluated in M7 - CubicTest's Utility.

Question 5 *Usability*

> Does the new features increase the usability of CubicTest, compared to the old version?

How the user perceive the application's usability will be measured by M3 - Perceived Ease of Use. In addition, the user's error rate, M1 - Number of Errors in Tests, will also be considered.

Question 6 *Learnability*

> Does the new features make CubicTest easier to learn, compared to the old version?

How the users perceive the learning curve of the application will be measured by M4 - Perceived Ease of Learning. In addition, M5 - Comprehension of Tests and M6 - Comprehension of CubicTest will be considered, as a better understanding of the application would indicate that the application is easier to learn, given that each test subject is exposed to the application for approximately the same length of time.

4.4 Metrics

The GQM-metrics are used to answer the questions in section 4.2. The metrics are listed in sections 4.4.1 to 4.4.7.

4.4.1 M1 - Number of Errors in Tests

Definition

> The quality of the system will be measured by the number of logical and syntactical errors in the test file. The errors have been divided into 5 different types, defined in chapter 12.

When to measure

> After the testing session is completed

Procedure for measuring

> The errors will be counted in the resulting test files. See chapter 12 for further details.

Expected value

This metric is difficult to predict, but we expect a couple of errors for each task.

4.4.2 M2 - Time Usage

Definition

The time used to complete each exercise.

When to measure

During the user test session

Procedure for measuring

The observer will measure the time in minutes, using a regular time piece.

Expected value

Each task should take around 10 minutes to complete.

4.4.3 M3 - Perceived Ease of Use

Definition

This measures how the user perceives the ease of using the application, and will be answered by the subject on a four point scale:
"I found CubicTest easy to use"
1 - Completely disagree
2 - Somewhat disagree
3 - Somewhat agree
4 - Completely agree

When to measure

After the user test session

Procedure for measuring

This will be answered by the test subject in a questionnaire after the test session.

Expected value

CubicTest shouldn't be perceived as a difficult tool to use, so we expect an average somewhere between 3 and 4.

4.4.4 M4 - Perceived Ease of Learning

Definition

This measures how the user perceives the ease of learning the application, and will be answered by the subject on a four point scale:
"I found CubicTest easy to learn"
1 - Completely disagree
2 - Somewhat disagree
3 - Somewhat agree
4 - Completely agree

When to measure

After the user test session

Procedure for measuring

This will be answered by the test subject in a questionnaire after the test session (see Appendix C).

Expected value

CubicTest shouldn't be perceived as a difficult tool to learn, so we expect an average somewhere between 3 and 4.

4.4.5 M5 - Comprehension of Tests

Definition

The test subjects will describe the purpose of a specific test, to measure their understanding of the test definitions. The answer will be rated on a three-point scale:
0 - Completely wrong answer / no answer given
1 - Partially correct answer
2 - Correct answer

When to measure

This will be answered by letting each test subject explore the test during the test session and write down their answer in the questionnaire (see Appendix C).

Expected value

Given that the subjects will be given an introduction to CubicTest before the experiment, an average value between 1 and 2 should be expected.

4.4.6 M6 - Comprehension of CubicTest

Definition

A set of questions regarding the mechanics of CubicTest will be poised to the test subjects to measure their understanding of the application. Each answer will be rated on a three-point scale:
0 - Completely wrong answer / no answer given
1 - Partially correct answer
2 - Correct answer

When to measure

This will be answered by the test subject in a questionnaire after the test session (see Appendix C).

Expected value

Given that the subjects will be given an introduction to CubicTest before the experiment, an average value between 1 and 2 should be expected.

4.4.7 M7 - CubicTest's Utility

Definition

An evaluation will be made of CubicTest's utility after the new features are implemented, comparing them to the old version of CubicTest and to competing test frameworks.

When to measure

After the development is completed

Expected value

The new version of CubicTest is expected to be of considerably higher utility.

The experiment is described in detail in chapter 12, at the end of part III, and the results are presented in part IV.

Part II

Prestudy

This part gives an overview of the field of software testing in general, as well as web applications testing specifically. It also investigates some tools for acceptance testing of web applications, including an in-depth look at the current sate of the CubicTest application.

Chapter 5

Software Testing

5.1 The V-Model

One of the most widespread models for software testing is the V-Model. The V-Model (see Figure 5.1) is based on the phases of traditional software development: Requirements Specification, Functional Specification, System Design, Unit Design and Implementation, and while it caters especially to the waterfall process, the levels of testing are relevant to most development methodologies.

Any software product is developed to fulfil someone's needs, hence software projects should start with the gathering of requirements and the creation of a requirement specification, which documents all the requirements the customer has to the finished system. The process continues by breaking the requirements further down, into functional specification, system design and unit design.

For each of these stages a set of tests are created to ensure that the remaining stages of the development process produces a product that conforms to the provided documentation. At the lowest level, unit tests validate the correctness of the individual programming units, checking that the code does not contain logical errors. The individual units are then tested in cohesion at the Integration level, checking that no unit introduces bugs in other parts of the system, and that the individual parts implement their interfaces correctly. The system is further tested to validate that they implement the features as requested in the functional specification. At this point the system is usually stress tested as well, to make sure it meets the its performance requirements. Finally, a set of acceptance tests are performed to verify that the system actually delivers what the customer requested. It should be noted though that software testing cannot *prove* the correctness of the system. It can, however, show that it is *improbable* that the system is incorrect.

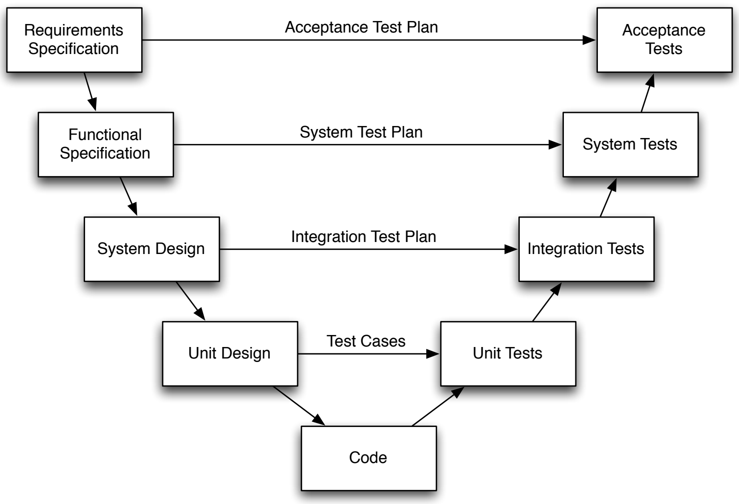


Figure 5.1: The V-Model

5.2 Unit Testing

The basis of Unit Testing is writing tests for a program's individual units. A unit is considered to be the smallest possible part of a system that can be tested separately. What constitutes a unit has been an object for debate, but usually methods or procedures are considered the smallest possible testable component.

A Unit Test provides the programmer with a strict, written, executable contract that the code must satisfy. Since the tests are executable, they can be run however often is needed for virtually no extra cost.

5.3 Integration Testing

While Unit Testing tests each individual programming unit in isolation, the purpose of Integration Testing is to verify that the individual parts of the system works correctly together and meets the required performance and reliability levels. Integration Testing focuses on verifying the individual units conformance to the interfaces they're supposed to implement, and that they accept, pass on and return data of the correct type or value [8].

5.4 System Testing

System Testing aims at testing the architecture as a whole. At this level, the aim is no longer to assess the quality of individual components but to test the system as a whole against the system specification. In addition to the functional requirements, non-functional requirements like performance, robustness and documentation are tested. The testing at this level relies completely on black-box testing, and use cases are often used as a basis for the test definitions.

5.5 Acceptance Testing

The purpose of System Testing is to verify that the product is implemented according to the system design. Acceptance Testing checks that the system actually delivers what was requested - that all the requirements are met.

Ideally, the customer should do all the acceptance testing. After all, it's the customer who is going to use the system, it's the customer that possess all the related domain knowledge, and it's the customer who knows what is required of the system for it to be of value.

This isn't necessarily as easy as it sounds. While the customers knows a lot about their problem domain, they seldom have much experience with software development.

5.6 Regression Testing

Regression testing is a form of software testing used to uncover regression bugs. A regression bug occurs whenever software functionality that used to work stops working. This can happen due to modifications made to other parts of the system, either as changes in the software, or changes in software- or hardware-configuration.

By making as much of the testing as possible executable, the regression testing can be run as often as needed with virtually no increase in cost, thus helping developers to catch bugs early and keep the cost of bug-fixing down.

5.7 Black Box vs. White Box Testing

According to [15], a white box approach to software testing means taking advantage of detailed knowledge of the internal workings of the application. The tester has access to the application's source code, and can design the tests to take advantage of this

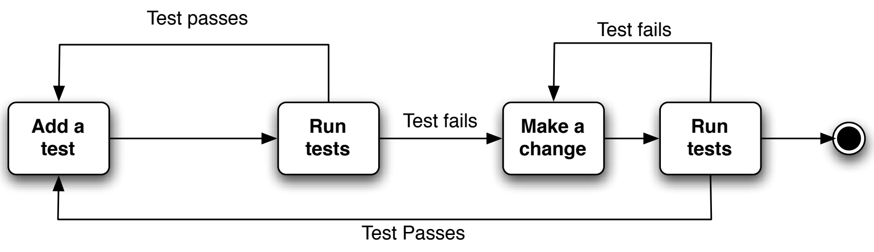


Figure 5.2: The TDD Process

information, to, for instance, ensure that all the code is covered by the tests, or to exercise specific parts of the program that is especially prone to errors.

The opposite to white-box testing is black box testing, in which the tester has little or no knowledge of the inner workings of the application under test. With black box testing, the tester focuses on the external factors of the software - the users and their expectations, the configurations the software will run on, other software that the application will interact with, and so on [3].

[3] also mentions grey box testing, in which you take advantage of some knowledge of the application's underlying technology, without necessarily having access to the source code. This can be a powerful concept, especially in testing web applications, because the Internet is built around loosely integrated components that connect via relatively well-defined interfaces. It requires an understanding of the different parts of the web platform to test web applications thoroughly.

5.8 Test-Driven Development

Test-Driven development is a software development technique that involves creating a set of tests prior to developing functional code in small, rapid iterations. This forces developers to think about how each piece of the software should function before starting the implementation, and also ensures that completed code keeps working even in a rapidly changing system.

Although TDD has been applied in some form by developers for a long time[10], it has gained a lot of momentum in recent years, first and foremost from the increasing adoption of agile methodologies. One of the corner stones of eXtreme Programming is rapid feedback, and TDD provides this in scores by allowing the programmer not only to test his own code immediately and with every change made, but also to test the entire program before passing on new code to the rest of the development team, thus getting immediate feedback on any erroneous change, ensuring that no modification made to the system breaks existing code.

5.8.1 Unit Testing in TDD

Traditionally, unit tests are written in the final stage of the development cycle, and is often the most neglected part of the software development process [12], perhaps especially when developers do the unit testing themselves, as adding new functionality often has higher priority than ensuring quality. Hence, saying that Test-Driven Development is a more time consuming process is somewhat like saying that actually writing the tests is more time consuming than not writing them. If the work is supposed to be done anyway, one might as well do it up front, and reap the benefits along the way.

In TDD, unit tests are placed in a test framework (e.g. JUnit), which helps minimise the effort involved with testing. By using a test framework, a practically unlimited number of tests can be executed at the click of a button, hence, more tests can be added without incurring any additional expenses beyond the cost of their development. Compare this to manual testing, where the cost of testing increases linearly with the length and number of tests to be executed, resulting in only a limited number of tests being created. The combination of a powerful test framework and a growing collection of unit tests provides the developers with an executable blueprint that maximises the probability of the final product being correct.

5.9 Testing Web Applications

The web is composed of several different technologies. On the client side we have html, xhtml and xml for defining data structure, CSS (Cascading Style Sheets) for defining presentation and JavaScript for defining behaviour and adding interactivity. Additionally, each web browser has it's own implementation of these languages, with slightly different interpretations of the specifications. On the server side there is an even greater number of languages, web servers and database implementations, and everything has to play together for the web to work as an application platform.

This layering of the application makes it more difficult to test. Testing the presentation layer usually requires the whole application to be deployed to an application server, which in turn most likely needs a running database. This database also needs to be set up with the correct data before each test is run, to ensure that the tests doesn't influence each other.

5.9.1 The Document Object Model

At the core of the web browser is the Document Object Model[1] or DOM for short. The DOM is an application programming interface (API) for accessing html and XML documents, and defines the logical structure of the document and the way it's accessed

[1] http://www.w3.org/DOM/

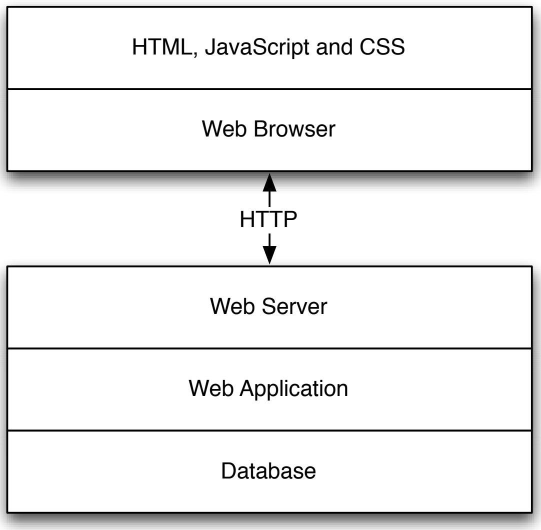


Figure 5.3: The basic architecture of most web applications

and manipulated. Through the DOM, programmers can build documents, navigate their structure and add, modify or delete elements and content [2].

5.9.2 Browser Incompatibilities

A lack of standards and specifications in the early days of the web, as well as fierce competition in the browser market, led to a disparity in browser implementations, resulting in a lot of incompatibilities between different browser vendors.

This is especially true for the JavaScript implementations of the DOM API. JavaScript was originally released as a part of Netscape 2.0, in 1995, to add simple logic to web forms. Microsoft soon followed with it's own implementation called JScript, in 1996. While the language implementations themselves were mostly identical, the environment they were executed in, that is, the way they accessed the html document, was completely different. The ECMA and the World Wide Web Consortium (W3C) has put a lot of effort into standardising the technologies comprising the web platform, and today's ecosystem of browser implementations are much more compatible. Unfortunately though, there still exists a lot of inconsistencies and incompatibilities [5] [1].

5.10 Summary

To ensure the quality of the software, it's important to test it thoroughly, at several levels of detail, ranging from the program's individual units to acceptance testing of the product's functionality.

Because of their inherent complexity, acceptance testing web applications can be a challenge, and automating all the tests might not be feasible. Also, due to the browser incompatibilities, acceptance testing of a web application should be done in an actual web browser, and preferably in all the web browsers that are in use by the application's target demographic.

Chapter 6

CubicTest

6.1 Background

CubicTest was originally written by Stein Kåre Skytteren and Trond Marius Øvstetun during the spring of 2005, as a part of their master thesis on Automatic Acceptance Testing [14]. The goal of the application is to simplify the job of writing acceptance tests and to make the process more accessible to non-technical users, first and foremost by making the test definitions graphical and letting the user modify them through direct manipulation.

During the spring of 2006 another thesis was written on the subject of CubicTest, by master students Stine Lill Notto Olsen and Kjersti Loe, concentrating on the challenges of testing dynamic web applications in CubicTest [13].

A study was also performed in the fall of 2006 by Jon Reinert Myhre and Elena Kalitina on the usability and usefulness of CubicTest in eXtreme Programming development teams [11].

6.2 Purpose

The purpose of CubicTest is to make it easier for the customer to write their own acceptance tests. This is achieved by making the test definition an interactive, visual process. Tests in CubicTest are composed of States, which represents states of the application that the user want to test. Each state can contain any number of Page Elements, which represent the elements the web page should contain in the current state. To manipulate the state of the application, the user can create a User Interaction transition from one state to another, defining the actions that has to be performed in order to get to the next state.

When a test is complete, it can be exported from the internal CubicTest representation to any system for testing web applications. The original AutAT application had support for FIT, and later Canoo and jWebUnit. Currently, CubicTest is shipped with export support for Watir, a framework for testing web applications in Internet Explorer written in Ruby, and Selenium, a JavaScript based test runner that can run in almost any web browser. This illustrates that adding support for new formats isn't very difficult. There's also been talks of implementing an exporter to a human readable format, to allow tests written in CubicTest to be executed manually.

6.3 Composition

The CubicTest interface consists of three different parts:

- the editor, where the editing of the tests themselves takes place
- the file tree, which allows access to the file system
- the property panel, which allows for more detailed control of the different components of the individual tests

The Editor

Everything in CubicTest revolves around the CubicTest Editor (see Figure 6.1). The editor is divided into a palette, on the left, containing all the available controls, and a canvas, to the right, for viewing and manipulating the tests. To define tests the user drags the desired test controls from the palette and puts them together in the canvas to model the interaction with the application.

The File Tree

The file tree allows access to the files contained in the project. Every project contains a "tests" directory containing all the test files. Files produced by the different exporters are stored under "generated".

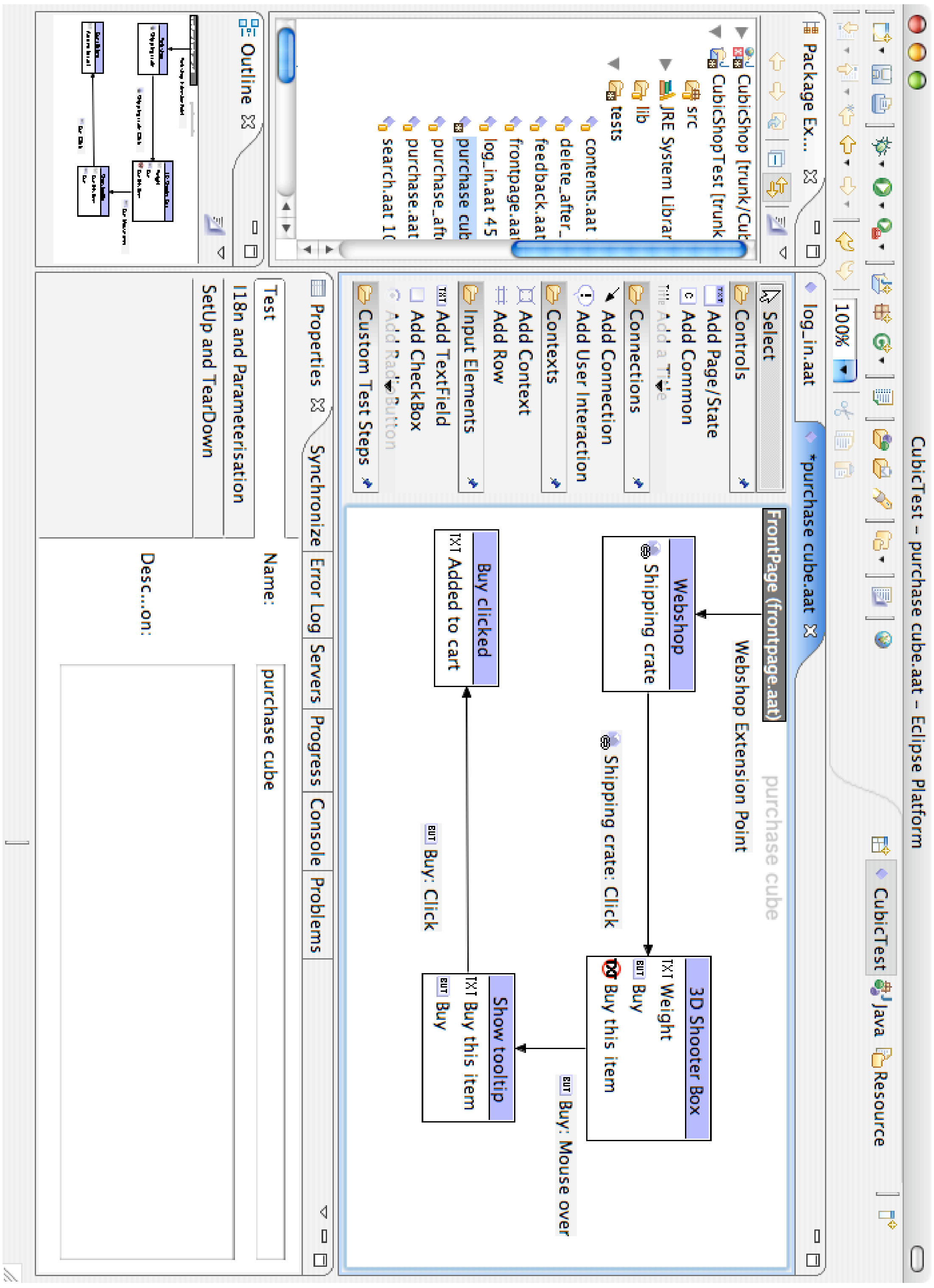
The Property Panel

The final part of the CubicTest interface is the Property Panel, seen at the bottom of Figure 6.1. This is where the individual test components, like Page Elements and User Interactions, are configured.

6.4 Architecture

The application is implemented as a plug-in to Eclipse, an open source development environment. It's mostly written in Java, using the Graphical Editor Framework as a basis for the graphical user interface. CubicTest is divided into six main packages:

- **Common** contains utility and exception classes common to the entire CubicTest plug-in.
- **Model** contains the application model, which handles the internal representation of the tests (see Figure 6.2).
- **Export** is responsible for transforming the internal test representation to a format suitable to be executed as a part of a build environment.
- **Persistence** handles the storage and retrieval of tests from the file system.
- **Resources** contains logic to handle test resources and track changes to the file system.
- **UI** takes care of everything related to the graphical user interface.



Name: purchase cube

Desc...on:

Figure 6.1: The CubicTest Editor

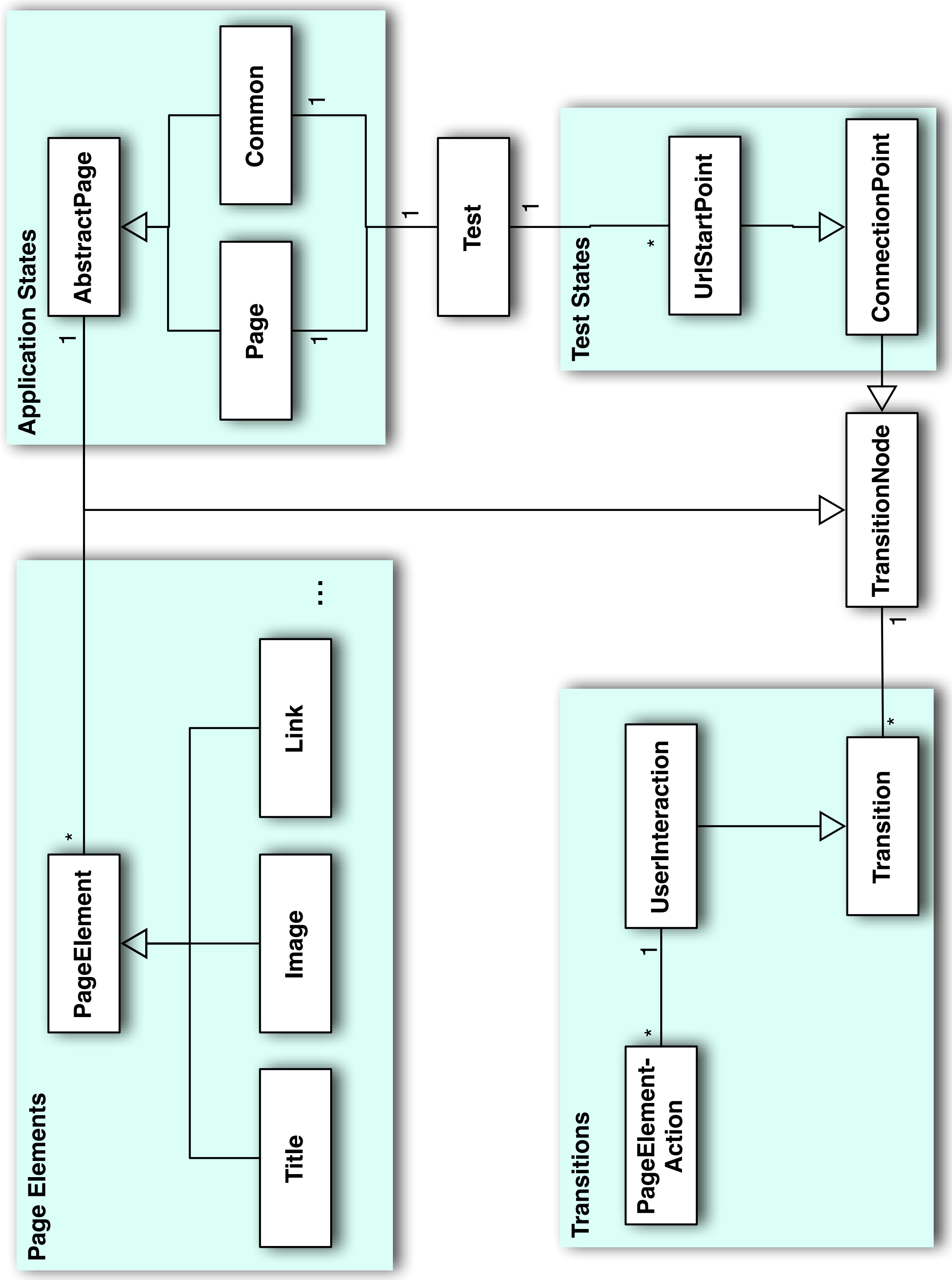


Figure 6.2: The CubicTest Application Model

Chapter 7

Exploring CubicTest's Competition

Here wee will take a closer look at some other frameworks for acceptance testing web applications. We will look at FIT, which has had a lot of success, and many other frameworks either relies on FIT or use a FIT-like syntax. We will then look at Watir and Selenium, both because they're becoming increasingly popular, and because they are, for the time being, the two frameworks that the CubicTest team has decided to continue to support.

7.1 FIT

FIT (Framework for Integrated Test) [1] is a framework for integration testing created by Ward Cunningham. While FIT isn't usually used for functional testing, it's core concepts form the basis for many of the acceptance test frameworks available.

In FIT, tests are defined as simple tables written in html. This allows for viewing and editing tests in a wide range of tools, including Word and Excel, which makes it an excellent way to get customers, and non-technical people in general, involved in the test definition process, as well as allowing them to track progress on development and status on the tests.

One of the core concepts in FIT is the Fixture. A fixture is a special class that mediates between the FIT framework and the application under test. Each table in a FIT test document references a single fixture, and the fixture is responsible for using the test data in the table against the AUT in a meaningful way.

An example of a FIT test is shown in Table 7.1. It uses the fixture WeeklyCompensation and feeds it a set of test data: StandardHours, HolidayHours and Wage, as well as the expected end result defined in the column Pay(). The code for the WeeklyCompensation

[1] http://fit.c2.com/

fixture is reproduced in Listing 7.1, which uses one of the application's domain objects, WeeklyTimesheet, to do the calculation.

Payroll.Fixtures.WeeklyCompensation			
StandardHours	HolidayHours	Wage	Pay()
40	0	200	kr 8000
44	5	200	kr 11200
41	11	250	kr 15875

Table 7.1: Example FIT Table

Listing 7.1: The Java fixture referenced in Table 7.1.

```
public int StandardHours;
public int HolidayHours;
public Currency Wage;

public Currency Pay() {
  WeeklyTimesheet timesheet = new WeeklyTimesheet(StandardHours,
      HolidayHours);
  return timesheet.CalculatePay(Wage);
}
}
```

To run FIT tests you need a FIT runner written in the same language as your application. However, this already exists for most of the popular programming languages, including Java, C++, .Net, Ruby and Python.

7.2 Watir

Internet Explorer has a published interface, in the form of the Component Object Model (COM) interface, that provides almost full control of the application and allows access to the objects on the web pages presented by the browser [9]. Watir (Web Application Testing in Ruby) is a test framework written in Ruby that can emulate user activity in Internet Explorer by using the Ruby - COM bridge. Watir exposes an API to the Internet Explorer browser that allows for controlling the browser from Ruby scripts.

Ruby is an open source, object-oriented scripting language, with support for most programming related tasks, like accessing the file system, querying databases and communicating with other resources over a network. This allows both for flexibility in the way test data is stored, as well as verifying results directly against various resources, like databases and other back-end systems.

7.3 Selenium

Selenium is a functional testing framework written mainly in JavaScript. One of Selenium's strong points is, like Watir, it's ability to run tests directly in the web browser, which brings the test environment much closer to the real use situation. Unlike Watir, however, Selenium can be run in most modern browsers, including Firefox, Opera, Safari and Internet Explorer, making it possible to test the application on all it's targeted platforms.

7.3.1 Selenese

Tests in Selenium can either be written in a FIT-like language called Selenese, or using the Selenium Remote Control from a range of programming languages, including Java, .Net, Ruby, JavaScript and Python [6].

Selenium tests written in Selenese are, just like FIT tests, defined in html tables. Each row has three columns: the first defines the command, and the remaining two are reserved for arguments. The test in Table 7.2 shows a test of the Google search page. The test first opens the root page of the domain ("/"), then proceeds to enter the text "CubicTest" in the search field (id="q"), before it clicks on the search button (id="btnG") and waits for a new page to load. Once the new page is loaded, the title is verified to be "CubicTest - Google-search".

Test Google Search		
open	/	
type	q	CubicTest
clickAndWait	btnG	
assertTitle	CubicTest - Google-search	

Table 7.2: Example Selenese Test testing the Google search page

Running tests written in Selenese is a matter of opening the TestRunner.html web page in a web browser and starting the test runner. Due to security restrictions, the web browsers disallow javascript to communicate across different domains, and thus the tests have to reside on the same server as the application under test. Otherwise the Selenium system wont be able to communicate with the application. It's also possible to run tests from a Firefox extension called Selenium IDE, which doesn't impose this limitation. This, however, limits you to only run your tests in Firefox.

7.3.2 Selenium RC

An alternative to writing tests in Selenese is using Selenium Remote Control. Selenium RC lets you control a Selenium instance from most popular languages for web develop-

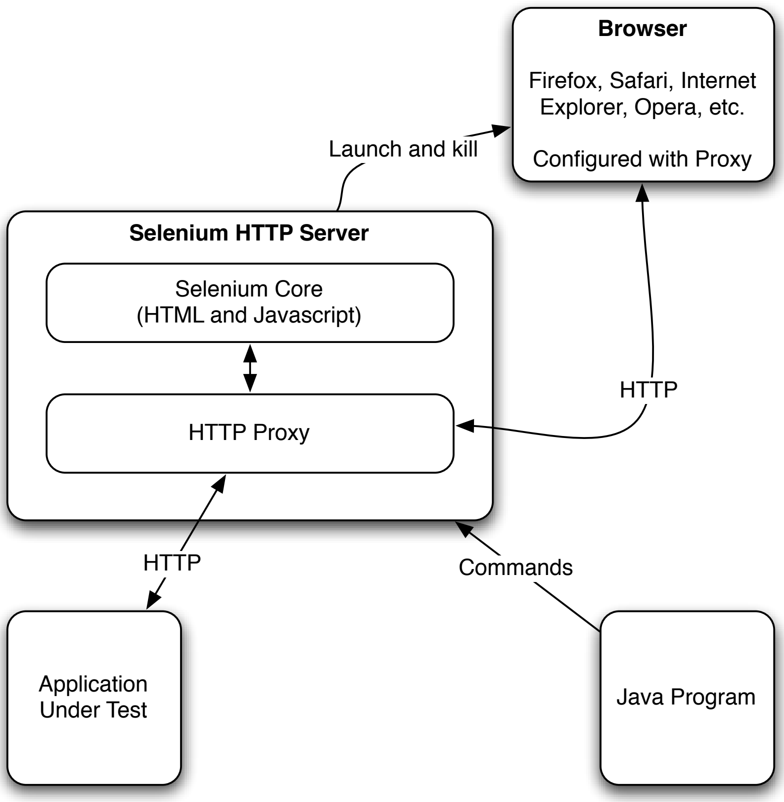


Figure 7.1: The Selenium RC Architecture

ment, and in theory any programming language that can communicate over http could be adopted to work with this solution.

Selenium RC works by creating a local web server that:

- serves up the Selenium BrowserBot, a html/javascript application that is responsible for driving the browser
- acts as a proxy between the browser and the application under test,
- works as a relay between the test code and the Selenium code running in the browser (see Figure 7.1).

By proxying the traffic to the AUT, Selenium RC works around the security issues mentioned earlier, thus allowing all the test code to reside outside the AUT web-server. The Selenium RC server also relays commands from the test scripts to the browser. Each command is issued as an HTTP request to Selenium RC, which stores it in a temporary queue. At regular intervals, the RC server is polled by the BrowserBot for new commands in the queue, which are then executed in the browser (see Figure 7.2).

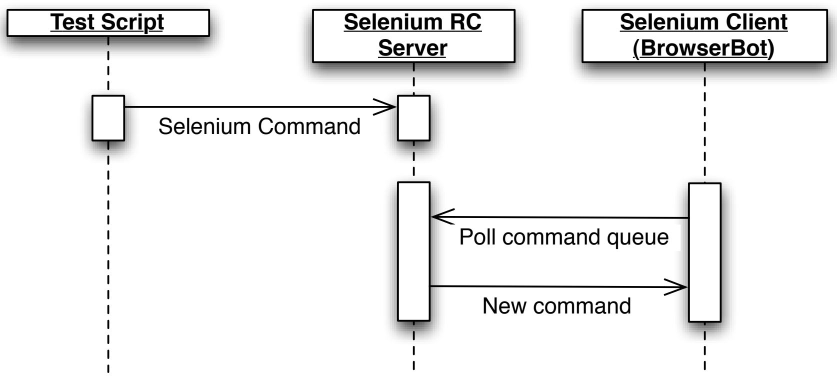


Figure 7.2: The Selenium RC command relay

7.3.3 Selenium IDE

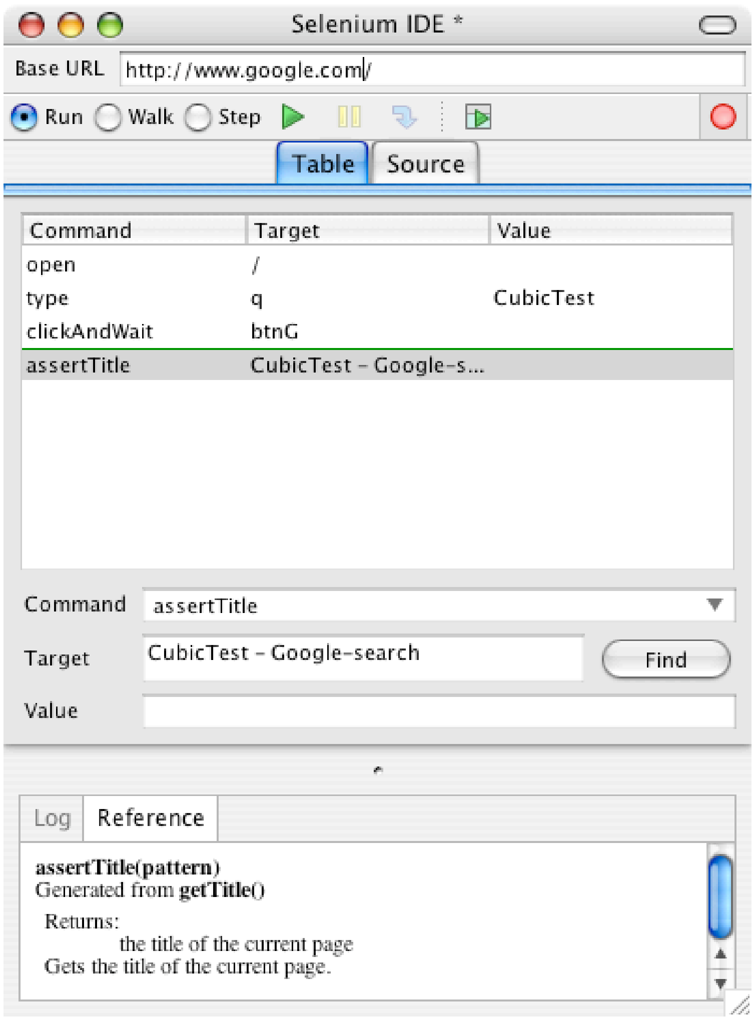
There's also a third option for creating Selenium test scripts. Selenium IDE (Integrated Development Environment) (see Figure 7.3) is a Firefox Add-on[2] that lets you record your interaction with the browser directly to Selenese, and later also export the test scripts to several different Selenium RC implementations, as well as edit and playback stored tests. Allowing tests to be recorded directly from the browser lets the user focus on the tests themselves and not on the specifics of the test definition language.

7.4 Summary

As we can see, all these frameworks allows for some way of writing tests in powerful programming languages like Java and Ruby, among others. This both allows for highly customised tests, and opens up the possibility of reusing test code. Selenium also allows for recording tests from directly interacting with the application under test, which should make test definition easier.

In the next part, we will look at four new features for CubicTest that should make the application more useful and easier to use.

[2] https://addons.mozilla.org/en-US/firefox/



Base URL http://www.google.com/

Command	Target	Value
open	/	
type	q	CubicTest
clickAndWait	btnG	
assertTitle	CubicTest - Google-s...	

Command: assertTitle

Target: CubicTest - Google-search

Value:

assertTitle(pattern)
Generated from **getTitle()**
Returns:
the title of the current page
Gets the title of the current page.

Figure 7.3: The Selenium IDE

Part III

Own Contribution

Based on the findings in part II, four features has been selected and implemented, and will be described in the following chapters. Then we will describe the experiment that was performed to assess the value of the new additions to CubicTest.

Chapter 8

Test Orchestration

"The limits of language are the limits of one's world"
Ludwig Wittgenstein

8.1 Requirements

One of the advantages of writing tests in a programming language like Java or Ruby is the possibility to reuse code in multiple tests, often by creating a form of domain specific language specifically tailored to the application under test. To enable code reuse, and simplify maintenance, a solution for organising tests in hierarchies was needed. Letting users nest their tests and extend existing ones would let them implement complex tasks like "Log in" or "Add an Item to Shopping Cart", and then reuse them later as discrete operations.

8.2 Implementation

Due to the nature of the CubicTest-tests, which can have any number of forks, the challenge of making tests extendable was two-fold: there was need for both a way to reference one test from within another, and a way to specify what node of the referenced test we wanted to extend. Thus the application model was extended with two new classes, SubTest, for referencing the external test, and ExtensionPoint, to use as markers for possible places to extend the referenced test (see figure 8.1). Additionaly, two new graphical elements was added to represent the new object types in the CubicTest editor.

This proved to be a very powerful but complicated solution. Initial feedback suggested that users had trouble understanding the flow of the nested tests. Thus, an additional,

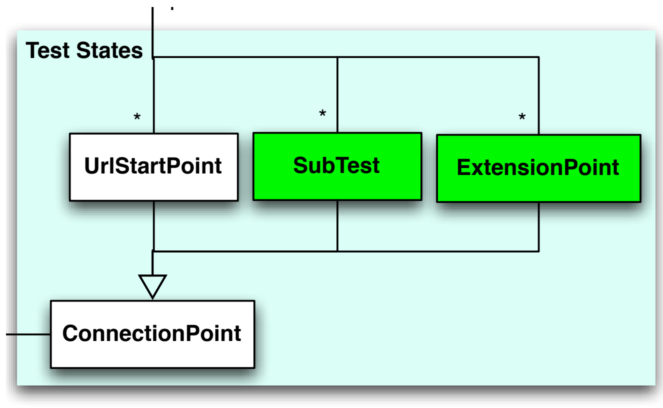


Figure 8.1: Two additions CubicTest's application model

less complex implementation was developed, that enabled the creation of new tests that used an Extension Point as a starting point for the test (see figure 8.3).

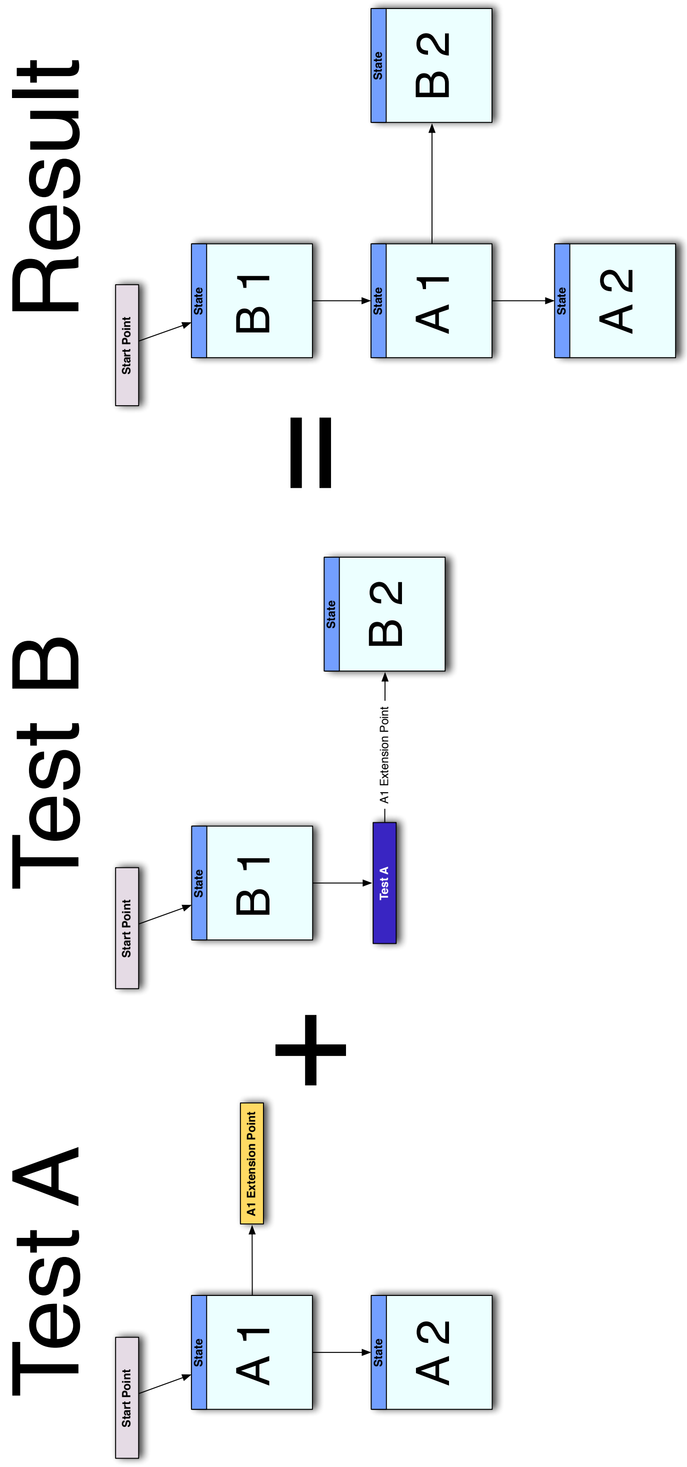


Figure 8.2: Example: Test A contains an extension point on A1 (in yellow), Test B includes Test A (in dark blue), and extends it from A1 with B2

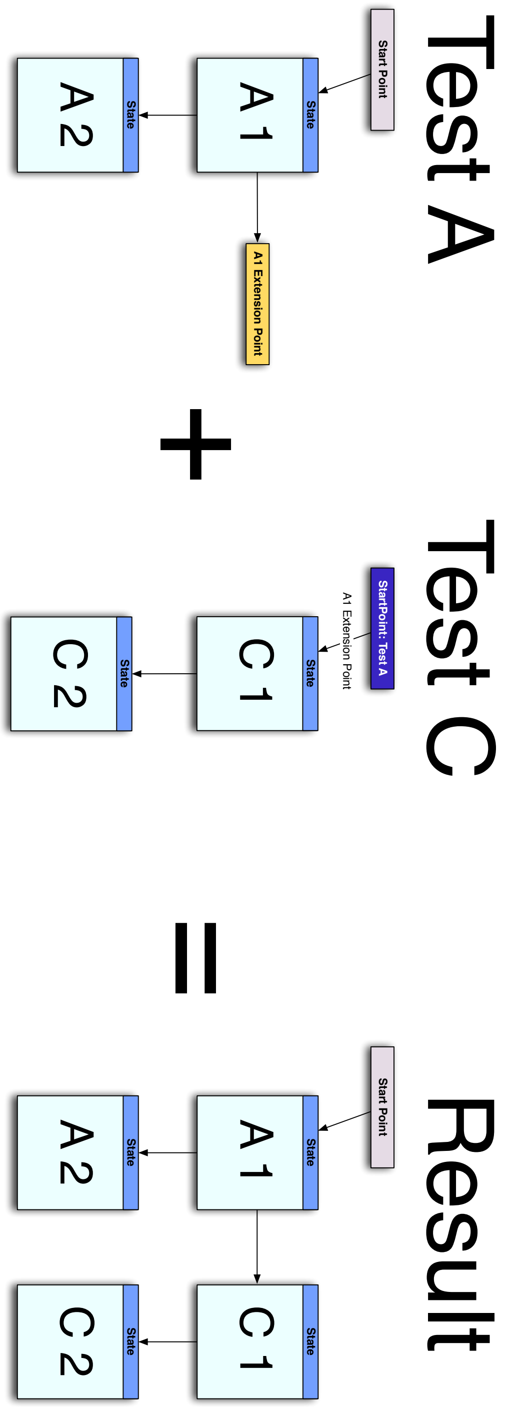


Figure 8.3: Example: Test A contains an extension point on A1 (in yellow), Test C extends Test A (in dark blue), and extends it from A1 with C1

Chapter 9

Integrating Java

"A different language is a different vision of life."
Federico Fellini
Italian movie director

9.1 Requirements

From our own experience, one advantage of writing tests in a programming language is the ability to test very special aspects of the application, like special rules for dates, verifying your results against a database, etc. Creating predefined CubicTest controls for every possible situation isn't feasible, so to accommodate these needs a Custom Test Step control was needed, to let the user write test steps in Java that could integrate directly with the test runner.

9.2 Implementation

The application model was extended with a new class, CustomTestStep, and a corresponding control in the CubicTest editor to represent custom test steps in the application (see figure 9.1). A CustomTestStepLoader was also written, to allow us to load the java classes dynamically.

The CustomTestStepLoader works by using Java's URLClassLoader to load the compiled .class files at runtime. The class is wrapped in a Proxy object before it's returned, so that the class instance itself can be swapped at any time if the source file is rebuilt. This enables the user to see any changes made to the source code reflected instantly in the CubicTest editor.

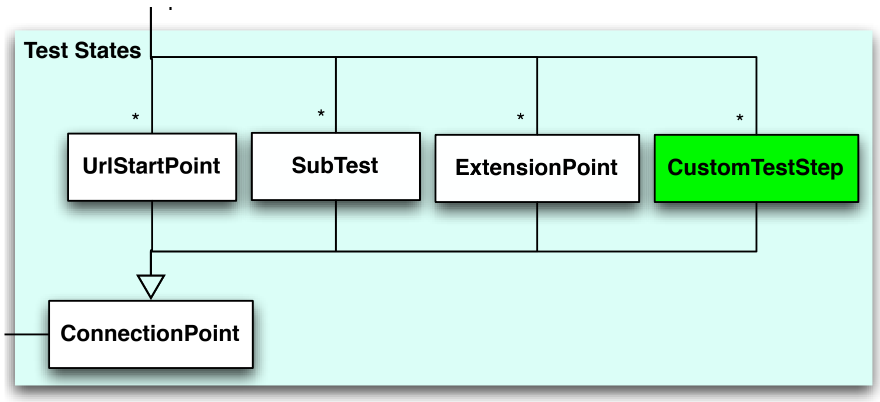


Figure 9.1: Another addition CubicTest's application model

9.3 Example

A user is creating an acceptance test for an online banking system, and wants to verify that the due date the application suggests is today or some date in the future. From figure 9.3 we see that the user first uses the ”Add Custom Test Step” command (1) and drags the control onto the work area (2) and enters a name, in this case “Verify Due Date”. At this point CubicTest creates the source code for the new test step and stores it in the project's src folder. As soon as the source file is compiled, a reference to the new “Verify Due Date” control appears at the bottom of the palette in the ”Custom Test Steps” section (3). The user can now edit the source of “Verify Due Date” by opening the file from the file browser or double-clicking the ”Verify Due Date” state (2) in the work area. Listing 9.1 is a simple example on what the source code for ”Verify Due Date” might look like.

Listing 9.1: Custom Test Step Example

```java
public class VerifyDueDate implements ICustomTestStep {

public void execute(Map<String,String> config, IElementContext context,
    Document document) throws Exception {
  Input element = (Input) document.getElements("//input[@id='dueDate']").
      get(0);
  Date today = new Date();
  DateFormat format = new SimpleDateFormat();

  try {
    Date dueDate = format.parse(element.toString());

    if(dueDate.before(today)) {
      throw new Exception("Due date is before today's date");
    }
  } catch (ParseException e) {
    throw new Exception("Invalid date format");
```

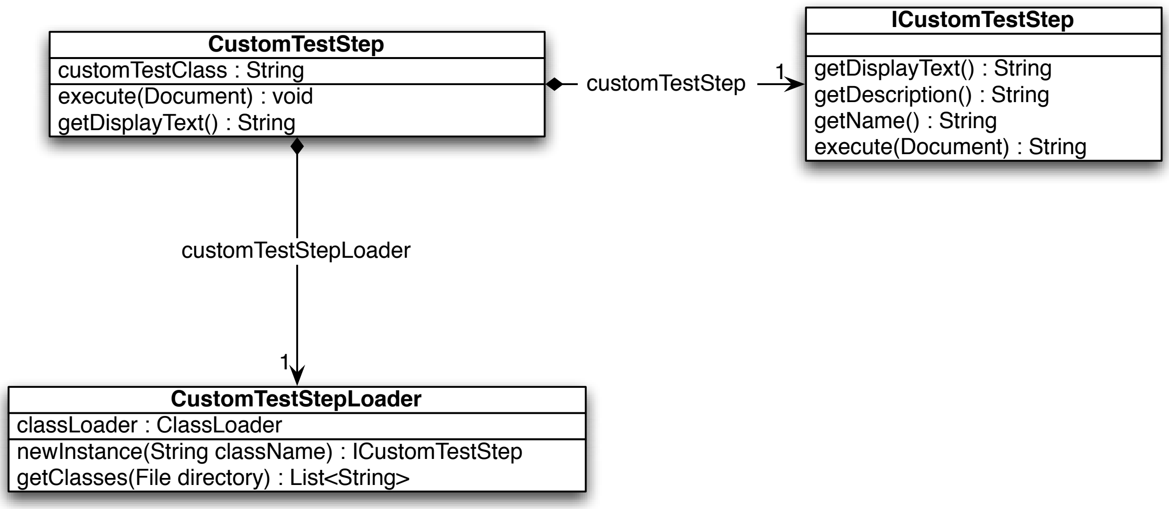

CustomTestStep
customTestClass : String
execute(Document) : void
getDisplayText() : String
customTestStep
1
ICustomTestStep
getDisplayText() : String
getDescription() : String
getName() : String
execute(Document) : String
customTestStepLoader
1
CustomTestStepLoader
classLoader : ClassLoader
newInstance(String className) : ICustomTestStep
getClasses(File directory) : List<String>

Figure 9.2: The Custom Test Step classes

```
    }

}

// Tooltip text
public String getDescription() {
  return this.getName();
}

 // The text is displayed in the editor
public String getDisplayText() {
  return this.getName();
}

 // The name of the control
public String getName() {
  return "Verify Due Date";
}

public List<String> getArgumentNames() {
  return null;
}
}
```

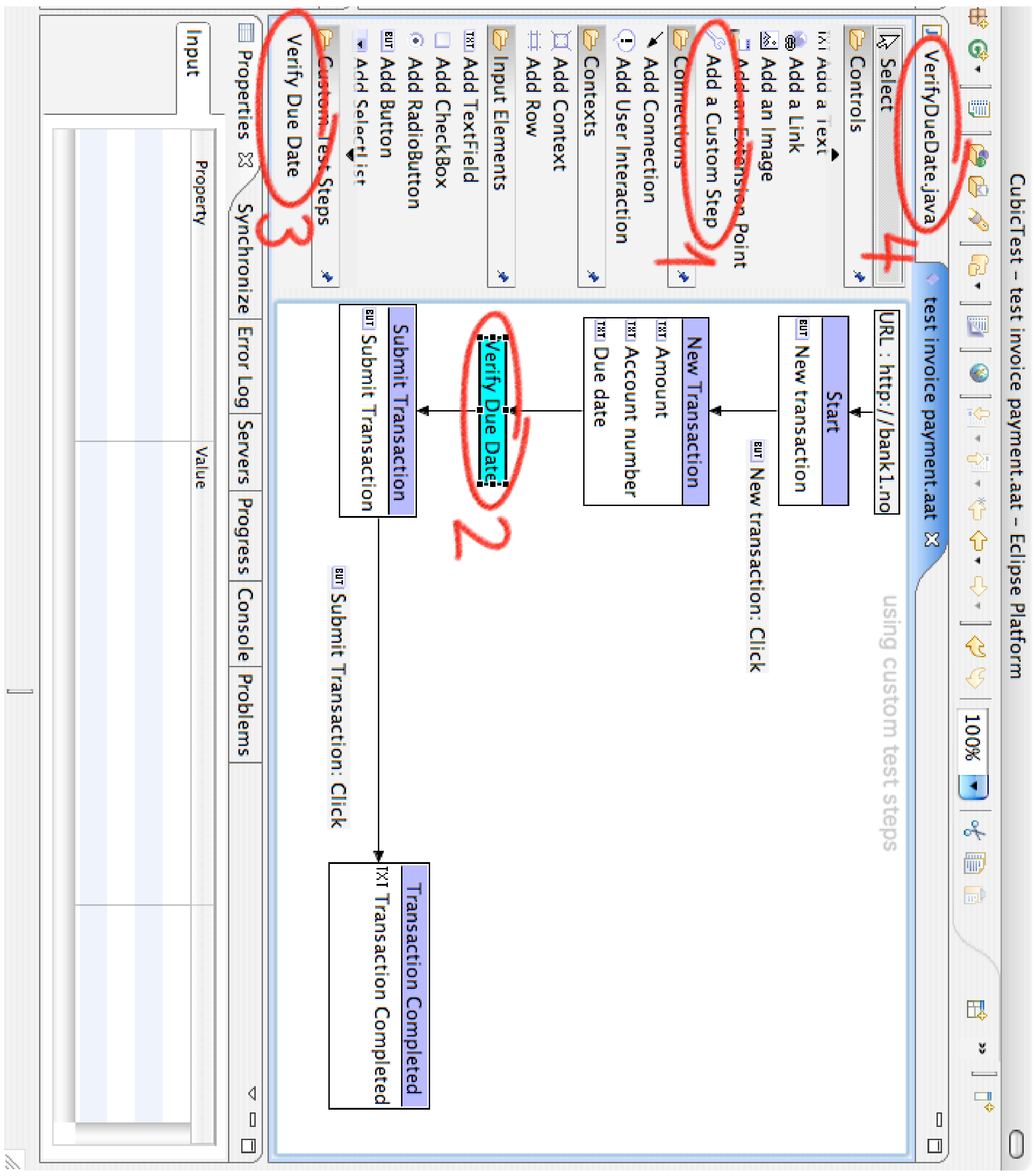


Figure 9.3: CustomTestStep Example Test

Chapter 10

Visualising Tests

"Every creator painfully experiences the chasm between his inner vision and its ultimate expression."

Isaac Bashevis Singer

American author

10.1 Requirements

It can be somewhat difficult, especially for non-technical users, to comprehend exactly what the test they're creating represents in terms of functionality. To help users with the process of creating tests up front, and hence better support the use of CubicTest as a part of the requirements specification process and as a driver for development, a way for users to explore their tests interactively was suggested. Giving the users the possibility to explore a prototype of their application will increase their understanding of the tests they produce.

10.2 Implementation

Using test definitions as a model of application composition and behaviour, a new exporter was implemented that produces a simple mock-up of the functionality defined in the individual tests. Every State is converted to a separate web page, with each of their Page Elements converted to their corresponding html elements, and each web page is linked together with the pages they are connected to in the test (see Figure 10.2). In addition, User Interactions are converted to a set of javascript rules that emulate the way the real web pages are supposed to behave, so that the interaction itself can be tested. Each transition is converted to a rule set that make up the required steps that needs to be performed for a transition to be triggered.

10.3 Example

The following example shows a test of the login system of CubicShop (see Figure 10.1), and the first web page from the corresponding html prototype (see Figure 10.2). CubicShop is an example application that's been developed along side the CubicTest plug-in to serve as a test bed for new features and as a demo application in documentation and presentations.

The web page displays all the elements from the state ”Frontpage”, as well the elements defined in the Common. On the right hand side, all the outbound transitions are displayed. In this case there's only one, which takes you to the ”Logged in” state. Performing the steps outlined in the transition will take the user to the next state. This is accomplished by the javascript embedded in the webpage, reproduced in Listing 10.1. Line 2 defines the rules for the user interaction, stating that the 'value' property of the first element should be the username ”skytsteink”, and that the value of the second element is irrelevant, the click action itself is all that we're interested in. Lines 5 and 10 defines the elements themselves, and connects the right actions on them to their corresponding rules (rule 0 and rule 1 on line 2).

Listing 10.1: The javascript code for the user interaction

```
 1  <script type='text/javascript'>
 2  UserInteractions.actions['1157551842439.html'] = [['value', 'skytsteink
        '], ['value', null]];
 3  </script>
 4  [...]
 5  <input type="textfield" value="Username" name="username"
 6  id="username"
 7  onKeyUp="UserInteractions.test('1157551842439.html', 0, this);"
 8  class="actionable" />
 9  [...]
10  <input type="button" value="Log in" name="logIn"
11  id="logIn"
12  onClick="UserInteractions.test('1157551842439.html', 1, this);"
13  class="actionable" />
```

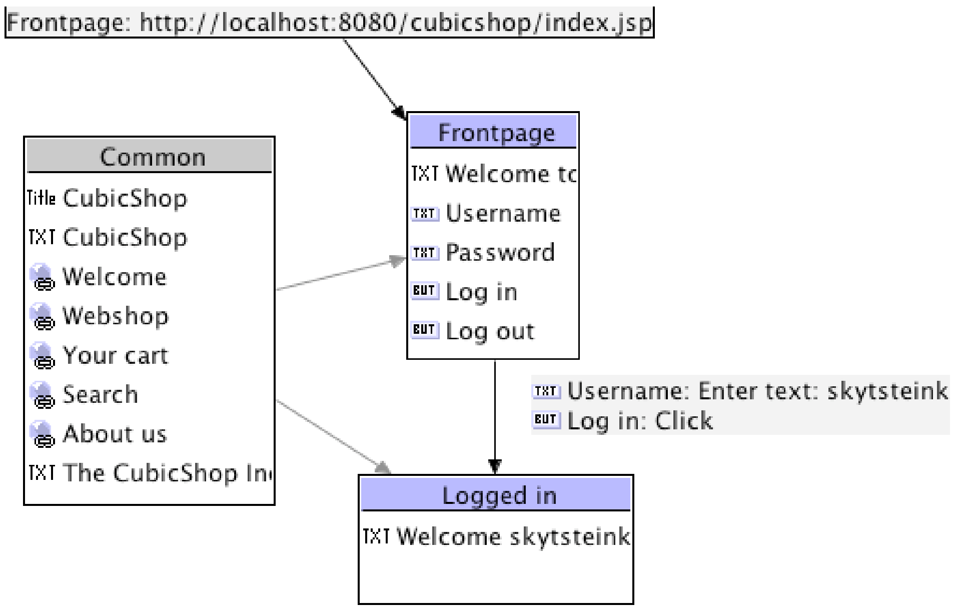


Figure 10.1: Test of the login form of the CubicShop application

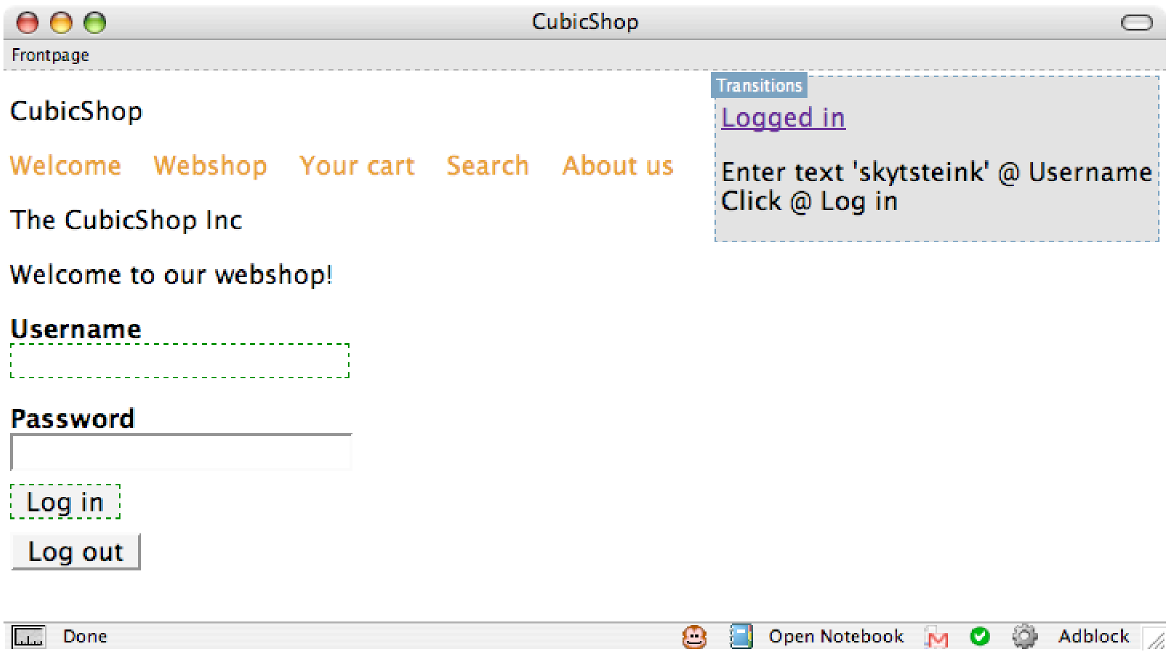


Figure 10.2: The resulting html prototype of the test in figure 10.1

Chapter 11

Testing Existing Applications - Visual Recording of User Interaction

"Automatic simply means that you can't repair it yourself."
Mary H. Waldrip
British author

11.1 Requirements

Recognising that not everyone wants to do acceptance testing as part of a test driven development process, or even do TDD at all, we wanted to simplify the process of creating tests "after the fact". Allowing the user to create tests by interacting directly with the running web-application would hopefully speed up the test creation process and reduce the number of errors in the tests.

11.2 Implementation

To accomplish this, a way to integrate a web browser with the CubicTest Plug-in was needed. There is a built-in web browser in the Java Swing framework, but the exposed API turned out to be much too simplistic for our needs, since we needed a way to communicate directly with the browser's Document Object Model as well as catch any events triggered by the user.

We finally settled on using Selenium Remote Control. Selenium RC is a part of the Selenium test framework, written in JavaScript, and built to run directly in the browser, while being controlled remotely from Java (see Section 7.3.2). Using Selenium RC as a relay between the browser and the application would give us full access to the browser, as well as allowing the record functionality to work across platforms and browser software.

11.2.1 Architectural Overview

To keep the CubicTest source tree lean, we decided to implement the recorder as a separate Eclipse plug-in (see Figure 11.1). Sending commands to the browser was already implemented in Selenium RC, but to communicate back from the browser to the recorder we decided to use JSON-RPC-Java[1]. JSON-RPC-Java is a Remote Procedure Call implementation for exposing server side Java objects as JavaScript objects on the client, using JSON[2] as the message carrier. JSON, or JavaScript Object Notation, is a subset of the JavaScript language made specifically for describing data structures. By extending the Selenium RC server with JSON-RPC-Java, and exposing an instance of JSONRecorder on the client side (see Figure 11.2), we could control the entire recording process from the browser through an easy-to-use JavaScript object.

On the client side, we implemented a JavaScript library to handle the capturing of the user's interaction with the browser. By registering keyboard- and mouse-event listeners on the root node (body element) of the web page we could capture all user input to the web page. In the current implementation, only mouse clicks and text entry is handled, and then only when triggered on objects that can be represented in the CubicTest model.

When a valid event is triggered on a recognised html element, the html element is serialised to JSON on the client side and then fed via JSON-RPC to JSONRecorder's addAction method, together with the correct action identifier (CLICK, DBLCLICK or ENTER_TEXT). On the server side, JSONElementConverter is responsible for converting JSON strings to CubicTest PageElements, before the PageElement and a corresponding UserInteraction is added to the data model of the CubicTest editor. We also implemented a custom context menu on the browser side to allow adding various checks without creating UserInteractions.

[1] http://oss.metaparadigm.com/jsonrpc/
[2] http://www.json.org/

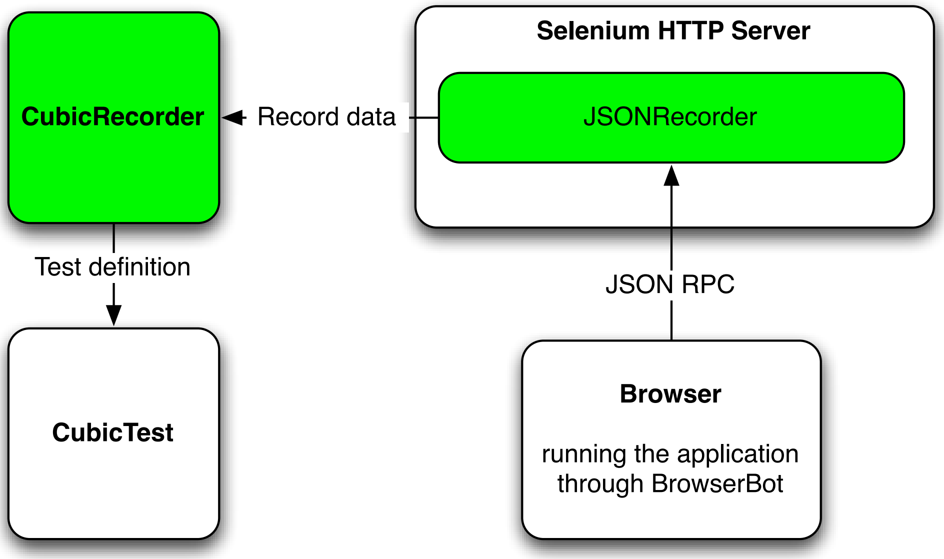


Figure 11.1: The CubicRecorder architecture

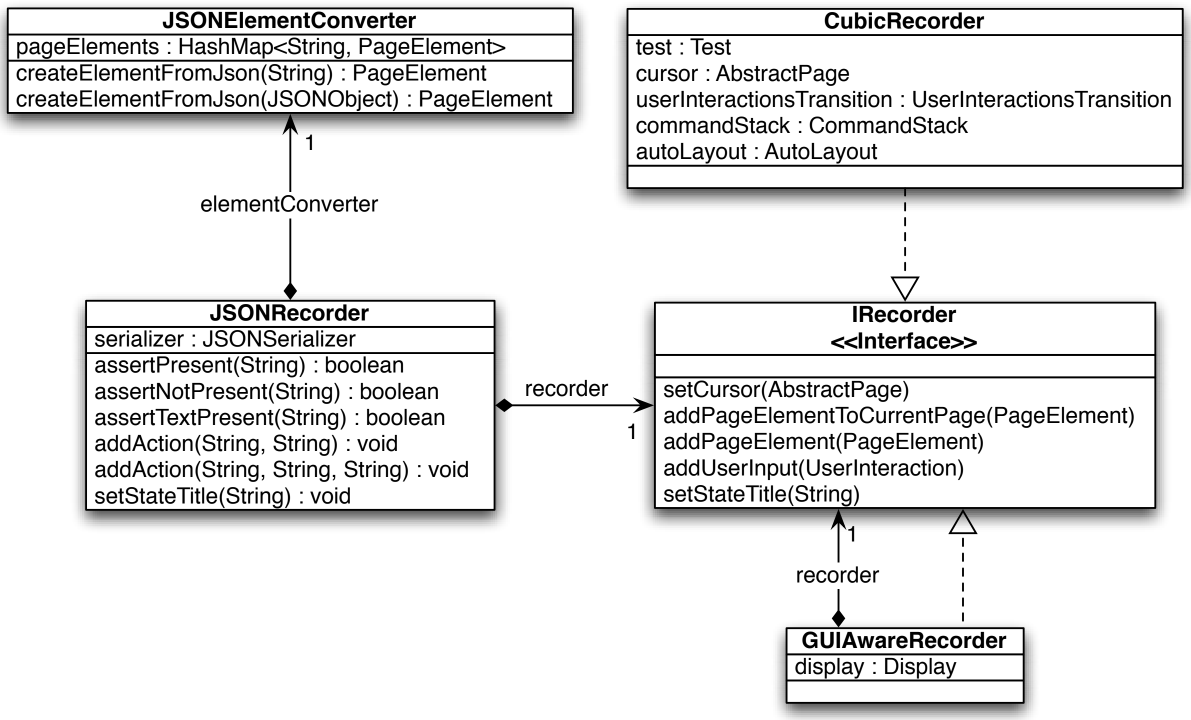

JSONElementConverter
pageElements : HashMap<String, PageElement>
createElementFromJson(String) : PageElement
createElementFromJson(JSONObject) : PageElement
1
elementConverter
CubicRecorder
test : Test
cursor : AbstractPage
userInteractionsTransition : UserInteractionsTransition
commandStack : CommandStack
autoLayout : AutoLayout
JSONRecorder
serializer : JSONSerializer
assertPresent(String) : boolean
assertNotPresent(String) : boolean
assertTextPresent(String) : boolean
addAction(String, String) : void
addAction(String, String, String) : void
setStateTitle(String) : void
recorder
1
IRecorder
<<Interface>>
setCursor(AbstractPage)
addPageElementToCurrentPage(PageElement)
addPageElement(PageElement)
addUserInput(UserInteraction)
setStateTitle(String)
1
recorder
GUIAwareRecorder
display : Display

Figure 11.2: The most important classes of the CubicRecorder Plug-in

Chapter 12

Experiment Design

"A thinker sees his own actions as experiments and questions - as attempts to find out something. Success and failure are for him answers above all."
Friedrich Nietzsche
German philosopher

To collect data for the GQM measurement plan outlined in chapter 4, we held a set of user testing sessions. Each user was first given an introduction to the software, followed by a set of exercises and a questionnaire.

The study aimed to evaluate the following hypotheses:

H_{01}: The html exporter has no effect on the number of errors produced
H_{02}: The html exporter has no effect on the user's understanding of the tests
H_{03}: The html exporter has no effect on the user's efficiency when creating tests
H_{04}: The recorder has no effect on the number of errors produced
H_{05}: The recorder has no effect on the user's efficiency when creating tests
H_{06}: The new features (the html exporter and the recorder) has no effect on the user's understanding of CubicTest
H_{07}: The new features (the html exporter and the recorder) has no effect on the user's preception of CubicTest's usability
H_{08}: The new features (the html exporter and the recorder) has no effect on the user's preception of CubicTest's learnability

Each of these hypotheses will be tested against their counterparts:

H_{11}: The html exporter has an effect on the number of errors produced, either positive or negative
H_{12}: The html exporter has an effect on the user's understanding of the tests,

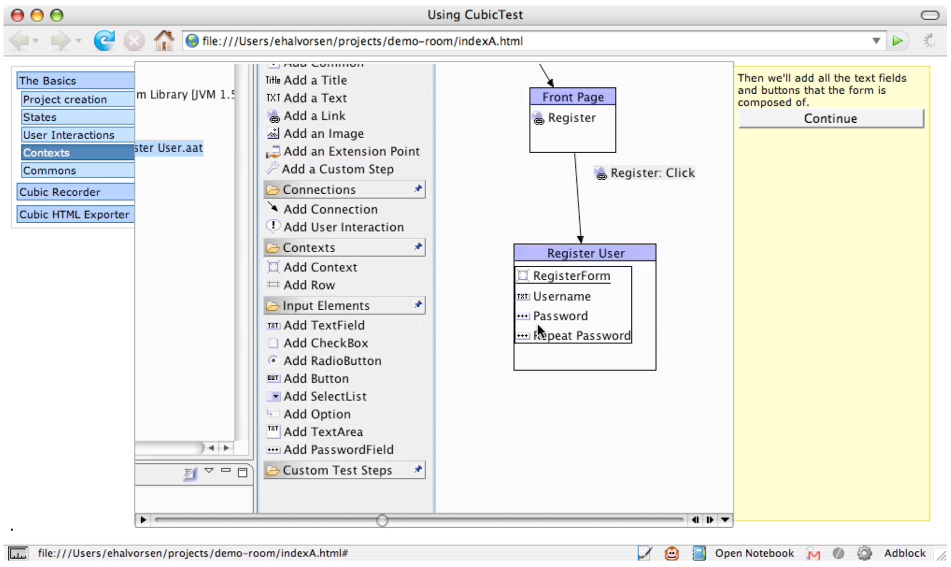


Figure 12.1: The subjects was shown a set of videos to introduce them to CubicTest

either positive or negative
H_{13}: The html exporter has an effect on the user's efficiency when creating tests, either positive or negative
H_{14}: The recorder has an effect on the number of errors produced, either positive or negative
H_{15}: The recorder has an effect on the user's efficiency when creating tests, either positive or negative
H_{16}: The new features (the html exporter and the recorder) have an effect on the user's understanding of CubicTest, either positive or negative
H_{17}: The new features (the html exporter and the recorder) have an effect on the user's preception of CubicTest's usability, either positive or negative
H_{18}: The new features (the html exporter and the recorder) have an effect on the user's preception of CubicTest's learnability, either positive or negative

12.1 Introduction

Each participant was given a short introduction, explaining the basics of eXtreme Programming and software testing in general, as well as acceptance testing. To ensure everyone got approximately the same background information, a set of video tutorials was created, covering the basic usage of CubicTest, as well as the usage of the new plug-ins. The introduction and video tutorials took approximately 15 minutes.

12.2 Exercises

Each test subject was randomly assigned to one of two groups. Group A was given a version of CubicTest with the new plug-ins, while Group B used version of CubicTest with the new features disabled. After the introduction, the subject worked through a set of exercises, shown in Appendix C. Each exercise was designed with specific metrics in mind:

Exercise A1 was given to answer *Metric 1 - Number of errors in tests*, with the the subjects in Group A inspecting their test with the html Exporter (see chapter 10) and making further adjustments to their test as they saw fit. After doing a pilot test with the exercises defined here, the following error categories were defined:

- **Missing actions in User Interaction**
  Omitting required actions from a User Interaction (see chapter 6), like forgetting to fill out a form field or checking a check box.
- **Missing Page Elements**
  Forgetting to include a Page Element integral to the User Story being tested, for instance forgetting to include a Submit button on a form.
- **Missing User Interactions**
  Missing an entire User Interaction transition.
- **Missing States**
  Missing an entire State / Transition Node
- **Missing verification**
  Forgetting to verify that the action performed actually resulted in the desired outcome, for instance a feedback message saying that some action was performed successfully.
- **Wrong use of Page Elements**
  Using the wrong type of Page Element, for instance using a password field instead of a regular text field, or vice versa.
- **Erroneous or incomplete actions**
  Creating wrong or partial actions, for instance selecting a click action instead of ”enter text”, or selecting ”enter text” without specifying the text to be entered.
- **Logical errors**
  Other errors that deal with the underlying logic of the test, for instance omitting a whole step in some process, or misplacing a Page Element.

The purpose of **Exercise A2** was to measure *M5 - Comprehension of Tests*, looking at how inspecting an unknown test in the html exporter influences Group A’s understanding of the test.

Exercise B was designed to study how using the recorder plug-in affects time usage and error rate (Metrics *M1 - Number of Errors in Tests* and *M2 - Time Usage*).

The exercises were given as simple User Stories, and they were hence quite open-ended, as defining each exercise in too much detail would leave little room for the participants to make their own interpretation of the software and make their own decisions and mistakes.

Doing comparisons on the number of errors would be difficult with this set-up, as one might argue that longer tests will open for more possibilities of making mistakes. There are, however, several ways one can count errors:

- **Number of errors**
  This is probably the simplest form, but it raises some issues, for instance how to count missing actions in User Interactions - is three missing actions one error (faulty User Interaction) or three (missing actions)? Nor does it account for the length of the test, so subjects creating longer tests will potentially create more errors.
- **Number of errors divided by number of test elements**
  This resembles the faults / Lines of Code metric often used in studies dealing with software quality. It has many of the same problems as Number of Errors, with the exception that it does take the length of the tests into account.
- **Number of observed error types**
  Instead of counting individual errors, one can look at what types of errors are present in the test. This both deals with the problem of deciding what exactly constitutes *one* error, and isn't greatly affected by the length of the test, but doesn't really represent the number of faults, only what types of errors the user makes.

This experiment will use both *Number of errors divided by number of test elements* and *Number of observed error types* as a basis for measuring error rate. The length of each test is measured as follows, starting with the Start Node:

- Each Transition Node (Start Node, Page, Common, etc.) counts as 1
- Each transition counts as 1 + the number of actions it contains
- Each Page Element counts as 1, including Contexts

The total length is then calculated by adding together the test's length with the number of missing elements.

12.3 Questionnaire

After the exercises, the subject filled in a questionnaire relating to several aspects of the CubicTest software, designed to measure *M6 - Comprehension of CubicTest*, and their subjective opinion of the software's usability and learning curve (metrics M3 and M4), before participating in a short, informal interview about their experience with the software, to allow for clarifications and to let the subjects ask any questions they might have. Each test session took approximately one hour from start to finish.

Part IV

Evaluation

The user testing was conducted during week 13, using the usability lab at IDI. In total, 13 first and second year students participated. This part of the thesis will take a closer look at the collected data and evaluate the results of the study. Then the individual features and their effect will be discussed. Finally, some suggestions for future work related to CubicTest will be presented.

Chapter 13

Threats to Validity

Here we will highlight some of the most important threats to the validity of the study. The following list is based on a set of possible threats suggested by Wohlin et al. in [4]:

- **Low statistical power**
  An experiment's statistical power expresses the experiment's ability to reveal a true pattern in the data. Lower statistical power leads to a higher risk of drawing an erroneous conclusion, either by rejecting a valid hypothesis, or failing to reject an invalid one. There are rather few testers in this experiment, and hence this is an important threat to consider. Appendix A examines the statistical power of the study, and concludes that we have to increase the β to around 30%, given our sample size of 13 and the measured effect sizes.

- **Reliability of measures**
  The validity of an experiment is highly dependent on the reliability of the measures. This in turn may be influenced by many factors, including poor question wording, bad instrumentation or bad instrument layout. The basic principle is that when you measure a phenomenon twice, the outcome shall be the same. Some of the metrics defined in this experiment are based on subjective measures, and can as such not be guaranteed to yield the same results if measured again. However, a standardised introduction, and guidance to how one should do the exercises should help mitigate this threat.

- **Maturation**
  This is the effect of the subjects reacting differently as time passes, for instance becoming bored, or learning from interacting with the software. For the most part, this should affect both groups equally. However, if the learning curve differs between the two versions of CubicTest, this could be a problem.

- **Selection**
  How the subjects are selected may influence the outcome of the study. People who volunteer to participate in a study, as is the case here, are often more motivated

to learn new things. Even though the test subjects was randomly assigned to the two groups, and hence selection should affect each group equally, this does affect the external validity of the experiment, that is, it might limit our ability to apply the results to the general population. However, we are mostly concerned with the effects of the different treatments, and, provided they have a moderate to large effect, it should be possible to draw some general conclusions.

- **Fishing and experimenter expectancies**
  This threat deals with researchers' and test subjects' preconceptions about the specific outcome of the study. The participants was not informed about the exact focus of the study, and as such shouldn't have too many opinions on the outcome. They were also informed that the software is still a proof-of-concept, and of the importance of giving accurate and honest responses. As for the researchers, this threat can be somewhat mediated by defining the study in detail before conducting the experiment.

Chapter 14

Measurements

In this chapter we will take a closer look at the data collected in the experiment. The data is summarised in figures 14.2 and 14.3. All the data has been evaluated using two-sided t-tests, with a significance level of 0.10.

14.1 M1 - Number of Errors in Tests

Looking at the data from the exercises, we see that, for exercise A1, which was conducted with the HTML Exporter by group A, there is a significant difference ($p < 0.1$) for both the occurrence of different error types, overall number of errors and errors / element. We also see that the difference is quite large - about 60%, and the effect size ranges from 1 to around 1.3, which is considered to be moderate to large effect.

For exercise B, completed by group A by using the recorder, we see similar results, but with even higher differences between the two groups.

This shows that Group A, using the new additions to CubicTest, created significantly fewer errors than the control group. The data also suggests that Group A spent slightly more time on exercise A. Although this is as expected, since they performed the additional task of inspecting their test in the HTML Exporter, this could have had some impact on the error reduction. However, looking at the scatter plot of Time Usage vs. Errors / Element Figure 14.1 and the results of a regression analysis (Table 14.1) we see that there is no observable correlation, with R^2 around 0.1.

This leads us to reject the hypotheses

H_{01}: The html exporter has no effect on the number of errors produced
H_{04}: The recorder has no effect on the number of errors produced

We will thus accept the alternative hypotheses

Multiple R	0.31614502
R^2	0.09994767
Adjusted R^2	0.01812473
Observations	13

Table 14.1: Regression analysis of Time Usage vs Errors / Element for exercise A1

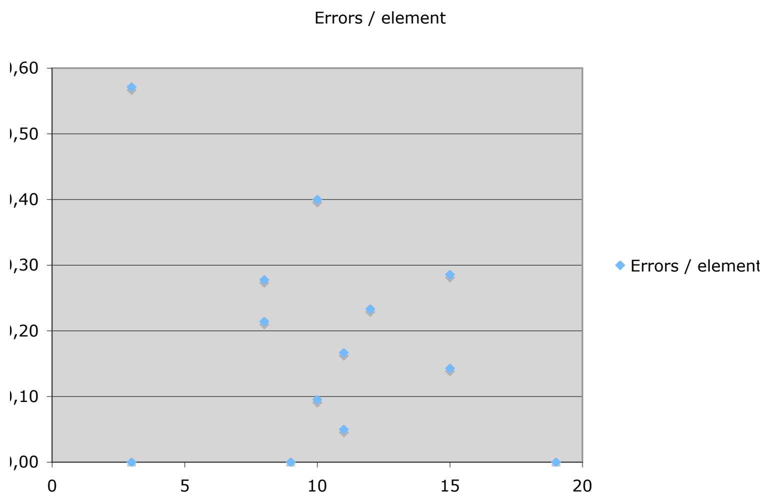


Figure 14.1: Scatter plot of Time Usage vs. Errors / Element for exercise A1

H_{11}: The html exporter has a positive effect on the number of errors produced
H_{14}: The recorder has a positive effect on the number of errors produced

14.2 M2 - Time Usage

The data shows no significant difference in time usage between the two groups for exercise A1, and the effect size is only moderate, at around 0.6, leading us to keep the null hypothesis

H_{03}: The html exporter has no effect on the user's efficiency when creating tests

However, time usage for exercise B has a significant difference, with group A being, on average, about 30% faster, and the effect size at 1.0 is considered to be moderate to large, leading to the rejection of the hypothesis

H_{05}: The recorder has no effect on the user's efficiency when creating tests

and the acceptance of the alternate hypothesis

H_{15}: The recorder has a positive effect on the user's efficiency when creating tests

14.3 M3 - Perceived Ease of Use and M4 - Perceived Ease of Learning

The new features does not seem to have a noticeable effect on the user's perception of the application's ease-of-use or ease-of-learning, with both high p-values and only moderate effect sizes. As the users in group A did become more efficient, and produced fewer errors, it might be that the effect simply is too small to be noticed in such a small experiment. Both groups also scored the application quite high, with no score below 3, on a four point scale, leaving little room for improvement. A more fine grained scale could perhaps have produced a more discernible effect.

In any case, we have to keep the following null hypothesis

H_{08}: The new features (the html exporter and the recorder) have no effect on the user's preception of CubicTest's learnability

14.4 M5 - Comprehension of Tests

This metric was measured through exercise A2, with the answers scored from *0 - Completely wrong* to *2 - Correct* (see page 14). The data shows no difference between the performance of the two groups, and only a small to moderate effect size, leading us to conclude that the HTML Exporter has little effect on the users' comprehension tests, and thus we keep the null hypothesis

H_{02}: The html exporter has no effect on the user's understanding of the tests

14.5 M6 - Comprehension of CubicTest

To measure the users' understanding of key concepts in CubicTest, they were given a questionnaire after the exercises were completed. The answers were rated from *0 - Completely wrong* to *2 - Correct* (see page 15), and are shown in the last four columns of figure 14.3.

With such high p-values and low effect sizes, the results cannot be considered significant, and thus we have to keep the null hypothesis

H_{06}: The new features (the html exporter and the recorder) have no effect on the user's understanding of CubicTest

14.6 M7 - CubicTest's Utility

To keep the scope more manageable, the new test elements, *Java Test Steps* and *Sub-Tests*, were excluded from the experiment.

The SubTest control, however, has been used with great success in tests for our demo application, CubicShop, and has made it possible to rapidly create several, complex tests that otherwise would be both time consuming to write and unwieldy to maintain. By being able to reuse the same test procedures over and over, we have been able to cover an increased number of test scenarios, almost without increasing the size of the test suite. At the time of writing, the test suite contains, in total, 233 test controls distributed over 12 files (see Table 14.2). By calculating the length of each test including each subtest, we get the total number of elements that would have been needed to recreate the same test suite without subtests, in this case 394, or an increase of about 70%.

This is, however, anecdotal evidence, and the frequency by which subtests are used might be exaggerated, since the main purpose of the test suite is to test CubicTest, and not the demo application. To get a more accurate picture of the usefulness of this feature, one would have to look at "real world" CubicTest-test suites. Unfortunately, at the time of writing, this is not available.

The *Java Test Step* control relies on a test runner written in Java. Two such projects, CubicUnit and Cubic Selenium Runner, are in the works, but not yet completed, making it problematic to do a good assessment of the value of the new feature. Thus, this area will have to be investigated further. The Cubic team does, however, consider custom test steps to be an integral part of getting CubicTest more widely adopted, as being able to write tests partly in Java will go a long way towards eliminating the need for writing functional tests outside of CubicTest.

Test Name	**Test Elements**	**Extension Points**	**Subtests**	**Total number of Elements**	**Referenced SubTests**
contents	52			52	
delete after purchase	20		1	82	purchase after login
feedback	17			17	
frontpage	32	5		32	
login	13	1		13	
purchase cube	5		4	47	login, open webshop, buy box, log out
purchase after login	15	1	3	62	purchase, login, purchase
purchase	17	1		17	
search	11		1	43	frontpage
buy box	12	2		12	
log out	10	1		10	
open webshop	7	2		7	
Total	211	13	9	394	

Table 14.2: Statistics from the CubicShop test suite. Total number of Elements is calculated as Test Elements + Total number of Elements for each subtest.

Group A: New version
Group B: Old version

		Exercise A1					Exercise A2	Exercise B				
User	**Group**	**Time usage**	**Error types**	**Errors**	**Total Length**	**Errors / element**	**Score (0-2)**	**Time usage**	**Error types**	**Errors**	**Total Length**	**Errors / Element**
1	A	11	1	3	18	0,17	2	10	0	0	22	0,00
3	A	12	2	7	30	0,23	0	9	1	1	23	0,04
5	A	15	1	2	14	0,14	2	11	1	1	24	0,04
7	A	3	0	0	16	0,00	2	2	1	1	22	0,05
9	A	10	2	2	21	0,10	2	4	0	0	22	0,00
11	A	19	0	0	27	0,00	2	5	0	0	24	0,00
13	A	11	1	1	20	0,05	2	6	1	1	24	0,04
2	B	15	2	6	21	0,29	2	9	2	9	24	0,38
4	B	9	0	0	17	0,00	2	8	2	2	23	0,09
6	B	10	3	8	20	0,40	2	8	3	3	24	0,13
8	B	8	3	5	18	0,28	2	12	2	6	26	0,23
10	B	3	5	8	14	0,57	2	10	1	1	23	0,04
12	B	8	2	3	14	0,21	2	10	0	0	25	0,00
	p-value:	0,293	**0,056**	**0,088**	0,206	**0,034**	0,377	**0,088**	**0,032**	**0,044**	**0,078**	**0,045**
	Avg:	10,31	1,69	3,46	19,23	0,19	1,85	8,00	1,08	1,92	23,54	0,08
	Avg. A:	11,57	1,00	2,14	19,23	0,10	1,71	6,71	0,57	0,57	23,00	0,02
	Avg. B:	8,83	2,50	5,00	17,33	0,29	2,00	9,50	1,67	3,50	24,17	0,14
	Diff. In %:	31%	-60%	-57%	11%	-66%	-14%	-29%	-66%	-84%	-5%	-83%
	Effect size:	0,62	-1,16	-1,03	0,77	-1,30	-0,53	-1,07	-1,33	-1,21	-1,07	-1,20

Figure 14.2: Summary of the data from the exercises

Group A:	New version
Group B:	Old version

						Comprehension			
User	**Group**	**Purpose of CubicTest**	**"CubicTest is easy to use"**	**"CubicTest is easy to learn"**	**Experience with Eclipse**	**Page/State**	**Common**	**Context**	**User Interaction**
1	A	3	3	4	3	2	2	0	2
3	A	3	4	3	3	2	2	2	2
5	A	2	3	4	2	0	0	0	2
7	A	2	3	4	4	2	2	2	2
9	A	3	3	4	3	2	2	1	2
11	A	3	4	4	3	2	2	2	2
13	A	3	4	4	3	2	0	0	1
2	B	3	3	3	3	2	2	2	2
4	B	3	3	3	3	2	0	0	1
6	B	2	4	4	3	0	2	0	1
8	B	3	3	4	3	2	2	0	2
10	B	0	3	3	3	1	1	0	2
12	B	3	3	4	4	2	2	0	2
p-value:		0,459	0,349	0,193	0,567	0,637	0,891	0,220	0,459
Avg:		2,54	3,31	3,69	3,08	1,62	1,46	0,69	1,77
Avg. A:		2,71	3,43	3,86	3,00	1,71	1,43	1,00	1,86
Avg. B:		2,33	3,17	3,50	3,17	1,50	1,50	0,33	1,67
Diff. In %:		16%	8%	10%	-5%	14%	-5%	200%	11%
Effect size:		0,41	0,55	0,76	-0,33	0,27	-0,08	0,73	0,42

Figure 14.3: Data from the questionnaire

Chapter 15

Goal Attainment and Conclusion

15.1 Goal

In chapter 4 we specified the following goal for this study:

> Analyze **CubicTest**
> for the purpose of **improving the product**
> with respect to **usability, utility, user comprehension, quality and efficiency for writing and analyzing acceptance tests**
> as seen from the **software developer**'s point of view
> in the context of **web application development**

To evaluate the attainment of this goal, five questions were defined. We will now look at each question, and the answers provided by our metrics.

15.2 Questions

Question 1 *Quality*

> Does the use of the new features increase the quality of the tests produced, compared to using CubicTest without the new features?

To measure this, we have metric M1 - Number of Errors in Tests. As we can see from chapter 14, there's an overall improvement in error rates both from using the HTML Exporter and the Recorder.

Question 2 *Efficiency*

> Does the use of the new features increase the efficiency of the test creation process, compared to using CubicTest without the new features?

To answer this question we have metric M2 - Time Usage. Looking at the results in chapter 14, we see that the Recorder does have a positive effect on time usage, with the users of group A being on average about 30% faster than the control group.

Question 3 *Comprehension*

> Does the use of the new features improve the user's comprehension of CubicTest compared to using CubicTest without the new features?

This question is answered by metrics M5 - Comprehension of Tests and M6 - Comprehension of CubicTest. The data shows no difference between the two groups in this area, and we must conclude that the new features do not improve the users' comprehension and understanding of the concepts in CubicTest.

Question 4 *Utility*

> Does the new features increase the utility of CubicTest both with respect to the old version, and compared to other, similar tools?

While M7 - CubicTest's Utility lacks reliable, empirical data, it seems safe to conclude that the new features has increased CubicTest's utility and usefulness.

Question 5 *Usability*

> Does the new features increase the usability of CubicTest compared to the old version?

Looking at metric M3 - Perceived ease of Use, we see no noticeable improvement, all the test subjects found the application's usability to be more or less the same. However, M1 - Number of Errors in Tests shows that Group A did create significantly fewer errors, which does indicate an improvement in the usability or understandability of the application.

Question 6 *Learnability*

> Does the new features make CubicTest easier to learn, compared to the old version?

M4 - Perceived Ease of Learning shows no difference between the two groups. Neither does the metrics for comprehension, M5 and M6. Thus we can only conclude that the new features doesn't influence the application's learning curve.

15.3 Conclusion

This project set out to explore further improvements of CubicTest, to make it a more valuable tool for acceptance testing. Four suggestions has been presented and implemented, and two of them has been evaluated through an empirical experiment. The results shows that the new features have increased CubicTest's usability, utility, quality and efficiency, and all of them have been accepted into the CubicTest source code repository, and will be included in the final 1.0 release later this year.

Chapter 16

Future Work

16.1 Future Work on CubicTest

While a lot of new functionality has been added to CubicTest over the last year, both in relation to this project as well as additions made by the rest of the Cubic team, there will always be room for further improvements:

- *"Record from here"*
  The new Recorder does make it easier to create new tests. However, by combining the Recorder with the Selenium Runner, it should be possible to let the user continue recording from an arbitrary point in the test, buy running the existing steps up to the selected state, and then hand over control to the recorder.

- *Interactive creation of User Interactions*
  By combining the HTML Exporter with the Recorder, it should be possible to create User Interactions by interacting with the html mock-up.

- *Test Refactoring*
  It should be possible to automate various refactoring tasks, like extracting part of a test as a SubTest, or vice versa, merging two or more States, etc.

- *Embedding the Recorder*
  The Recorder, as it is implemented now, uses an external browser instance to run the record session. It should be possible to use one of the Java web browser widgets available to run the record session within Eclipse.

- *Connecting code and acceptance tests*
  It would be interesting to look at the possibility of connecting tests from CubicTest with the actual code for the project, and keep track of test coverage, track code changes, etc.

- *Supporting other platforms*

The development of CubicTest has so far been entirely focused on testing web applications. It should, however, also be possible to support testing of other kinds of GUI platforms, for instance Java/Swing, Qt, or Adobe Flex.

16.2 Future Empirical Experiments

As CubicTest reaches the final 1.0 release later this year, we will hopefully start seeing some projects adopting CubicTest to handle acceptance testing. If so, it would be very interesting to take a closer look at how the teams use the tool and it's various features, as well as customer involvement and how the developers and customers collaborate on acceptance testing.

Part V

Appendix

Appendix A

Statistical Power

Using the formula $N = \frac{4(U_{\alpha/2}+U_{\beta})^2}{ES^2}$ we can calculate the needed effect size, given the number of test subjects (in this case 13) and the selected α and β values. Ideally, these values should be as low as possible. If we choose $\alpha = 10\%$ and $\beta = 20\%$, we get

$$ES = \sqrt{\frac{4(1.65+0.84)^2}{13}}$$
$$ES = 1.38$$

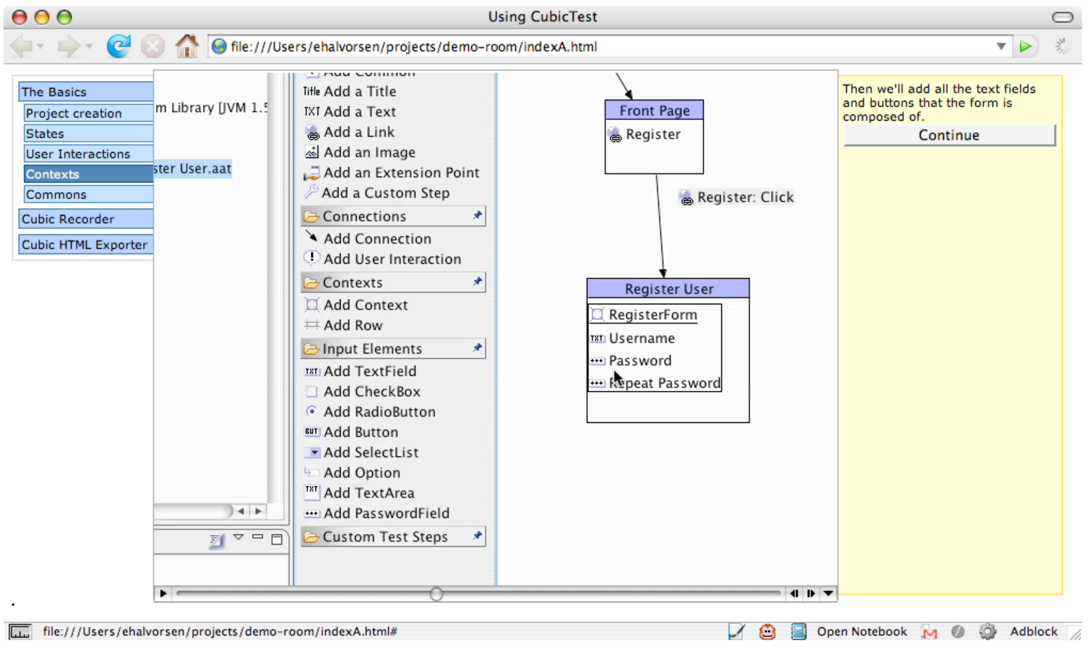
Looking at the results in figure 14.2 we see that the largest effect sizes are in the range 1.0 to 1.33, thus the statistical power is a little weak. We have to increase the β to 30% to get the required effect size around the levels of our results:

$$ES = \sqrt{\frac{4(1.65+0.52)^2}{13}}$$
$$ES = 1.20$$

Appendix B

User Documentation

A video introduction to using the new features can be found in the attachment, in the folder *documentation*. To view it, open index.html in Firefox or Internet Explorer (Opera is unfortunately not supported).



Then we'll add all the text fields and buttons that the form is composed of.

Figure B.1: Screenshot of the accompanying video tutorials

Appendix C

Exercises & Questionnaire

This is the exercises and questionnaire document used during the user testing (in norwegian):

Spørreskjema – Brukertest av Cubic Test

Brukernr: _______

Linje: _______________

År: _____

1. A – SquirrelMail

US1 - Send Epost med kopi
Brukeren ønsker å sende en epost med kopi (CC)

Tidsforbruk (i min): _______

US2 - ??
Beskriv kort hva testen gjør:

2. B - CubicShop
US1 - Registrering
Brukeren ønsker å registrere seg som ny bruker, med fornavn, etternavn, passord, og epost-adresse

Tidsforbruk (i min): _______

3. Hva er formålet med Cubic Test?

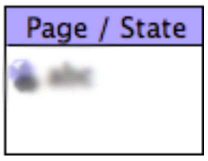
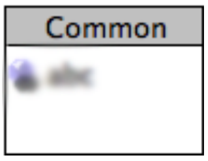
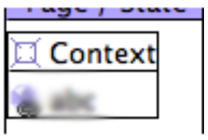
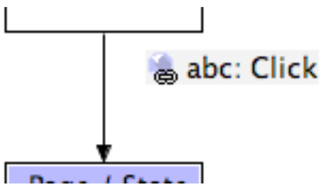
Under følger noen påstander. Kryss av på skalaen i hvilken grad du er enig/uenig i påstanden.

4. ”Cubic Test er enkelt å bruke”
[ ] Helt enig
[ ] Delvis enig
[ ] Delvis uenig
[ ] Helt uenig

5. ”Cubic Test er lett å lære”
[ ] Helt enig
[ ] Delvis enig
[ ] Delvis uenig
[ ] Helt uenig

6. Hvor mye erfaring har du med Eclipse?
[ ] Har brukt det mye
[ ] Har brukt det en del
[ ] Har nesten aldri brukt det
[ ] Har ingen erfaring med Eclipse

7. **Forklar følgende funksjoner i Cubic Test:**

	**Page / State**
	**Common**
	**Context**
	**User Interaction**

8. **Var det funksjoner eller konsepter i Cubic som du synes var spesielt vanskelig å forstå?**

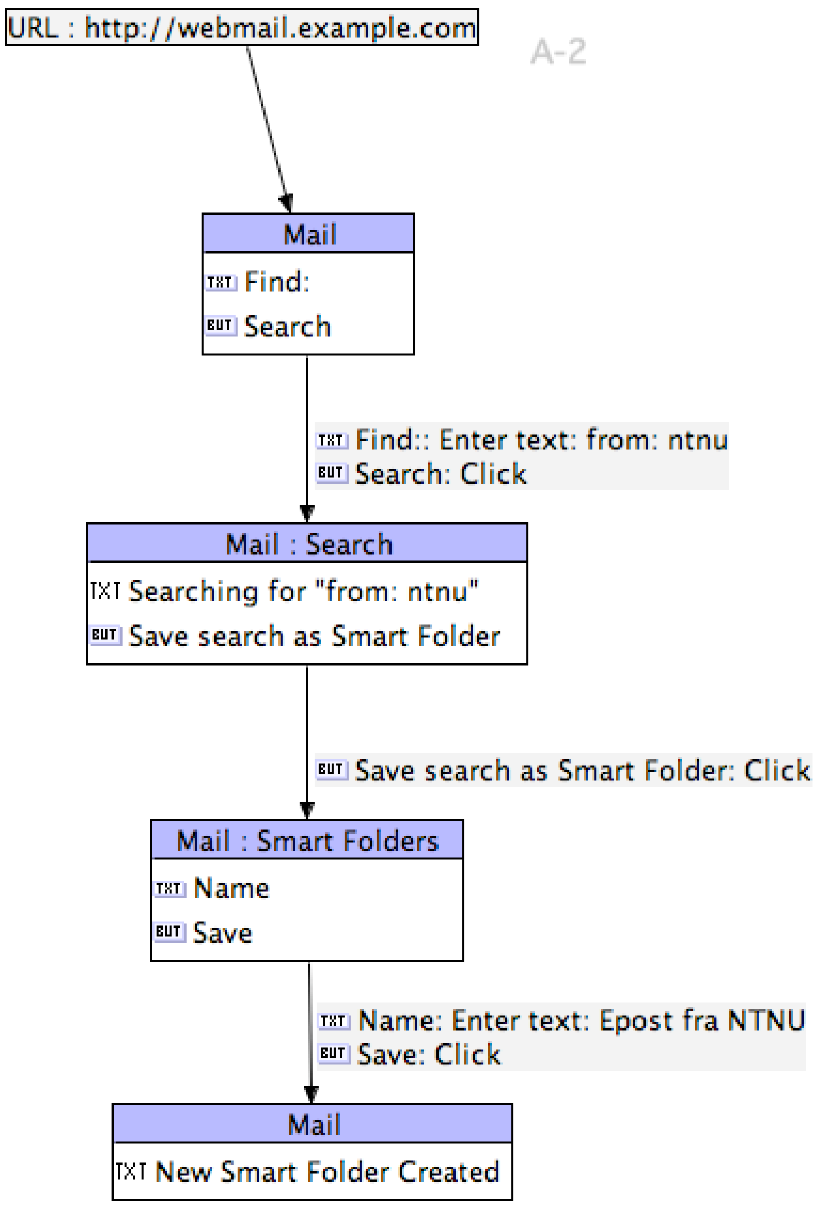

URL : http://webmail.example.com
A-2
Mail
TXT Find:
BUT Search
TXT Find:: Enter text: from: ntnu
BUT Search: Click
Mail : Search
TXT Searching for "from: ntnu"
BUT Save search as Smart Folder
BUT Save search as Smart Folder: Click
Mail : Smart Folders
TXT Name
BUT Save
TXT Name: Enter text: Epost fra NTNU
BUT Save: Click
Mail
TXT New Smart Folder Created

Figure C.1: Screenshot of the test presented in exercise A2

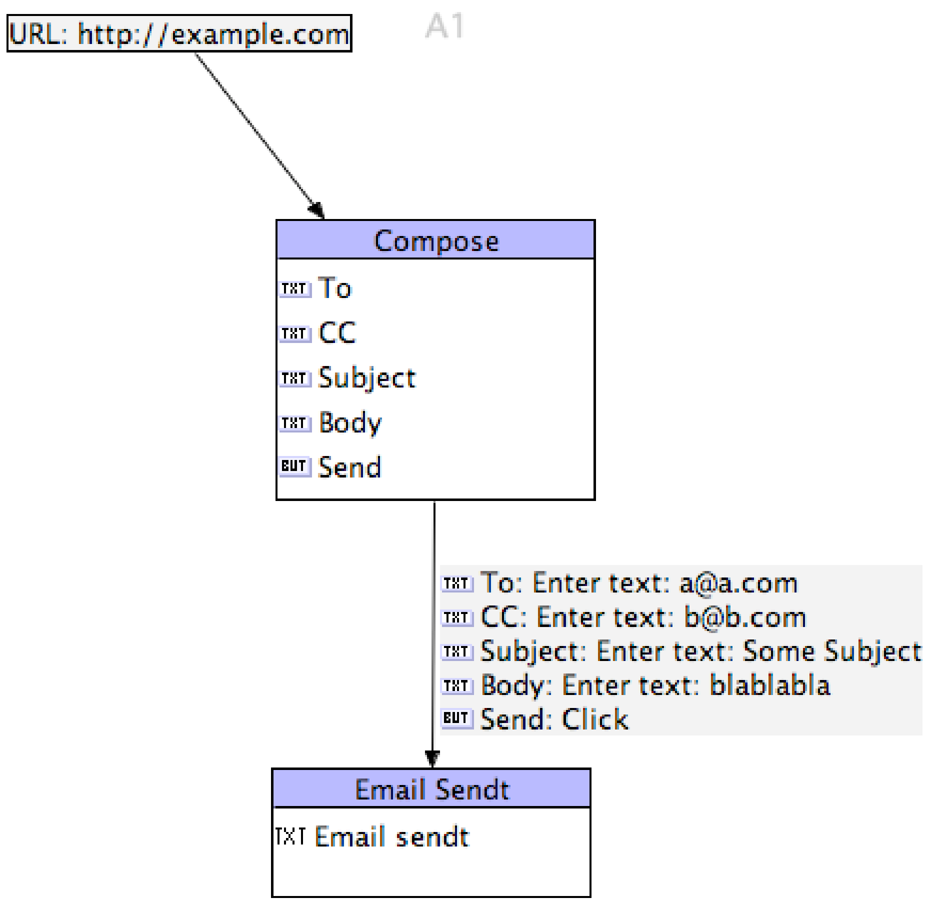
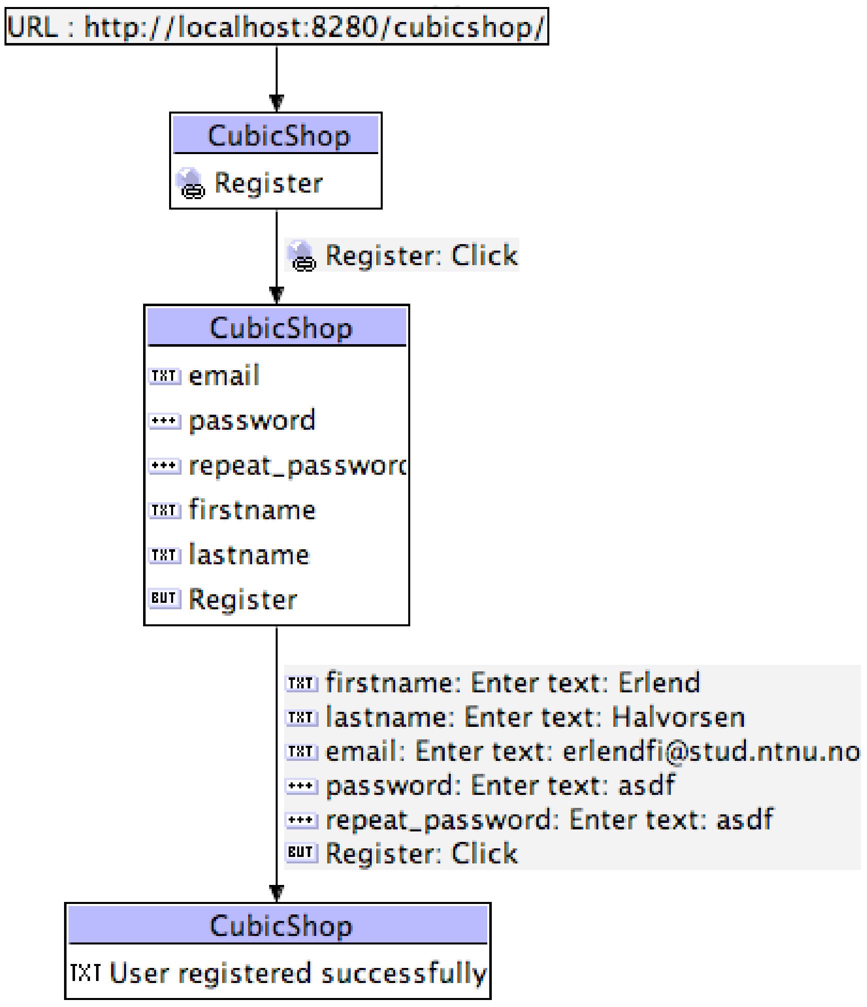

URL: http://example.com
A1
Compose
TXT To
TXT CC
TXT Subject
TXT Body
BUT Send
TXT To: Enter text: a@a.com
TXT CC: Enter text: b@b.com
TXT Subject: Enter text: Some Subject
TXT Body: Enter text: blablabla
BUT Send: Click
Email Sendt
TXT Email sendt

Figure C.2: Correct solution to exercise A1



TXT firstname: Enter text: Erlend
TXT lastname: Enter text: Halvorsen
TXT email: Enter text: erlendfi@stud.ntnu.no
••• password: Enter text: asdf
••• repeat_password: Enter text: asdf
BUT Register: Click

Figure C.3: Correct solution to exercise B

Appendix D

Questionnaire Responses

Below are the written responses to the exercises in the questionnaire presented in appendix C. All answers were given in norwegian. The rest of the answers are shown in figures 14.2 and 14.3 on pages 72 - 73. The exercises answered in CubicTest can be found in the accompanying attachment.

D.1 Subject #1

Linje: Datateknikk
År: 2

Exercise A2
- Åpner en mail-side
- Skriver inn søke teksten (Finn mail fra ntnu), og trykker søk
- Søket blir lagret i mappe, for raskt å kunne finne siden/ssidene senere

Exercise 3 - "Hva er formålet med CubicTest?"
Formålet er å gi kunden/brukeren mulighet for å teste funksjonalitet ved et system, før noe kode er skrevet. Ved å bruke "CubicTest" kan kunden selv danne seg et bilde av hva han ønsker seg.

Exercise 7 - "Forklar følgende funksjoner i CubicTest"

- **Page/State**
  Dette representerer en webside.
- **Common**
  Dette er en side brukt av hele systemet

- **Context**
  -
- **User Interaction**
  Denne pilen fører brukeren (ved klikk på hyperlinken pilen er relatert til) til siden pilen peker på

Exercise 8 - "Var det funksjoner eller konsepter i Cubic som du synes var spesielt vanskelig å forstå?"
-

D.2 Subject #2

Linje: Datateknikk
År: 2

Exercise A2
Søker etter mail fra ntnu, og lagrer søket

Exercise 3 - "Hva er formålet med CubicTest?"
Skal brukes til akseptansetesting. At kunden selv skal kunne lage tester som hanselv skal bruke for å kunne verifisere at programmet innfrir kravene som er satt.

Exercise 7 - "Forklar følgende funksjoner i CubicTest"

- **Page/State**
  En webside eller en tilstand i programmet. En side kan ha flere tilstander.
- **Common**
  Brukes hvis man vil ha noen standardlinker som skal gå igjen på flere sider.
- **Context**
  "Spesifiserer" en del av siden, eller lager grupperinger på siden
- **User Interaction**
  Setter en hendelse på en knapp. ved trykk kan man forandre tilstand

Exercise 8 - "Var det funksjoner eller konsepter i Cubic som du synes var spesielt vanskelig å forstå?"
Add User Interaction, brukte add connection i stedet, visste ikke hva jeg skulle trykke på i tilstanden for å sette user-interaction.

D.3 Subject #3

Linje: MTIØT-DK
År: 2

Exercise A2
-

Exercise 3 - "Hva er formålet med CubicTest?"
Å la kunder (brukere) selv kunne designe sine krav (jf. user stories), samt tester til disse. Er ment å være et verktøy for å teste (og utvikle krav?) til webapplikasjoner.

Exercise 7 - "Forklar følgende funksjoner i CubicTest"

- **Page/State**
  En tilstand i applikasjonen, f.eks "logg inn"
- **Common**
  Et element som forekommer flere ganger, f.eks. en meny.
- **Context**
  En spesifik del av en side. (F.eks. et skjema)
- **User Interaction**
  Hva brukeren gjør (for å komme fra en state til en annen).

Exercise 8 - "Var det funksjoner eller konsepter i Cubic som du synes var spesielt vanskelig å forstå?"
Det meste var rimelig enkelt å sette seg inn i. Kanskje testene kunne forklart litt bedre hva man skulle "asserte"

D.4 Subject #4

Linje: Datateknikk
År: 2

Exercise A2
Søker først etter eposter sendt fra ntnu. Så blir søket lagret som en smart folder.

Exercise 3 - "Hva er formålet med CubicTest?"
Å teste om en internettside fungerer riktig

Exercise 7 - "Forklar følgende funksjoner i CubicTest"

- **Page/State**
  Her kan man sette opp registrasjonsskjema, tekst, mail, osv.

- **Common**
  Tror ikke jeg prøvde dette under testen
- **Context**
  -
- **User Interaction**
  Her må man oppfylle bestemte betingelseer for å komme til neste side, f.eks. passord, fylle ut skjema, trykke på en knapp.

Exercise 8 - "Var det funksjoner eller konsepter i Cubic som du synes var spesielt vanskelig å forstå?"
-

D.5 Subject #5

Linje: Datateknikk
År: 2

Exercise A2
Søker etter mail-adresse ved å søke på navn, så kommer det opp "forskjellige" lagringsknapper, og til sist blir det opprettet en ny folder "New Smart folder", som du får bekreftet på slutten at er opprettet.

Exercise 3 - "Hva er formålet med CubicTest?"
For kunde / bruker å teste hva som skjer

Exercise 7 - "Forklar følgende funksjoner i CubicTest"

- **Page/State**
  En slags overskrift
- **Common**
  -
- **Context**
  -
- **User Interaction**
  Hva som skjer, om en knapp blir trykket på eller om en check-box blir merket av

Exercise 8 - "Var det funksjoner eller konsepter i Cubic som du synes var spesielt vanskelig å forstå?"
Forsåvidt ikke, men det trengs nok mer tid en hva jeg fikk til å sette seg inn i det (hvertfall for meg).

Kanskje "common" og "context" var vanskelig, etter som jeg ikke kunne svare på hva det var.

D.6 Subject #6

Linje: Informatikk
År: 1

Exercise A2
Søker etter ntnu og legger det man finner i en egen mappe

Exercise 3 - "Hva er formålet med CubicTest?"
Teste brukeren: Går igjennom sjekk på hva kunden forventer skal skje.

Exercise 7 - "Forklar følgende funksjoner i CubicTest"

- **Page/State**
  -
- **Common**
  Går igjen for alle states / pages hvis man for eksempel ønsker en meny som skal være felles til alt, så kan den brukes.
- **Context**
  -
- **User Interaction**
  Går igjennom hva som skal overføres til neste state

Exercise 8 - "Var det funksjoner eller konsepter i Cubic som du synes var spesielt vanskelig å forstå?"
Ikke svært vanskelig, men vanskelig å lære seg noe og bruke det uten problemer den første tiden. Tar litt tid å komme seg inn i akkurat hva som skjer mellom alle Page / state.

D.7 Subject #7

Linje: Datateknikk
År: 2

Exercise A2
Et søk kan lagres som en "smart folder". Da vil søket vises som en "mappe" der alle e-poster som passer med søket vises i den mappen.

Exercise 3 - "Hva er formålet med CubicTest?"
Å teste web-applikasjoner gjennom ulike metoder, som ved å legge opp siden i CubicTest

og deretter kjøre verifisering på en eksisterende side, eller å la CubicTest ta opp interaksjonssekvenser som så vises som et tilstandsdiagram i CubicTest.

Exercise 7 - “Forklar følgende funksjoner i CubicTest”

- **Page/State**
  Representerer en tilstand ewb-applikasjonen er i, det vil si en side som vises for brukeren.
- **Common**
  Et element som er felles for flere sider / tilstander, for eksempel en meny.
- **Context**
  En gruppering av elementer, for eksempel et nytt skjema.
- **User Interaction**
  En aktiv handling fra brueren, f.eks. å skrive inn tekst eller trykke på en knapp.

Exercise 8 - “Var det funksjoner eller konsepter i Cubic som du synes var spesielt vanskelig å forstå?”
I sidebyggingen var ikke funksjonen for å legge til brukerinteraksjon så enkel å forstå etter å ha brukt de andre elementene, som tekstfelt og knapper. Disse dras inn i diagrammet, men brukerinteraksjonen dras fra tilstand til tilstand.

D.8 Subject #8

Linje: MTING
År: 2

Exercise A2
søker gjennom en mailbox
legger resultatet inni en folder som kan gies et nytt navn
Bekreftelse på at folderen er opprettet

Exercise 3 - “Hva er formålet med CubicTest?”
Å gjøre det enkelt for programmereren å skjønne hva kunden ønsker, ved å la kunden sette opp testen selv

Exercise 7 - “Forklar følgende funksjoner i CubicTest”

- **Page/State**
  En side, som brukeren ser
- **Common**
  Det som er felles for flere sider

- **Context**
  vet ikke
- **User Interaction**
  Hva som får brukeren fra den ene siden til den neste

Exercise 8 - "Var det funksjoner eller konsepter i Cubic som du synes var spesielt vanskelig å forstå?"
Surra mellom connection og user interaction. Programmet hadde vært veldig enkelt om man hadde hatt en hjelp-funksjon å ty til. Mye å huske når det bare er en intro å forholde seg til.

D.9 Subject #9

Linje: MTDT
År: 2

Exercise A2
Lager en mappe hvor all epost fra NTNU (avsender) legges i mappen "Epost fra NTNU"

Exercise 3 - "Hva er formålet med CubicTest?"
Formålet er å teste webapplikasjoner. De som lager kravene til applikasjonen (kunden), skal kunne lage tester selv, og det skal fungere å kjøre disse testene.

Exercise 7 - "Forklar følgende funksjoner i CubicTest"

- **Page/State**
  Tilstand som websiden er i. Kan være en tilstand per vindu, men og flere tilstander i samme.
- **Common**
  Common er ting som flerre vinduer har til felles, for eksempel en meny. Da kan man lage en common istedet for å lage disse feltene i hver eneste Page/State.
- **Context**
  Context er noe man fyller ut, som et skjema.
- **User Interaction**
  Knapper og linker man trykker på, som tar en fra en tilstand til en annen.

Exercise 8 - "Var det funksjoner eller konsepter i Cubic som du synes var spesielt vanskelig å forstå?"
Nei

D.10 Subject #10

Linje: MTDT
År: 2

Exercise A2
Program for å søke. Lagrer eposter i en ny katalog. Eposter som matcher søkeordet.

Exercise 3 - "Hva er formålet med CubicTest?"
Lage webapplikasjoner ved hjelp av et mer grafisk GUI. Gjøre det enklere å lage forms.

Exercise 7 - "Forklar følgende funksjoner i CubicTest"

- **Page/State**
  Skal forestille en webside. Lager en Page for hver side. Kan også være et form som det er flere av på en side.
- **Common**
  En felles meny man kan linke til mange sider.
- **Context**
  -
- **User Interaction**
  Hva som skal skje når man gjør en handling

Exercise 8 - "Var det funksjoner eller konsepter i Cubic som du synes var spesielt vanskelig å forstå?"
Nei, men var litt vanskelig å få oversikt over funksjoner.

D.11 Subject #11

Linje: MTKOM
År: 2

Exercise A2
Den finner alle mail hvor avsender er "ntnu". Så lagrer den disse i en "smart folder" som de her kaller "Epost fra NTNU". Så får man beskjed om at den er lagret.

Exercise 3 - "Hva er formålet med CubicTest?"
Formålet er at kunden kan med dette programmet vise "programmereren" hvilke funksjonaliteter han ønsker å ha.

Exercise 7 - "Forklar følgende funksjoner i CubicTest"

- **Page/State**
  Det her ser jeg på som en side eller "frame". Inholdet i denne viser hva som blir vist på siden.
- **Common**
  En meny f.eks. som går igjen på flere Page / States.
- **Context**
  Begynnelse av en "form", her kan brukeren av applikasjonen fylle inn egne verdier.
- **User Interaction**
  Hvordan man kommer seg fra en Page/State til en annen

Exercise 8 - "Var det funksjoner eller konsepter i Cubic som du synes var spesielt vanskelig å forstå?"
Egentlig ikke, hadde jeg hatt veiledningen ved siden av meg tror jeg at jeg skulle klart det meste.
"Record" funksjonen var genial!

D.12 Subject #12

Linje: Datateknikk
År: 2

Exercise A2
Brukeren utfører et søk og kan lagre søket som en mappe som gjør at dersom ny epost kommer som matcher søket så havner det i denne mappa også. Her filtreres epost fra ntnu i fra-feletet.

Exercise 3 - "Hva er formålet med CubicTest?"
Formålet er å la kunder enkelt skrive tester for det systemet de ønsker. Kunder kan sette opp hva for elemetner de vil ha på forskjellige sider og hvordan en bruker kommer seg fra én side til en annen.

Exercise 7 - "Forklar følgende funksjoner i CubicTest"

- **Page/State**
  En enkelt nettside
- **Common**
  Element som går igjen over flere page/state
- **Context**
  Et element? La ikke merke til at ordet "context" dukket opp noe sted.

- **User Interaction**
  Hva bruker må gjøre for å gå mellom sider/tilstander.

Exercise 8 - "Var det funksjoner eller konsepter i Cubic som du synes var spesielt vanskelig å forstå?"
Ikke akkurat, men når en skulle endre en overgang så måtte jeg lete for å finne den før jeg til slutt oppdaga den var rett under "hovedbildet".

D.13 Subject #13

Linje: MTING
År: 2

Exercise A2
Lagrer treff på epost fra ntnu i en mappe.

Exercise 3 - "Hva er formålet med CubicTest?"
Gjøre det enklere for brukerne å lage tester

Exercise 7 - "Forklar følgende funksjoner i CubicTest"

- **Page/State**
  Er de forskjellige "sidene" f.eks. på en side registrerer man seg, på en annen får man bekreftet registreringen.
- **Common**
  Litt usikker, men den kobles til alle Page/State
- **Context**
  Innholdet i en page/state
- **User Interaction**
  Kobler to Page / State sammen

Exercise 8 - "Var det funksjoner eller konsepter i Cubic som du synes var spesielt vanskelig å forstå?"
Nei, egentlig ikke... Med litt mer tenking og utforking hadde jeg kanskje skjønt hvordan alt fungerer sammen. Slik som Common -> state, osv.

Bibliography

[1] Quirksmode - dom compatibility. http://www.quirksmode.org/dom/compatibility.html. Visited April 24th 2007.

[2] Document object model (dom) level 2 specification. http://www.w3.org/TR/DOM-Level-2-Core/, 1999. Visited April 11th 2007.

[3] James Bach Cem Kaner and Bret Pettichord. *Lessons Learned in Software Testing*. Wiley Computer Publishing, 2002.

[4] Martin Höst Magnus C. Ohlsson Björn Regnell Claes Wohlin, Per Runeson and Anders Wesslén. *Experimentation in Software Engineering - An Introduction*. Kluwer Academic Publishers, 2000.

[5] David Flanagan. *JavaScript - The Definitive Guide*. 2002.

[6] Grig Gheorghiu. A look at selenium. *Better Software*, October:38–44, 2005.

[7] J Grudin. Utility and usability: Research issues and development contexts. *Interacting with Computers*, 4:209–217, 1992.

[8] Raine Kauppinen. Testing framework-based software product lines, 2003.

[9] Jonathan Kohl and Paul Rogers. Watir works. *Better Software*, April:40–45, 2005.

[10] C. Larman and V.R. Basili. Iterative and incremental development: A brief history. *Computer*, June:47–56, 2003.

[11] Jon Reinert Myhre and Elena Kalitina. Cubictest usability, 2006.

[12] Patrick O'Connor. Neglect testing at your own peril. *IEEE Spectrum*, July:18, 2001.

[13] Stine Lill Notto Olsen and Kjersti Loe. Automatisert testing av dynamisk html, 2006.

[14] Stein Kåre Skytteren and Trond Marius Øvstetun. Autat - automatic acceptance testing of web applications, 2005.

[15] Steven Splaine and Stefan P. Jaskiel. *The Web Testing Handbook*. STQE Publishing, 2001.

[16] Tor Stålhane. Spiq: Etablering av måleplaner. *SPIQ notat*, June, 1998.

[17] Rini van Solingen and Egon Berghout. *The Goal/Question/Metric Method: a practical guide for quality improvement of software development.* McGraw-Hill Publishing Company, 1999.