NTNU
Det skapende universitet

Automatisert testing av dynamisk HTML

Kjersti Loe
Stine Lill Notto Olsen

Master i datateknikk
Oppgaven levert: Juni 2006
Hovedveileder: Tor Stålhane, IDI

Norges teknisk-naturvitenskapelige universitet
Institutt for datateknikk og informasjonsvitenskap

Oppgavetekst

Web-sider blir stadig mer dynamiske. Sider med enkel forretningslogikk i JavaScript og muligens AJAX er vanlige. For eksempel velger man "Oslo" i dropdown 1, så skal "Trondheim" og "Bergen" vises i dropdown 2.
Slike dynamiske web applikasjoner er vanskelig å teste automatisk. De blir derfor ofte testet manuelt, og det er en dyr og tidkrevende prosess hvis det er en stor og kompleks web applikasjon. Det finnes test-rammeverk for automatisert testing av dynamisk HTML, men disse krever en god del teknisk forståelse for å kunne skrive tester.
Det medfører at det er utviklerne som må skrive testene fordi det er for vanskelig for de som "eier" funksjonaliteten (kunden) å skrive testene selv. Kunden kan da heller ikke bruke testverktøyet til f.eks å spesifisere funksjonalitet, samt lage tester for feil de selv finner.
Oppgaven består av følgende:

1. Utredning

- Hva er vanlige JavaScript / AJAX skript på nettsider i dag?

- Finnes det test-rammeverk som er skikket til å teste dette?

2. Utvikle konsepter for bruk i testing

- Hvordan skal tester representeres?

- Hvordan gjøre det lett og uttrykksfullt for en kunde å skrive tester?

3. Utvikle en egen open source editor for disse konseptene som legges ut på boss.bekk.no (BEKK Open Source Software). Denne editoren bør ha muligheten til å eksportere til et testrammeverk som kan eksekverer testene.
Det bør bygges på verktøyet "AutAT" (boss.bekk.no/autat) som brukes til mer tradisjonell testing av webisder (f.eks sideflyt og utfylling av skjemaer).
Oppgaven gis i samarbeid med Bekk Consulting AS, Steria AS og Mesan AS, og passer fra en til tre studenter. Rapporten bør skrives på engelsk.

Faglærer: Tor Stålhane (stalhane@idi.ntnu.no)

Medveileder(e): Christian Schwarz (BEKK), Trond M Øvstetun (Mesan).

Oppgaven gitt: 20. januar 2006
Hovedveileder: Tor Stålhane, IDI

Abstract

Today, customers often perform acceptance testing of dynamic web-applications manually. This can be costly and time-consuming if the web-application is complex or if testing has to be performed often. Agile development methods are characterized by short development iterations, test-driven development and regression testing. As these methods are becoming more popular, the need for automated acceptance tests have increased. In these agile methods, it is the customers (the owners of the funcionality in the application) who should create and maintain the acceptance tests. However, there is a lack of tool support with a high abstraction level that can assist customers (non-technical/business-side persons) to create these tests. The result is that automated acceptance tests are usually created and maintained by developers. This results in that the customer cannot use tests to specify new functionality.

The original AutAT was developed in a master thesis the spring of 2005 to enable non-technical users to create automated acceptance tests for static HTML, in a graphical user interface. The conclusion in their master thesis indicates that AutAT has better usability, efficiency and quality, compared to other state-of-the-art test frameworks. We have in this master thesis continued the development of AutAT to support the creation of tests for dynamic web-applications created with Ajax- and JavaScript. The new version of the AutAT tool has been thoroughly tested to assess its usability from a customer's perspective, as well as its usefulness in software development projects. The assessment showed that the new version of AutAT can be used by customers after proper training, but that the tool is most useful when used in cooperation between developers and customers.

Keywords: AutAT, Automatic Acceptance Test, Web Application, Testing Tool, Test Driven Development, Ajax, JavaScript.

Preface

This master thesis documents the work done by Stine Lill Notto Olsen and Kjersti Loe, in the spring of 2006. The thesis is related to the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU), in addition to Bekk Consulting AS, Steria AS and Mesan AS. The project is a study of an acceptance test tool (AutAT), and focus on automated acceptance testing of dynamic web-pages.

We would like to thank our supervisors, Tor Stålhane at IDI, Christian Schwarz at Bekk consulting, Stein Kåre Skytteren at Steria and Trond Marius Øvstetun at Mesan, for their involvement and guidance during our work this semester. We really appreciate their time, feedback and efforts helping us.

We would also like to thank all participants involved in the tests and interviews during the assessment of AutAT. Both the non-technical users and developers familiar to Watir used their time to give us valuable feedback on the tool. Special thanks to the representatives from the companies in Trondheim and Oslo, for interesting conversations and useful information. Therefore thanks to:

- Skule Johansen, Kantega
- Knut Bliksås, Fundator
- Odd Martin Solem, Abeo
- Trond Johansen, Proxycom
- Jan Thoresen, Bekk

Kjersti Loe Stine Lill Notto Olsen

Trondheim, June 16, 2006

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The intention of AutAT is to enable technical and non-technical users to create automated acceptance tests. AutAT can be used by customers of software development projects to define required functionality, specify changes and report bugs in the web-application. Automated tests can in addition lead to advantages for customers. We therefore believe that if AutAT can support testing common dynamic web-applications, it could be used in software development projects, which would benefit both the customers and developers.

1.1 Background and Motivation

When web-applications are tested, it often includes testing the correctness of forms, links, textual content, etc., and is traditionally performed manually. Testing a web-application manually, by i.e. clicking on links, submitting forms, etc., can be time consuming and prone to human errors [18]. Traditionally, web-applications were developed with the waterfall model, in which testing was performed once to prove an application's correctness in the end of the development process. Testing an application once involved a high risk in case the software did not pass the final tests, or if the customers' requirements had changed during development. This risk was decreased when software instead were developed in modules, since each module were then separately tested, and later compound into an application and tested as a whole. Now, the use of eXtreme Programming (XP) and other agile development methods are becoming increasingly popular among software developers. Most of these development methods can be characterized by short development iterations, continuous integration and Test Driven Development (TDD) [27].

In TDD, tests are written before the application code [27]. These tests are used continuously during development to check that new code works correctly, and that the old code has not been damaged by adding new. It is therefore important that the tests are automated so they can run repeatedly without much additional effort, and efficiently test large parts of the applica-

tion. In difference, when testing an application manually, this is often done to reveal errors on only the parts of the application with the largest possibility of being affected by changes in the code, to reduce the effort used on testing. Automated tests provide more thorough testing of web-applications because they in addition to reveal errors, also increase the confidence in the developed software.

When running automated tests continuous during agile development, it requires that the tests are always up to date with the changes in the application. The tests can therefore be used as documentation of the functionality in the application. This is an advantage because changes in an application results in that only the tests need to be updated. With manually performed testing, both the written test specifications and the application documentations usually must be synchronized.

Customers are responsible for the final approval of an application. This is done through acceptance testing, where the customer verifies that the application meets their requirements. There exist tools and frameworks that can help in the creation of acceptance tests. However, most of them require several types of programming skills. Tools with better usability, especially for non-programmers (such as customers), can help customers to create automated acceptance tests. Stein Kåre Skytteren and Trond Marius Øvstetun created a tool called "AutAT - Automatic Acceptance Testing of Web Applications" as a master thesis in the spring of 2005 [30]. AutAT is a tool for designing tests in a way that is easy for customers to understand. The tool allows users to specify web-pages in a GUI interface, and convert them into test-scripts that can run on already existing test frameworks. They showed that using AutAT would increase quality and efficiency when creating automated acceptance tests for web-applications, compared to "state-of-the-art" open source tools [30].

Customers can use AutAT to create automated acceptance test, which could improve the specification phase in software development project. The acceptance tests created in AutAT could be used as a part of the requirement specification since they specify the functionality requested by customers. Customers could also report bugs they find and specify changes in the application by creating test cases for respectively, the input resulting in the exception and the requested functionality. The AutAT tests could also be useful for customers to insure that their required functionality works correctly. These activites would increase the customers involvement in the development projct and therefore decrease the possibility for misunderstandings between customers and developers.

1.2 Problem Definition

The current version of the AutAT tool, supports only the creation of test scripts for static HTML (Hyper Text Markup Language) with a simple request-reply architecture. However, most web-applications today include some form of dynamic content. JavaScript is a commonly used web-technology, and we believe that Ajax will become popular in the future. The aim of this master thesis is therefore to develop a tool that can be used by customers for creating and maintaining automated acceptance tests for testing applications containing Ajax- and JavaScripts.

An important part of this thesis is to explore new concepts for the graphical user interface in AutAT. These concepts must enable a customer to simply and expressively model dynamic content in web-applications. In addition to the graphical changes in the AutAT tool, a new exporter must be created to convert AutAT models with dynamic features into automated test scripts. Another important part of this thesis is therefore to find a test framework that supports testing Ajax- and JavaScript, and address the challenges with making AutAT automatically create test-scripts for dynamic HTML.

1.3 Project Context

This project is an extension of the master thesis [30] carried out in cooperation with the Software Engineering Group, which is a part of the Department of Computer and Information Science at the Norwegian University of Science and Technology (NTNU) in Trondheim. This thesis is related to BEKK Consulting AS, Steria and Mesan.

1.4 Project Outline

The rest of this document is organized as follows:
Chapter 2 "Research Agenda", introduces the goal of this Master Thesis. It describes the questions we need to ask and the process planned to achieve this goal.

Chapter 3 "Prestudy: Testing", presents the field of testing to the reader and describes common practice of testing in five companies in Norway.

Chapter 4 "Technical Prestudy", describes the background information for the new version of the AutAT tool. It defines a dynamic web-page and explores dynamic web technologies. It also presents state-of-the-art of existing testing frameworks.

Chapter 5 "Own Contribution", presents the requirements specification of the new version of the AutAT tool. It also describes the different concepts

that we evaluated before we decided on a solution.

Chapter 6 "Design and Implementation", presents the changes in the design and implementation in the new version of the AutAT tool. In addition, a user documentation of the AutAT tool is presented to show how tests can be created.

Chapter 7 "User Testing and Results ", presents the tests we have performed on different users. It describes how the tests were performed and the results from these tests.

Chapter 8 "Evaluation and Discussion", presents the evaluation of the AutAT tool, based on tests and interviews.

Chapter 9 "Conclusion", contains the conclusions drawn from evaluating the AutAT tool.

Chapter 10 "Further Work", contains suggestions for further development and research of the AutAT tool.

Chapter 2

Resesarch Agenda

This chapter presents the focus of our work, by describing the goal of this master thesis and research questions. It also describes the research methods during our work and an overall work plan for this thesis.

2.1 Goals and Research Questions

Researching state-of-the-practice of testing shows a demand for tools that support customers when creating automated acceptance tests for dynamic web-applications. To our knowledge there do not exist tools for this purpose that can be used before the application is developed, and in addition are based on a graphical user interface (GUI). GUI based applications are usually easier to use and learn for customers, which often do not have a technical background. The goal of this thesis is therefore:

to create a tool that can be used by customers in software development projects and make acceptance testing of dynamic web-pages simple and more efficient.

To create this tool, we had two options. We could either extend AutAT to support dynamic web-applications, or create a new tool from scratch. AutAT has been created after thorough research, and it has received positive feedback from software developers after presentations, i.e. in XP-Meet Up (January 2006). In addition, there were few restrictions on how we could change the AutAT-editor and we saw several possibilities. We therefore decided to continue on the development of AutAT.

When continuing the development of AutAT, it is important to understand the process where it is meant to be used, and the needs of the intended users. It is also important to investigate different concepts to find out how dynamic web-applications can be modeled in a graphical user interface. This to be sure that dynamic content can be modeled in the new version of AutAT. For

a tool to be taken into use, it is important that it has good usability and is useful. The intention of AutAT is that it can be used by customers to create and maintain acceptance tests [30]. However, AutAT can be used to create both acceptance tests and system tests. We will therefore investigate the usability and usefulness of the new version of AutAT, for customers as well as developers.

The research questions to investigate in order to reach our main goal, are therefore:

1. How is testing, and especially acceptance testing, of dynamic web-pages performed in companies today?
2. How can we represent dynamic web-applications created with Ajax- and JavaScripts, simple and expressive in AutAT?
3. Is AutAT useful for companies when developing web-applications?
4. How is the usability of AutAT? Where usability is defined as "the effectiveness, efficiency, and satisfaction with which specified users can achieve specified goals in a particular environment"[1].

2.2 Research Methods

A brainstorming was performed to create new concepts for how to make the AutAT-editor best suitable for modelling tests for dynamic web-pages. Each concept was then evaluated against common web-pages to investigate the concepts restrictions and applicability.

A study of how testing is done in companies was preformed to understand the process of where and by whom the tool is meant to be used.

A literature study was preformed in order to gain knowledge about how software applications are tested. In addition, we have explored web-technologies used on common dynamic web-pages, and challenges these technologies introduce for automated tests. We have also studied typical dynamic functionality in web-pages and different open-source test frameworks.

Design and implementation of a new version of AutAT, was preformed in order to do more research on the usefulness and applicability of the tool.

Interviews and questionnaires was done in several companies in Trondheim and Oslo to get feedback on the new concept.

2.3 Project Process

The Gantt diagram in figure 2.1, shows the overall process of the development of our thesis. We started with a literature study of the initial version of

[1] http://www.sqatester.com/glossary/

AutAT, and dynamic web-technologies, especially JavaScript and Ajax. We studied typical web-sites to find commonly used scripts that AutAT should support to be applicable in web-application development processes. While studying web-sites, we also had several brainstorming sessions to come up with concepts for how to best model dynamic web-applications in AutAT. These concepts where continuously evaluated during this process.

When we had decided upon a concept, we started on the design and implementation of the changes needed in AutAT. At the end of the implementation, when the main functionality had been implemented, we prepared interviews to get feedback on the usefulness of the new version of AutAT. We tested AutAT on several possible users, and interview them after the test to get feedback. A report was written during the whole project process.

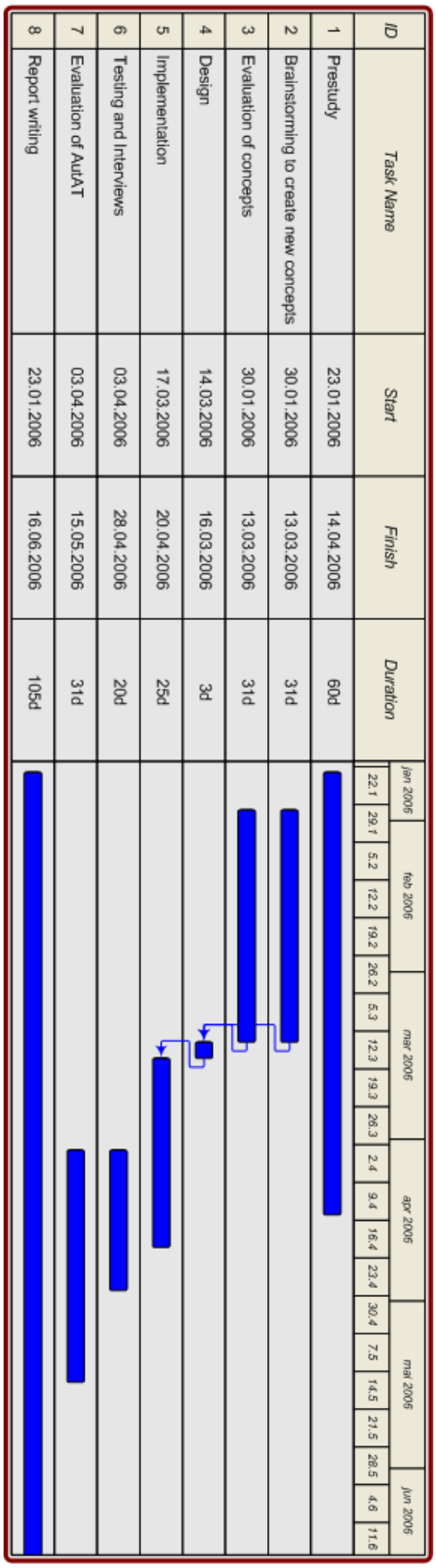

Figure 2.1: The Project Process

Chapter 3

Prestudy: Testing

Testing is an important part of the software development process. Testing is either done to detect errors, or to increase the confidence in the developed software [35]. When the objective is to detect errors, a successful test is one that exposes failures during the execution of a system. When the objective is to increase confidence, a successful test is one that verifies that a software system meets its intended specification.

Testing of a software system is performed at different levels, as is illustrated in the section 3.1 with the V-model. Section 3.2 explains regression testing, which is often used during development of web-application, and section 3.3 explains an evolutionary approach to development; the Test Driven Development. Both are environments where AutAT is meant to be helpful. To get an understanding of the process where AutAT is meant to be used, section 3.4explores how testing is done in a selection of companies in Trondheim and Oslo.

3.1 The V-model

To test an entire software system, tests on different levels are performed. The V model [29], shown in figure 3.1, illustrates the hierarchy of tests usually performed in software development projects. The left part of the V represents the documentation of an application, which are the Requirement specification, the Functional specification, System design, the Unit design. Code is written to fulfil the requirements in these specifications, as illustrated in the bottom of the V. The right part of the V represents the test activities that are performed during development to ensure that an application corresponding to its requirements.

Unit tests are used to test that all functions and methods in a module are working as intended. When the modules have been tested, they are combined and *integration tests* are used to test that they work together as a

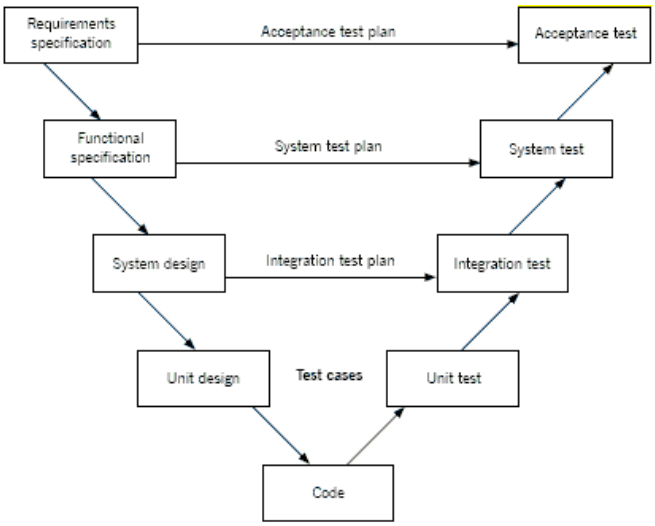

Figure 3.1: The V-Model [29]

group. The unit- and integration test complement the system test. *System testing* is done on a complete system to validate that it corresponds to the system specification. A system test includes checking if all functional and all non-functional requirements have been met. Unit-, integration- and system tests are developer focused, while acceptance tests are customer focused. *Acceptance testing* checks that the system contains the functionality requested by the customer, in the Requirement specification. Customers are usually responsible for the acceptance tests since they are the only persons qualified to make the judgment of approval. The purpose of the acceptance tests is that after they are preformed, the customer knows which parts of the Requirement specification the system satisfies.

3.2 Regression Testing

Regression testing, also referred to as verification testing, is the process of validating that modified software, or added software, does not introduce new errors into previously tested code [17]. It is a quality control measure to ensure that unmodified code has not been affected by maintenance of an error in the code, and that newly modified code still complies with the software. For these reasons, regression testing are often used to test software during development and maintenance [17]. Regression testing often requires that tests are run frequently, so automated test can be an advantage.

3.3 Test-Driven Development

Test-Driven Development (TDD) has sprung from agile development methods, such as eXtreme Programming (XP) [27]. In TDD, tests are written

before the application is developed. The development is done in short iterations, each with the creation of i.e. one unit-test and the corresponding application code [2], see figure 3.2. Regression testing is used, and if all tests run successfully, the code are refactored to ensure high quality code. Refactoring includes renaming variables and taking away redundant code. If the refactoring leads to that a test fails, then the code must be corrected until the test is passes. When all tests have passed, the developer can continue with a new iteration of creating a test and the corresponding application code. The tests usually specified in TDD are unit tests, integration tests and acceptance tests [30].

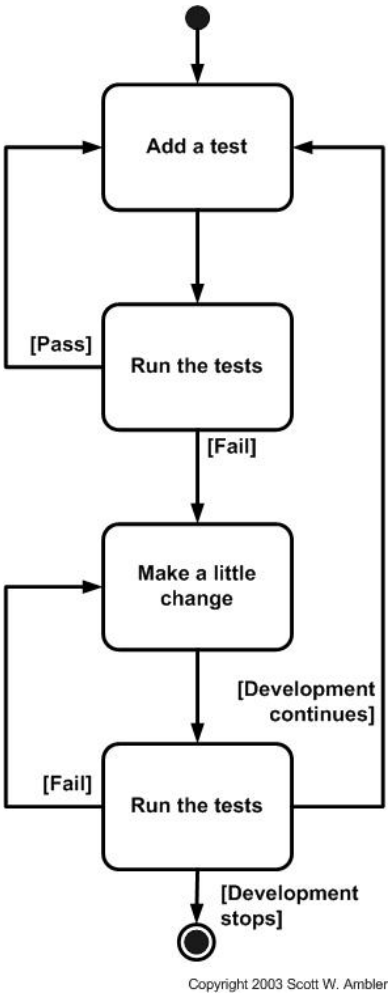

Figure 3.2: The Test-Driven Development Process [2]

When writing tests with the intention of executing every single line of code, as in TDD, full test coverage of the application code is achieved [2]. This is difficult to achieve with traditional testing, so TDD therefore provides de-

velopers with more confidence in that the software works as indented. TDD also gives documentation of the source code. In example, the unit tests can be a part of the technical documentation, because it shows how to use the application and which conditions are needed for it to work as intended. Acceptance tests could be a part of the requirements specification, because they define what the stakeholders in the project expect of the system. Unit- and acceptance tests can, for some developers and business stakeholders, cover the majority of documentation needed [2]. However, it is likely that these tests are not sufficient documentation for many developers and stakeholders.

One of the practices in TDD, is that unit tests are automated [2]. Acceptance tests should in addition be automated, because of the advantage of running them more frequent [27]. Developers often create one or more acceptance tests, to reflect the User Stories [1] written by customers [27]. This way, customers can after each development iteration verify and approve the developed software. The acceptance tests can also be run by the developers, during the development, to validate that the required functionality is created. These tests can in addition give an estimation of the general progress of the development of the application [30].

3.4 State-of-the-Practice of Testing

To investigate state-of-the-practice in testing , we have interwieved project managers from five companies in Trondheim and Oslo. The results can be found in appendix A. We will here summarize the current practices of software testing in web-applications project for all companies. The companies used different development methods, in addition to that different development methods were used on different projects in each company. Hence, they had different methods for testing their software. Some companies used a customized version of Test Driven Development (TDD), and others used TDD on some of the tests. Many companies developed software in modules and tested their code after it had been written. However, common for all the project in all companies were that a software systems were tested with unit-, integration-, regression-, system-, and acceptance tests.

Unit-, integration-, regression-, and system tests, were mainly written and performed by developers in all companies. In the companies where developers also wrote acceptance tests, these were used in system testing during development before it was delivered to the customer. In other companies, customers used the system tests written by developers, when they performed acceptance testing. A common opinion was that if customers had created automated acceptance tests before the development of an application started, these tests could then be run as system tests for developers during develop-

[1] User Stories are written by customers to specify tasks they want to perform in the developed software [27]

ment. Current practices for customers were to perform acceptance testing manually. This was often done by clicking through the web-site and following tasks on a written test-specification document. The test-specification document could be written by the software developers or by the customers, based on the customers' requirements. This document was usually written after the software was developed and often based on the system tests and system test data.

Common for all companies were that automated tests were often used with unit-testing. Some companies had in addition, used (or tried to use) tools to create automated tests for the functionality in the graphical user interface (GUI) of web-applications. However, most of the companies did not create automated tests for GUI, becuse they were time-consuming to keep updated when changes occured. Proxycom had tried a capture-replay[2] tool called Rational Robot, which resulted in high maintenance costs of keeping the tests updated. This because the tests needed to be updated every time changes occurred in a web-page, such as moving elements to different locations, changing text, etc. Bekk used Selenium in a current project, which is a scripting tool that access elements in the browser based on i.e. HTML-ids, values, etc. Selenium-tests is therefore not affected by changes in the location of elements and text in the browser. However, Selenium requires the user know the syntax of how to write tests. In addition, the user need to supply the HTML-id for i.e. buttons, as well as the actual value that are displayed in the browser.

Automated tests were in general, not used in the companies' projects. However, the project managers agreed on that automated tests were an advantage since they can be run with little effort. It is therefore easier to perform regression testing, as well as testing the whole web-application for faults after parts of it are changed. They also agreed that it was better if customers could create automated acceptance tests, because they are more likely familiar with users' needs than developers. They also meant that the current test-tools are difficult for customers to learn and use.

[2]Capture-replay tools are used to capture a users action in a web-application, which are replayed when testing the application [11].

Chapter 4

Technical Prestudy

This chapter explores the initial version of AutAT, which is the basis of this master thesis. It then defines a dynamic web site and presents common dynamic web technologies. This to investigate if AutAT can support several technologies, and hence, increase its usefulness. However, our focus is on Ajax- and JavaScripts, and therefore are these presented in more detail. Typical Ajax- and JavaScripts is presented to enable the new version of AutAT support commonly used scripts. In addition, this chapter presents the challenges with testing dynamic web-applications with Ajax- and JavaScripts, and presents state-of-the-art testing-frameworks.

4.1 AutAT

AutAT, described in [30], is an open-source project and is under continuous development. The initial version of AutAT supports creating tests for static HTML. The version presented here was downloaded on the 21st of January 2006 and is used as the basis for further development in this master thesis. Skytteren and Øvstetun built AutAT as an Eclipse-plugin, which has the advantage that it can be integrated in an Interactive Development Environment (IDE) used by many developers. It can also be created into an Eclipse Rich Client Platform (RCP), which enables developers to customize the layout and functionality in Eclipse to only include the AutAT plug-in. Thus, Eclipse can simulate a stand-alone client of AutAT. Another reason for why AutAT was built as a plug-in, was that it supports keyboard shortcuts and drag-and-drop functionality.

4.1.1 The User Interface

From a user's point of view, AutAT looks like a part of Eclipse, as shown in figure 4.1. The figure shows the User Interface with an example of a test. The User Interface structure in Eclipse is provided by a plug-in called *Workbench*, which is divided into the Navigator, the Outline, the Property Editor and the Editor.

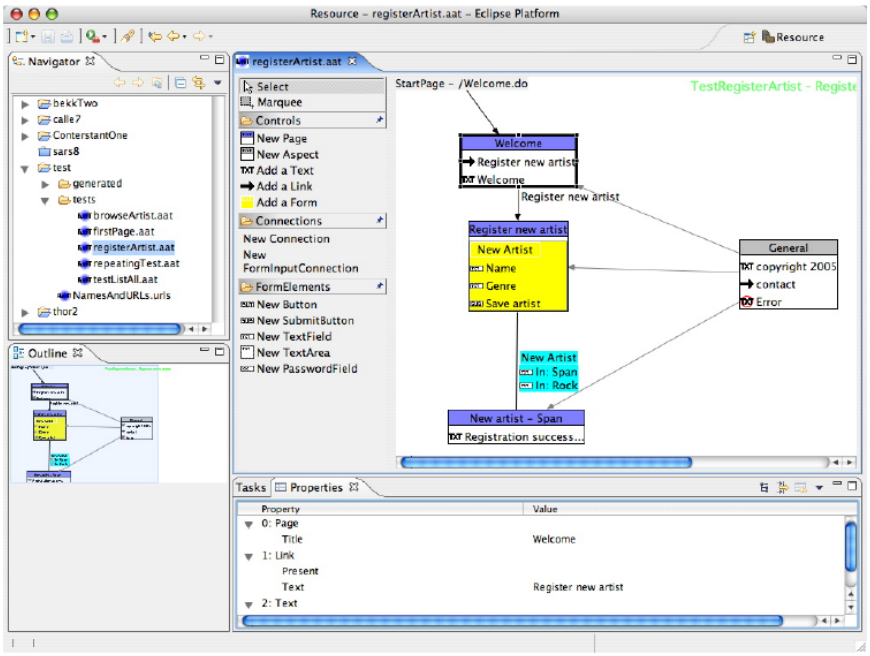

Figure 4.1: The AutAT User Interface

The Navigator shows the active projects and the files belonging to this project, sorted in a three structure. The Outline shows a document outline of the currently active editor. The Property Editor is available as an alternative to editing text directly in the editor window, and assigning property values to elements in the Editor. These views are not particular for AutAT, but a part of Eclipse. The Editor is in difference, particularly created for AutAT. It consists of a workarea on the right and a palette on the left, as seen in the top-right corner on figure 4.1. The palette consists of a set of elements that the user can drag into the workarea.

The user needs to define a starting point for a new test, which is the URL to the web-page where a test should start. The user can add types of elements, such as page, aspect, link, etc. to the test. Pages are shown with a blue header and represents a panel where other web-elements can be added. An aspect is modelled with a grey header and is similar to a page. However, an aspect contains the elements that are joint on the pages it is connected to. On the pages and aspects, the user can add page-elements, such as text, links and forms. When adding text, the user writes the full text that will be tested that exists in the web-page. When adding links the user writes the name of the link. The user can add a form to a page, which is represented with a yellow area on the page. The user can add form-elements such as buttons, textfields, etc. to the form. To navigate between two pages in a test, the user adds a connection between them. Depending of whether the new page

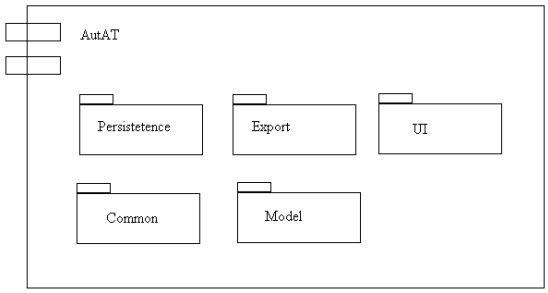

Figure 4.2: The main package structure of AutAT

is a result of pressing a link or connection is a link- or a form-transition, the user needs to provide information about the value of the link, or enter values for the fields in the form.

4.1.2 Architecture

The architecture of Eclipse was a restriction on the overall architecture of AutAT, because AutAT is a component within an Eclipse installation. In addition, some of the user stories created for AutAT [30], are supported by other plug-ins in Eclipse that AutAT has to relate to. For example, User Story 10 (US10) specifies that "several people must be able to work on the same project with its user stories and tests" [30], which is fulfilled by using the Eclipse Team Plug-in. The main package structure of the AutAT architecture is shown in figure 4.2, and consists of five high-level packages; Persistence, Export, Model, Common and UI.

The *Persistence package* handles tests and starting-points in an project in AutAT. The package is responsible for reading a test and its starting point from files, converting them from XML to objects, and performing the reverse operation. AutAT defines two file formats; the *.aat for files containing acceptance tests and the *.urls for names and URLs for defining the starting-points. The *Persistence package* is implemented using the Builder pattern, which divides the effort of creating a complex object between several objects, each responsible for a smaller part of the total.

The *Export package* transforms AutAT-tests from its object representation into a representation that is suitable for execution in a test framework.

The *Model package* contains the "value-objects" that are needed to save a test in AutAT. These objects are specified in a class model, which is similar

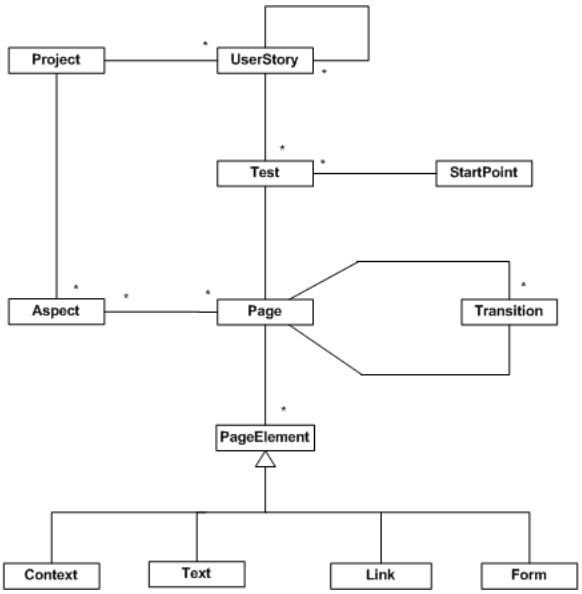

Figure 4.3: The Domain Model [30]

to the domain model in figure 4.3. The central classes of the *Model package* are the Test, URLStartPoint, Aspect, Page, Transistion, PageElement, Text, Link and Form. The Model package is used by the Persistence package to popluate objects with values, and by the UI package to present tests in the AutAT-Editor.

The *Common package* contains classes, such as exceptions, interfaces, resources and utils that are used by other packages.

The *UI package* is responsible for handling the Graphical User Interface (GUI) in AutAT. It consists of four packages, which are the Eclipse Specialization package, the Actions package, the Wizards package and the GEF package. The Eclipse Specialization package contains property pages for each project, and preference pages for the AutAT-plugin. An action represents a process, triggered by the UI, that must be performed i.e. to update the data model if the user has changed a property. The Wizard package is included to guide a user through performing specified tasks, such as the tasks necessary to get started. The GEF package contains the graphical test editor, which is used to create the workarea for the user and interactions the user can have with AutAT.

4.2 Dynamic Web Pages

The initial version of AutAT, described in section 4.1, supports creating tests for static web-sites. A web-site is a collection of web-pages that are related

through semantically content and syntactical links [3]. Static web-pages are fetched by simple request/response interactions with a server. There are two main definitions of dynamic web-pages depending on if it is generated on the server-side or client-side. Client-side dynamic pages are defined as web-pages that can react to user's input without sending requests to the Web server [6], such as pages containing JavaScript, Ajax, Java Applets, etc. Server side dynamic pages are defined as web-pages that are generated on the server based on a user's input, state of the database, etc [38]. This includes web-pages created with PHP, ASP, JSP (Java Server Page), Microsoft .net and CGI/Perl. We define a dynamic web-page as a page that can change on the client-side without reloading the browser, as a result of user input, date/time, server state, etc. We adopt the definition of web-application in [3]: "A Web application is a program that runs in whole or in part on one or more Web servers and that can be run by users through a Web site".

4.2.1 Dynamic HTML

The term often used on client-side dynamic web-sites are DHTML, Dynamic HTML [36]. DHTML is a set of technologies that use the object model (DOM), which provides developers with enhanced control of the elements on a web-page. With DHTML, it is possible to manipulate any page element at any time in a easy way with open and standards-based technologies. The technologies usually used to create DHTML is a combinations of HTML 4.0, Style Sheets and JavaScript. With HTML 4.0, all formatting can be moved out of the HTML document and into a separate style sheet, the Cascading Style Sheets (CSS) [36]. In HTML 4.0, the Document Object Model (DOM) is HTML DOM. The HTML DOM "defines a standard set of objects for HTML, and a standard way to access and manipulate HTML objects" [36]. JavaScript allows writing code to control all HTML elements. More about JavaScript is found in section 4.3 - JavaScript. With CSS, styles and layouts are separated from the document's structure and content.

4.2.2 Alternatives to DHTML

Macromedia has become popular with its Flash technology. Macromedia Flash is a multimedia graphics program for use on the Web [36]. It uses vector graphics to deliver content over the Web, which makes it suited for Internet. Vector graphics are graphics that can be scaled to any size without losing clarity or quality [36]. Flash is often used by designers and developers to create rich animations on the Web. These animations (or movies) appear as a part of the Web page and include a high level of interactivity. Flash have become popular because it has avoided the problem of browser incompatibilities by using installed client-side code rather than relying on scripts [20]. When Flash is downloaded, it has a TV-like appearance which makes it appealing to people. Flash is currently in the lead when building graphical interfaces that run in almost all clients, and its capabilities are still

growing.

Java Applets is also in the web application market. Although almost gone from most general Internet pages, applets are often used in the gaming world. At sites such as ”Yahoo” and ”AOL games interactively”, thousands or hundreds of people are playing games and that are powered by Java applets. Applets are required for the rapid, near-real-time calculations that exists in the games, which Flash scripting and DHTML are not fast and powerful enough for. An Applet is written in Java and is a program that is included in an HTML page. When using the Java technology, the browser can view a page that contains the Applet, and the Applets code is transferred to the clients and executed in the browser [23]. An alternative to Applets is Microsoft's ActiveX technology [20] . However, it has failed on the web because it has full access to the local machine, which gives downloaded code too much potential for abusing the machine.

4.3 JavaScript

JavaScript is a cross-platform, object-oriented, interpreted, lightweight and commonly used scripting lanuguage originally developed by Netscape [37] [34]. Microsoft later created JScript which is similar, but created to run on the Microsoft platform [9]. ECMAScripts were then created as a JavaScript standard [9]. We will use JavaScript as a collective term, for these different versions of the JavaScript technology since they only contain minor differences.

JavaScript can be seen as a hybrid between a ”markup” language and a programming language, where grammatical instructions are given to the computer [39]. In difference to programming languages, scripting languages tend to have simpler rules. JavaScript was designed to add interactivity to HTML pages and to provide programmatic control over objects in its environment on the World Wide Web [39] [37]. JavaScripts consist of lines of executable computer code that can execute without preliminary compilation and can be embedded directly into HTML pages [36]. JavaScripts can for example be used for [34]:

- Menus for navigation
- Form validation
- Popup windows
- Password protection
- Math functions
- Special effects with document and background

- Status-bare manipulation
- Messages
- Mouse-coursor effects
- Links

4.4 Asynchronous JavaScript and XML (AJAX)

In 2005, a new development technique that had the possibility of reducing the gap between web-based and desktop applications, rose in the form of AJAX [12]. AJAX stands for "Asynchronous JavaScript and XML", and is a label for interfaces that have increased richness, responsiveness and interactiveness. AJAX is a new way of using already existing technologies [12], such as:

JavaScript: Ajax applications, including the Ajax engine, is written in JavaScript [5]. The Ajax engine holds the Ajax application together. It is responsible for handling the user interaction, rendering the user interface by manipulating the DOM, as well as communicating with the server [5]. For these reasons, JavaScript is the central part of Ajax.

Cascading Style Sheets (CSS): CSS provides the look and feel in Ajax applications. Visual styles for web pages can with CSS, be defined and reused, and it holds the visual styling consistent. However, in an Ajax applications the styling of the user interface can interactively be modified [5].

Document Object Model (DOM): The DOM holds a structured scripting of web pages, and allows an Ajax application to modify the user interface by redrawing parts of the page [5].

XMLHttpRequest object: The XMLHttpRequest object allows asynchronous communication with the web server [5].

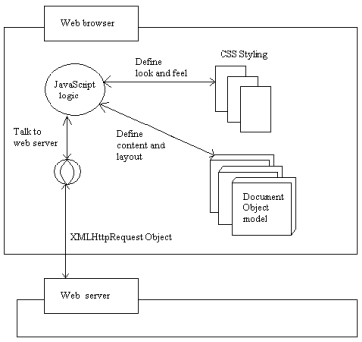

Figure 4.4: The four technologies of Ajax [5]

Figure 4.4 shows a model of the technologies and how they used together. CSS, DOM and JavaScript, have collectively been referred to as DHTML, see section 4.2.1. Ajax uses the benefits of DHTML to create interactive interfaces. However, where DHTML needed a full page reload from the server, Ajax uses the asynchronous request to the server and extends the functionality that is possible to perform on web-pages [5].

4.4.1 Communication with the server

In the traditional Web application model, communication is synchronous and it requires a page reload when data is transferred to/from the server. Page reloads result in that the user's workflow is interrupted. This disadvantage is avoided in Ajax, where the communication with the server is asynchronous [12] [5] and hence, it prevents the browser from being blocked while waiting on a response from the server. The user can continue interacting with the interface in the browser, while the web-application is communicating with the server. The synchronous and asynchronous communication-methods are shown in figure 4.5.

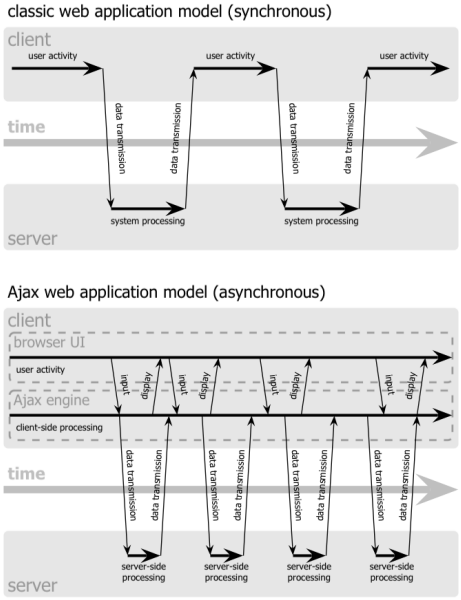

Figure 4.5: "The synchronous interaction pattern of a traditional web application (top) compared to the asynchronous pattern of an Ajax application (bottom)" [12]

In an Ajax application the Ajax engine is loaded into the browser during the first page-load. The engine is activated by the user actions that converted into a JavaScript call [12], instead of generating a HTTP request. The engine also handles user responses that does not require a request to

the server, such as simple navigation, editing data in memory and some data validation [12]. The engine is responsible for the asynchronous communication with the server via the XMLHttpRequest object [5]. The XMLHttpRequest object is explicity designed for fetching data in the background, and is not visible to the user. It has originated from Microsoft-specific ActiveX components, that was available as JavaScript-object in the Internet Explorer browser. Other browsers have implemented native objects with similar functionality [5]. The Ajax engine is also responsible for sending only relevant data to the server and process server response to update only relevant page elements. Figure 4.6 shows the comparison of the classic web application and the Ajax Web application with the Ajax engine.

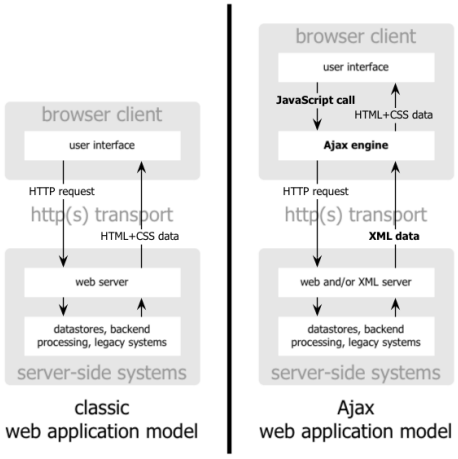

Figure 4.6: Classic web application model compared to the Ajax web application model [12]

Areas where there are advantages of using Ajax Scripts:

- Instant feedback on user input inserted into forms
- Long-running queries
- Deep hierarchical tree navigation
- Rapid user-to-user communication

Section 4.6, will give examples and explain more about the advantages gained from using Ajax.

4.5 Challenges when Automating Tests for Dynamic Web-applications

Automatically testing static HTML pages can be done by using i.e. crawlers; programs that recursively test all links, misspelling and HTML errors. Dynamic Web-applications are becoming more complex, and automated testing

has become a more difficult task. More complex web-applications results new challenges when creating automated tests, such as the following [11]:

- Dynamic web-pages can change frequently while in use, i.e. depending on server state, user input, etc.
- Dynamic web-applications can have a more complex interaction with other components, such as browsers, operating systems, databases, proxy servers, etc.
- Several users can be inexperienced, which adds a need to test unexpected behavoir of a web-application.
- Rapid changing technology, such Ajax.

Web-sites that contain Ajax and JavaScript, se section 4.3, introduces additional challenges because scripts are evaluated after a page has finished loading [39]. This means that functions defined in scripts are not automatically executed when the page loads. They are stored until called by a "users-actions" or input on the page [39]. Ajax in addition introduces new challenges when creating automated tests of web applications since it does not follow the traditional request/response architecture. The XMLHttpRequest protocol allows updates of parts of a page's content. When updating parts of a web-page, there is no page reload/refresh. As a result it is difficult to know when a web page is ready for testing after a user's action is performed.

With dynamic web-sites becoming increasingly complex, thoroughly and frequently testing is needed and requires more time and effort [11]. Automated tests could allow this, if created before the application and be run during development of the application [27]. However, in current automated testing tools such as "capture-replay" tools [11], tests cannot be created before the web application is built. Capture-replay tools are created for capturing a set of user actions while the user manually traverses the web-site. The captured scenarios can then be automatically replayed for testing.

4.6 Typical Scripts

It is important that the concept we include to AutAT, can be used with common dynamic web-pages that contains JavaScript and Ajax. JavaScript is today commonly used and Ajax is an increasingly used technique. The intention of finding typical scripts with JavaScript and Ajax, is to understand how dynamic web-pages can be created and how users can interact with these pages.

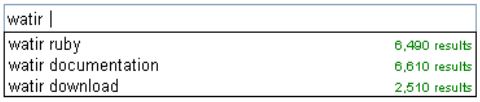

Figure 4.7: Suggestions When Typing

4.6.1 Instantly update the web-page based on written input

Dropdown-list where the user can type a letter to select the first element that starts with this letter, is a dynamic feature often used in web-pages. An example can be seen when selecting a destination at the Norwegian Airlines[1] web-page. Typing the letter 'T' when the list of destinations is selected, brings the user directly to 'Tallin', which is the first city in the list that starts with a 'T'. This functionality can be created with pure JavaScript.

Google Suggest[2] and ObjectGraph Dictionary[3] are examples of pages that support suggestions to the user when he starts typing a word in a textfield. The pages dynamically fill a dropdown-list with words that begin with the inserted letters. An example of this, can be seen in figure 4.7, where the user has inserted the word "watir". The difference with implementing this functionality with Ajax rather than JavaScript, is that the alternatives for the dropdown-list does not need to be downloaded together with the web-page at the start of a session.

Yahoo Instant Search[4] uses Ajax calls to provide the users with answers to a search as they type. Ajax Translator[5], as seen in figure 4.8, translates each word the user types into another language. Common for these examples are that there are too many alternative words to be uploaded together with the web-page. Therefore, the Ajax engine "silently" loads the information needed into the web-page as a result of what the user types letters into a textfield.

4.6.2 Validating form-input values

Another example of instantly updating the content of a web-page based on written input can be seen in form-validation. JavaScript allows checking of input values before sent to the server. If a user has inserted incorrect values, the page can respond with user feedback before anything is sent to the server. JavaScript can be used to check if the inputs are the correct

[1] http://www.norwegian.no
[2] http://www.google.com/webhp?complete=1&hl=en
[3] http://www.objectgraph.com/dictionary/
[4] http://instant.search.yahoo.com/
[5] http://ajax.parish.ath.cx/translator/

Figure 4.8: Translator

type, length, etc. An example, can be found in JavaScript Form Validator[6]. When a user types in numbers in a form where letters should be inserted, a popup window appears with a warning, after the user has clicked the submit-button. JavaScript cannot validate against values stored at the server. With Ajax the users can get immediate feedback about incorrect values checked against values stored in the server. In example, it can be used when selecting a new username, to check if the username is already taken by other users. An example of this can be seen when creating a Gmail[7] account. In Gmail, it is possible for a user to write in a desired login name and check the availability of that login name, before he writes in the rest of his personal information on the page.

4.6.3 Allowing real-time applications

Chat and instant message (IM) implemented with Ajax are browser-based applications, allows visitors on the same web-page to see and interact with each other without needing to use client-side software. I.e., the instant message application Meebo[8], which lets a user log into his IM network, such as msn, from any computer with a browser and a Internet connection. Another example is php Free Chat[9], which uses Ajax to ”silently” refresh and dispaly the chat and nickname zone. In traditional chat and mail web-applications, each message received would be sent to the client as a new page. The user would therefore be forced to wait for a new page reload for every message recieved from other users. In difference, chat and instant message applications implemented with Ajax, allow users to see its peers comments without needing to reload the page. As a results the Ajax applications will appear much like ordinary desktop instant message programs. These web-based

[6]http://www.softcomplex.com/products/tigra_form_validator/login_form_validation.html
[7]mail.google.com/
[8]http://www36.meebo.com/
[9]http://www.phpfreechat.net/demo.en.php

chat applications are examples of adding content to a web-page that is not a result of a user's input, but involve real time data processing via JavaScript.

4.6.4 Lazily loading hierarchical content

The expand/minimize functionality can be seen i.e. at Start[10] web-page and the Backbase[11] web-page, as well as in some demos found in the Backbase web-page. The concept is that content on a web-page is i.e. hidden and becomes visible when a user performs a mouseclick or a mouseover on a particular element on the page. If the content was visible, then the result of the mouseclick will be hiding the content. This concept can be created with pure JavaScript or Ajax. The difference in implementation is that with JavaScript, all the content in the web-page must be loaded into the browser when the page loads. The loading of all content may result in the browser loading unnecessary data into the page, i.e. content which the user does not make visible. Especially with deep tree navigation, the unnecessary data can significantly increase the loading time. This means that the user would have wait a long time before he could start working with the page, which could make him reluctant to use the page again. With Ajax, the desired content can be "silently" loaded into the page after i.e. a mouseclick. This means that content will be loaded if the user needs it, in other word lazily loaded. Ajax transfers only a small amount of data from the server, which gives it a higher performance and saves server resources. The user can in addition continuse working with the page, while he waits for Ajax to update the page based on the request. This makes the end-users experience is continuous and hence be more likely to use the page again.

4.6.5 Updating and sorting lists

Typical list operations are inserting, deleting and updating rows in a list, which can be done by in a web-page without reloading the page when using Ajax. A typical examples can be seen on the SimplyHired[12] and the KAYAK[13] web-page. The KAYAK web-page presents in a list the result of a search for flights. A user can i.e. check and uncheck select-boxes on the page, which will result in that the list is updated, as seen in figure 4.9. Each change on the page inflicted by the user, results in that an updated request is sent by the Ajax engine to the server in parallel to allowing the user to continue working on the page. Lists can also be filtered and sorted i.e. by date or name.

[10] http://www.start.com/3/
[11] http://www.backbase.com/
[12] http://www.simplyhired.com/
[13] http://www.kayak.com/

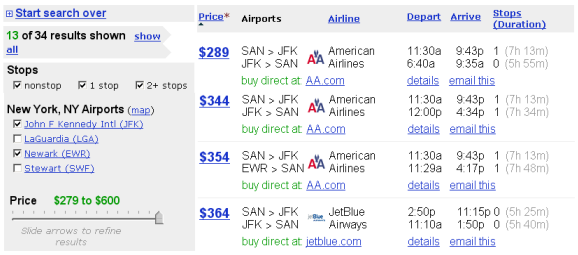

Figure 4.9: Refreshing the List Without Refreshing the Page

4.6.6 Summary of Advantages

Table 4.1 gives a summary of common functionality implemented with by Ajax and JavaScripts. The table shows the functionality, a short description of the functionality and some example web-pages.

4.7 State-of-the-Art of Test Frameworks

The test frameworks presented in the state-of-the-art in the master thesis [30], can be divided into two categories; frameworks that emulates the browser such as Canoo WebTest [1], and frameworks that run the browser such as Selenium [25] etc. Test frameworks that emulates browsers, perform testing by sending requests and verifying the responds [1], and can therefore not test client-side dynamic elements, such as functionality implemented with Ajax- and JavaScripts [14]. Test frameworks that run the browser can simulate a users behavior, and therefore activate Ajax- and JavaScript calls, as well as verify the result in the browser [14]. We will therefore look at test frameworks of this type, and investigate if each can be used for testing Ajax- and JavaScript.

4.7.1 Selenium

Selenium is a framework for testing the functionality of web applications. It is a open-source test framework and implemented fully in JavaScript. Selenium tests run in web-browsers and supports browsers such as Internet Explorer, Mozilla and Firefox [24]. Selenium supports two types of tests; test-runner and driven [18]. The tests are run by a JavaScript engine, called the BrowserBot [15], that is downloaded into the browser to the application that will be tested.

Test-runner scripts are written in static HTML- files that contain tables with commands and assertions which will be tested in the application [18]. The HTML-files are loaded into the browser where the application is deployed and executed by the BrowserBot engine.

Functionality	Description	Examples
Instantly update the web-page based on written input	The user does not need to wait until a form is submitted before content on the web page is changes based on the user input.	Norwegian Airlines, Google Suggest, ObjectGraph Dictionary, Yahoo Instant Search and Ajax Translator
Validating form-input values	Input the user types into forms can before submitted be validated against values stored at the server. It can also be checked that it contains the correct type of input, such as i.e. integer or string.	Log-in at Gmail and JavaScript Form Validator
Allowing real-time applications	Information from the server can be "silently" loaded into the webpage, allowing real-time application such as chat and instant messaging.	Meebo, Free Chat and mailing-list in Gmail
Lazily loading hierarchical content	Content not visible to the user when the page are loaded, can be fetched later when needed by the user.	Start, Backbase, demos in Backbase, i.e. Backbase PC Shop
Updating and sorting lists	Lists can be silently updated, i.e. based on user input into forms. They can also be sorted and filtered without the need for reloading the web-page	KAYAK and SimplyHired

Table 4.1: Summary of Typical Scripts

Driven test-scripts are written in any programming language supported by Selenium (Java, Ruby, and Python) [18], and is therefore more complex than test-runner scripts. However, they are more powerful and flexible, and can be integrated with xUnit frameworks. In difference to test-runner scripts, the driven scripts are executed outside the browser, in a separate process, and drive the browser by communicating with the BrowserBot engine [18].

Selenium can be used to test JavaScript and Ajax [18]. To wait for an element in a Selenium test, the "waitForValue"- and "waitForCondition"-statements

TestCommentary		
open	/message/20050409174524.GA4854@highenergymagic.org	
dblclick	//blockquote	
waitForCondition	var value = selenium.getText("//textarea [@name='comment']"); value == ""	10000
store	javascript{Math.round(1000*Math.random())}	var
type	username	user${var}
type	email	user${var} @mos.org
type	comment	hello there from user${var}
click	//form//button[1]	
waitForCondition	var value = selenium.getText("//div [@class='commentary-comment commentary-inline']"); value.match(/hello there from user${var}/);	10000
verifyText	//div[@class="commentary-comment commentary-inline"]	regexp:hello there from user${var}
clickAndWait	//div/div[position()="1" and @style="font-size: 80%;"]/a [position()="2" and @href="/search"]	

Figure 4.10: Example of a Selenium test with waitForCondition (FIT) [13]

Tests the show stock details page.	
click	link=Acme Oil
pause	500
verifyTextPresent	Details for Acme Oil
click	link=Acme Automotive
pause	500
verifyTextPresent	Symbol:ACA

Figure 4.11: Example of a Selenium test with Pause statement (FIT) [18]

can be used [13]. These statements allows a user to write a condition in JavaScript, and pause the test until the condition evaluates to true. An example of this can be seen in figure 4.10. To make a Selenium-test wait a given amount of time, a ”Pause”-statement can be used [18], as seen in figure 4.11.

4.7.2 Watir

Watir (”Web Application Testing in Ruby”) is an open-source automated testing tool which can be used when performing functional testing, automated acceptance testing or large-scale system testing on web-applications in Internet Explorer [22] [16]. Tests are written in Ruby and the Ruby scripting language is used to simulate possible user activities in a browser window through Watir, which gives access to the Component Object Model (COM) interface. The COM interface provides control of the Internet Explore browser and allows access to the objects in web-pages presented in the browser (through access to the HTML DOM) [22] [16]. When directly accessing the HTML-objects through the DOM, it does not matter where the element is located on the screen or whether the element is currently obscured by another window. In addition, using the DOM allows Watir-based tests to run in minimized browsers.

```
require 'watir'
require 'test/unit'

 class TC_test < Test::Unit::TestCase

        def test_sadf
                failedSteps =0
                passedSteps =0

                ie = Watir::IE.start("http://instant.search.yahoo.com")
                ie.text_field(:name, 'p').set('ruby')
                while ! ie.link(:id, 'yschakis').exists? do sleep 0.1; end
                begin
                        assert(ie.contains_text("Ruby Home Page") )
                rescue
                        puts("Step failed: Check text ='Ruby Home Page'")
                        failedSteps += 1
                end
        end
end
```

Figure 4.12: "Wait for element" in Watir

```
require 'watir'
require 'test/unit'

class TC_test < Test::Unit::TestCase

        def test_sadf
                failedSteps =0
                passedSteps =0
                ie = Watir::IE.start("http://www.google.com/webhp?complete=1&hl=en")

                ie.text_field(:name, 'q').set('ruby')

                sleep 2

                begin
                        assert(ie.contains_text("ruby bridges") )
                rescue
                       puts("Step failed: Check text ='ruby bridges'")
                       failedSteps += 1
                end

                ie.text_field( :name, 'q').set('ruby on rails')

                ie.button(:name, "btnG").click # "btnG" is the name of the button
        end
end
```

Figure 4.13: Sleep-command in Watir

Watir can be used for testing JavaScripts [16]. In Watir, there is no "waitForCondition"- or "waitForValue"-statement as it is in Selenium. However, a similar functionality can be created by checking that an element exists and wait (sleep) a given time if it does not, and then check for the element again. This is repeated until the element exists, as seen in the example in figure 4.12. The example is also implemented with a timer, such that the test will continue if the element never appears. The example in figure 4.13, shows how a sleep-command is inserted to pause the test-execution for two seconds. Watir can therefore be used for testing Ajax script.

4.7.3 JsUnit and jWebUnit

Both the JsUnit [31] and the jWebUnit [32] are a part of the xUnit family [33], which is a unit test framwork. The xUnit family have unit test frameworks for many languages, and all are open-source software. The basic members of the xUnit family have many extensions, which targets specialized domains. The JsUnit is targeted for JavaScript testing, while the jWebUnit is targeted for testing web-applications.

```
try {
      doRequest();
      waitForRequestToFinish();
      assertRequestWasSuccessful
  } catch (e) { // do something }
```

Figure 4.14: JsUnit Example that is claimed not work for testing Ajax

JsUnit is an extension to the xUnit testing framework, which is created for testing client-side (in-browser) JavaScript [31]. A Test Page in JsUnit is any HTML-page that includes the jsunit/app/jsUnitCore.js JavaScript file; this file is the JsUnit engine [31]. Each Test Page have Test Functions that are distinguished from other functions on the page by its name, which begins with "test". The HTML-page is passed as an argument to a TestRunner HTML page.

"**jWebUnit** is a Java framework that facilitates creation of acceptance tests for web-applications" [32]. jWebUnit is a result of a refactoring of HttpUnit and JUnit when creating acceptance tests. It includes a set of assertions to verify the application's correctness, which are navigation via links, form entries and submission, and other typical business web application features. Instead of using only JUnit and HttpUnit, the jWebUnit allows more rapid test creation with its simple navigation methods [32].

Since JsUnit is created for testing client-side JavaScript. It can therefore be assumed that it can be used for testing Ajax-scripts as well, because Ajax scripts are activated by JavaScript calls. However, it is claimed that JsUnit cannot test Ajax scripts because Ajax scripts can run in multiple threads [8], in difference to JavaScript that only run in one thread. The asynchronous XmlHttpRequest will therefore never finish until you completely finish from the current thread, and it is therefore believed that the example in figure 4.14, does not work. jWebUnit has support for testing JavaScript, but it is claimed that jWebUnit's api is too incomplete to confirm that it can test Ajax [21]. We will not investigate these issues further, but conclude there are other more appropriate frameworks we can use, which are not claimed to have such limitations.

4.7.4 FIT and FitNesse

FIT (Framework for Integrated Test) is a general framework [7] for acceptance testing. Tests are written in tables, in i.e. Microsoft Word, and the tables are read in HTML-files. The first row in the table contains filenames ending with "eg", which specifies a Fixture to use, and the rest of the table-rows are actions performed with this Fixture. A Fixture can be a special class or module that transforms input from the table to a form suitable for running the program. The result from the program is transformed to a form

that can be verified against the FIT test. A FIT runner can be used to execute a Fixture, but the applications must then be written in the same language as the FIT runner [7].

FitNesse is an extension of the original FIT framework and depends on two subsystems: the FIT framework and a Wiki Clone [28]. FitNesse is a framework for automating software tests. It provides an environment where programmers and customers can define and execute acceptance tests inside a web browser. The FIT framework have been described above, is integrated with the wiki clone. Wiki is a web-based collaborative editor that lets users edit the web pages it manages [28]. Wiki controls the layout of the pages and the users only needs to fill in the content. The tests in FitNesse are written as HTML-tables, the same as in the FIT framework. However, writing the actual tests are simpler than in FIT because of the wiki framework [28]. Tests are run in the same way as in FIT, where a runner can only be executed on applications written in the same language. In FitNesse, a TestSuite page is at the top of a hierarchy consisting of Test Pages and possible new TestSuites. To run tests, a user invokes a Test Page that will run all tests on the page. When running a TestSuite page, all tests on that page, and the tests on its sub-pages, will also be run.

HtmlFixture is a Fixture created to test web-pages [4], which can activate JavaScripts, submit forms, ”click” links, etc, in a test. We have not been able to find out if HtmlFixture can be used to test Ajax-scripts.

4.7.5 Summary of Existing Test Frameworks

The selection of existing test frameworks presented is some of today's most commonly used for testing web applications. They navigate their way through applications by clicking on links and inserting values into forms. Table 4.2 summarizes their advantages and disadvantages. Common for all the test frameworks, is that they can not be used for testing ActiveX plug-in components, Java Applets, Macromedia Flash, or other plug-in applications [10].

After researching several testing frameworks, we have found that most testing frameworks are created for synchronous communication and need to be modified to work with Ajax. The problem with asynchronous communication is that it is hard to know when the web application is finished requesting the server, and hence, all elements are present in the browser. Research shows that there are two solutions that solve the problem which arises when Ajax is used in web-pages. These solutions are possible because testing frameworks that runs a browser and simulates a users behavior, do no care about how content is inserted into a page, as long as it is present when it is tested.

The first solution was to insert a sleep command can into the test-script.

With this solution it is important that the time set for the test to sleep is (most likely) long enough for the page update to be performed. Disadvantages with this solution are that if the sleep command to short, then the test will fail even if the web-page is correct. In addition, the test will probably sleep an unnecessary amount of time; from when the element appears until the test is finished sleeping. An advantage, however, is that the solution is simple and easy to implement.

The second solution to enable test frameworks to test Ajax-scripts, is to make the test wait until the needed element is present. An advantage with this solution is that the test-script does not wait an unnecessary amount of time. A disadvantage, however, is that writing the test-script is more complex. A timer should also be added to prevent the test-script from waiting forever if the element never appears on the web-page.

These solutions were commonly used in Watir- and Selenium scripts [13] [18]. How to enable Fit/FitNess, JWebUnit and JsUnit to test Ajax-scripts are not as thoroughly discussed on the Internet. To test the concepts of AutAT, we only need one testing framework to which we can convert AutAT tests. We therefore not conclude that these frameworks can or cannot test Ajax-scripts, but postpone further investigation of these frameworks until it is needed in further development. We have selected to use Watir as the testing framework to which we will convert AutAT-tests, because we knew several persons that had used Watir to test Ajax- and JavaScript.

Type	Advantage	Disadvantage
Selenium	Can test application behavior in different browser such as Internet Explorer, Mozilla or Firefox. It is versatile because tests can be written both in HTML tables and in a programming language.	The driven tests can be a big challenge with some languages, such as Java, because of poor documentation. It is hard to work with Selenium in a distributed setting.
FIT and FitNesse	User can describe requirements in tables with text and no programming is necessary. Easy for the developer to map the requirements to the business logic and the execution of the tests can be automated.	There is no capture and replay possible in the framework. It requires the users to know commands and Fixture names.
JsUnit and jWebUnit	Both are simple to use, because the validations of presence is done by using a simple commands. It is possible to validate input elements in HTML forms in jWebUnit. JsUnit has the ability of managing multiple testing boxes and distributing tests among machines to run against multiple browsers.	Appropriate for programmers, not so much for non-programmers.
Watir	The tests are run from a browser, and Watir-tests the same way as people work with browsers. It is possible to use Ruby to interact with the system directly, for example accessing a database. With Ruby, it is possible to create unit test that can be used with Watir.	It only works with Internet Explorer, and there is no possibility to check the actions in the application in other browsers. It uses Ruby, which developers have to learn.

Table 4.2: Summary of Frameworks

Chapter 5

Own Contribution

This chapter first presents the requirement specification for the new version of AutAT. It then presents the ideas and concepts we were considering when selecting a solution for how to enable a user to model dynamic web-applications. This chapter also includes a detailed description of the chosen solution.

5.1 Functional Requirements Specification

The following requirements specification presents the functional requirements for the new version of AutAT. This version is an extension of the original AutAT found in the master thesis [30], and has therefore some of same requirements.

FR.1 Register Project The user must be able to create a new project which can contain several AutAT-tests.

FR.2 Create Test The user must be able to create AutAT-tests.

FR.3 Start Point The user must be able to define the URL to where a test must start.

FR.4 Edit Test The user must be able to edit a test at a later session.

FR.5 Test Web-elements The user must be able to test that text, links, form-elements, contexts and titles, are in a web-page.

FR.6 Define Actions The user must be able to define common actions performed on a dynamic web-page.

FR.7 Aspect The user must be able to create an aspect that can apply to many tests.

FR.8 Model Dynamic Changes The user must be able to model that a web-page dynamically changes as a result of user-input.

FR.9 Create Test-Script The system must be able to create test-scripts that can test Ajax- and JavaScript, and run on a state-of-the-art test framework.

5.2 Concepts Created to Fulfil The Requirements

It is important to find a new concept for AutAT that enables users to model possible functionality in dynamic web-applications, containing Ajax- and JavaScript. The main concept in Ajax can be seen as an ”event” that leads to changes in the browser. An ”event” can i.e. be that a user; inserts text into a textfield, clicks on elements, checks or unchecks a checkbox, etc. With Ajax, an event in a dynamic web-page can also be a result of changes in the state of the server, as seen in the chat applications in section 4.6. It was therefore necessary to find out how to model the triggering of an ”event”, and that parts of the browser changed as a result of the ”event”. We came up with several alternatives that will be presented here. These were continually evaluated to find the most suitable solution.

5.2.1 Alt. 1:The Input-Output Element With Several Outputs on a Separate Page

The triggering of an ”event” and the resulting changes in a web-page, can be seen as respectively, ”input” and ”output”. To model this input and output, we created a dynamic element called the ”Input-Output” element, which can be added to a ”static”-page (the traditional element representing a web-page). The element contains an input- and output-area, as seen in figure 5.1, where a user can insert page-elements, such as textfields and/or links. The elements in the input-area represents the source of the event, to which input-values can be specified. The elements in the output-area is the default output elements (it can be empty), which will be visible when the actual web-page is loaded into the browser, and when unspecified input-values are inserted.

The values inserted to the elements in the input-area and the resulting change in the browser, are specified in a dynamic-workarea (a separate page from where the main AutAT test is modelled). The dynamic-workarea is connected to the Input-Output element with a ”+”-box on the arrow between the two areas, seen in figure 5.1. In this workarea, it can be specified several output-areas for different input-values. It will by default contain the input-

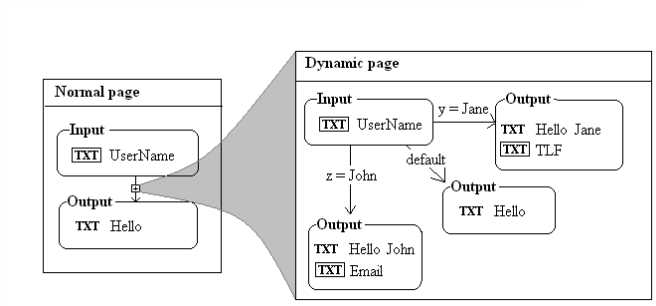

Figure 5.1: Alt. 1: The Input-Output Element with several output-boxes

and output-area from the Input-Output element. In figure 5.1, the input-area contains a textfield called ”UserName” and with a default output-area containing the text ”Hello”. ”John” and ”Jane” represents two input-values resulting in two different output-areas. The advantage of having a separate ”dynamic”-workarea, is that the static-page is easy-to-follow when there are several different output-areas, because it will only contain the Input-Output element. A problem with this concept is when an output-area in the dynamic-workarea includes elements which should be linked to a new static-page in the normal-workarea, which is not possible.

5.2.2 Alt. 2: The Input-Output Element With a Table on a Separate Page

We created another concept based on the Input-Output element, but which included a different dynamic-workspace. In this dynamic-workspace, a table is used to represent different input- and output-values, as well as the type of these values, see figure 5.2. The advantage with this solution is that it gives a good overview of the different input- and output-value combinations. The problem with this solution is that it is only possible to have one input-value. In addition, the input-value can only result in one output-value. For example, it is not possible to model that inserting text into a textfield and clicking on a button, can result in an output. It can neither be modelled that clicking on a button results in that a text appears in addition to a link. This concept is therefore only sufficient when the input and outputs are of a simple type such as text.

5.2.3 Alt. 3: The Dynamic Element

The Dynamic element uses the same concept of input- and output-areas, but allows several output-areas to be included in the traditional page in AutAT. The idea is taken from the form-element in AutAT, seen at the right hand side of figure 5.3. The Dynamic element by default, consists of two subareas; IN and STD OUT, which means input and standard output respectively.

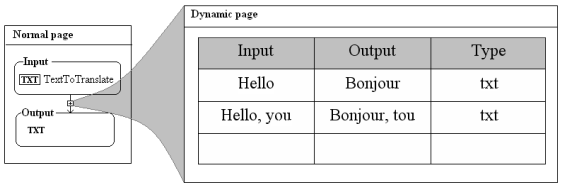

Figure 5.2: Alt. 2: The Input-Output Element with a table

The user can in addition add several OUT-areas representing different outputs as a result of different input-values.

The left hand side in figure 5.3 shows an example of the Dynamic-element with one added OUT-area. In the ”IN”-area, the user can insert links, form-elements and text. If the user adds a text into the IN-area, he must create an event to that text, such as ”onMouseOver” or ”onClick”. These events can also be used on form-elements and links, but this is not required since form-elements and links have standard events attached to them, i.e. a button is pressed. The IN-area only contains the elements in which input-values will be inserted, so a problem with this concept is that it is difficult to find a clear way of modelling the input-values. Another problem is that the Dynamic element will become large and complex if there are added several OUT-areas.

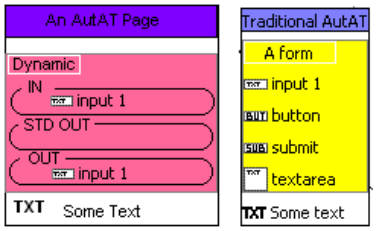

Figure 5.3: Left: Alt. 3: The Dynamic Element, Right: The Form Element in AutAT

5.2.4 Alt. 4: The State Concept

In the previous concepts, a page in the AutAT workarea (AutAT-page) represents a web-page that is loaded into a browser. This result in that dynamic changes, which do not concern reloading a web-page, must be modelled into the same AutAT-page. An AutAT-page can instead be seen as a state, which can be defined as a snapshot of the content on a web-page at a certain time. A state can change as a result of events, and they are independent on whether a page is reloaded or only parts of the web page is dynamically changed. This is an advantage for users that do not know the difference

in functionality implemented with Ajax, JavaScript and static HTML, since they do not need to decide when to model a new page.

Example: Backbase PC Shop
Figure 5.4 shows an example of how to model ”buying a computer” from the ”Backbase PC Shop[1]” web-page. This example is not created to test the whole page, which would require additional tests, but to present the State concept. Figure 5.4 has a starting point, which is adopted from the initial AutAT tool, that leads to the first state. Aspects are also adopted, which is used for writing tests that are performed on several AutAT pages. The starting point, ”Start Shopping”, leads to the first state; ”Select PC”. In this state the web page consists of four DIV-boxes; ”inShop”, ”price”, ”inBag”, where a DIV-box models that content is placed together somewhere on the web-page. When DIV-boxes with equal ids are placed both in a state and in an aspect, it means that the state contains the elements in the union of the two DIV-boxes. The ”Select PC” state, therefore, contains the elements from the union of the DIV-boxes in the ”Select PC” state and the two aspects; ”An aspect” and ”Price aspect”.

In difference to most web-shops, the shopping bag named ”inBag”, in the example, already contains a PC when the user starts shopping. This PC is represented by the ”GX60” DIV-box. This Div-box is moved from ”inShop” DIV-box in the ”Select PC” state, into the ”inBag” DIV-box in the ”new PC selected” state, and the states are connected with an ”event”-transition. An event-transition is used to specify that an action is performed on a particular element in the web-page. In this example, the element and the event-type is specified as respectively ”add” and ”click”, as shown in the emphasize box in figure 5.4. Therefore, clicking on the ”add” DIV-box that contains the text ”Add to bag”, will result in a transition from the ”Select PC” state to the ”new PC selected” state. The event-transition can also contain an input if needed, such as when a value is inserted into a textfield.

5.2.5 Alt. 5: The ”Page-on-Page” Concept

The ”page-on-page” concept models a web-page as a ”dynamic”-page, which can contain ”states” representing areas that can dynamically change, as seen in figure 5.5. This means that if parts of a web-page is changed, this is modelled as a new state inside a ”dynamic”-page. However, if a new web-page is loaded into the browser, it is modelled as a new AutAT-page. An advantage with this concept is that it can model that two browser-windows are open simultaneously. This is useful when i.e. modelling pop-up windows, which can appear in web-applications. A disadvantage is that a user has to know when to, and when not to, model a new page. This means that he has to

[1] http://www.backbase.com/

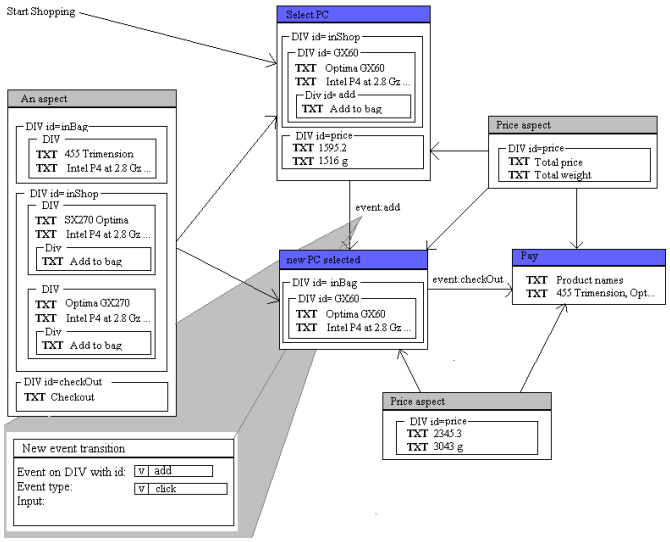

Figure 5.4: Example of the state-concept by modelling the Backbace PC Shop web-site

have knowledge about Ajax and JavaScript, which is not common for non-programmers.

Example: Backbase PC Shop
Figure 5.5 shows how to model ”buying a computer” on the ”Backbase PC Shop[2]” web-page with the ”page on page” concept. The event-transition and the DIV-box are the same as in section 5.2.4. In the example, the dynamic page named ”A dynamic page” (the large box in the figure) consists of two states; ”Select PC” and ”new PC selected”, and three DIV-boxes: ”inBag”, ”inShop” and ”checkOut”. The ”inShop” DIV-box is modelled in both the dynamic page and the ”Select PC” state. The latter contains the ”GX60” DIV-box, which represents the PC that is being bought. When the event ”add” occurs, the ”GX60” DIV-box is moved from the ”inShop” DIV-box in the top state-, to the ”inBag” DIV-box in the bottom state of ”A dynamic page”. When the ”GX60” DIV-box is in the ”inBag” DIV-box, as it is in the bottom state, a ”checkOut”-event will result in a new dynamic page.

It is possible to connect an event-transition from a dynamic page to another dynamic page or a state inside a dynamic page, as seen in figure 5.6 as respectively the links ’ab’ and ’df’. In this figure, the boxes ’A’ and ’B’ is dynamic pages, and the boxes ’C’, ’D’, ’E’ and ’F’ are states inside these pages. If an event-transition is connected from a dynamic page, such as transition ’ab’, it means that the event can be activated independent of which state the page is in. This is i.e. the case for menus that are visible for all web-pages

[2]http://www.backbase.com/

in a web-site. If the event is connected to a dynamic page, and not to a particular state in that page, then the states without any events connected to them (if any) should be visible in the browser when the page is loaded. For example in figure 5.6, if event-transition 'ab' is triggered, the content in box 'B' and 'E' will be visible in the browser. If the event-transition is connected to a particular state, then it is only the content not in any states and the content in that particular state, which is present. In example, triggering event-transition 'df' from state 'D', will result in that the content of box 'B' and 'F' is visible in the browser.

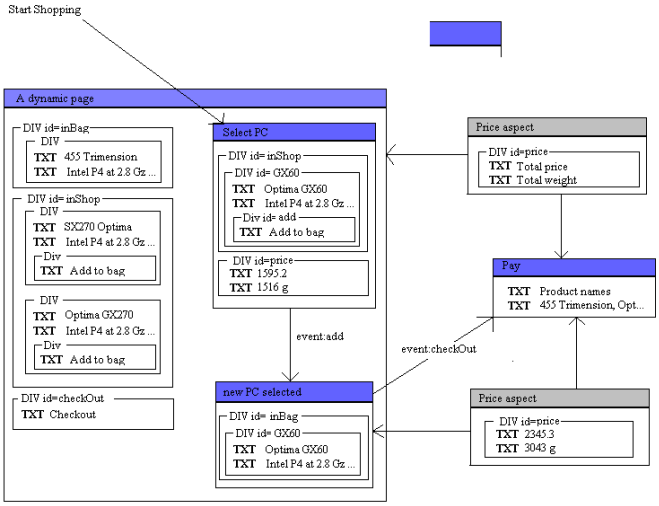

Figure 5.5: Page-On-Page Example

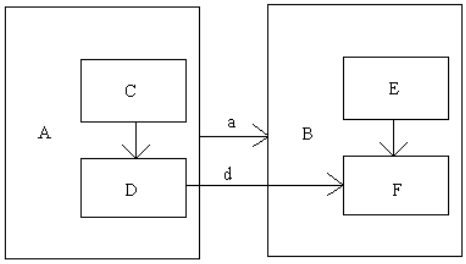

Figure 5.6: Transitions possible in "page-on-page"-concept

5.2.6 Evaluating the Concepts

Table 5.1 presents a summary of the concepts with their advantages and problems. Alternative two and three are both complex, and have problems with modelling several outputs from one input. These concepts can therefore

not be used to model tests in AutAT. Alternative 1; the ”Input-Output” element with several outputs on a separate ”dynamic”-page, do not have these problems. However, in this concept it is difficult to model connections from a ”dynamic”-page to a ”static” AutAT-page. This must be possible if users want to model i.e. that clicking on a button in the output-area, results in a new ”static”-page. This concept can therefore not be used.

The problems with the previous concepts, are not present in the ”state”- and ”page-on-page”-concept. The page-on-page concept is difficult to use without understanding Ajax- and JavaScript, which is not a problem for the state-concept. The ”state”-concept is in addition, easy-to-follow because several dynamic elements are not modelled into one page. States can be difficult to understand for non-technical users, but we believe this will not be a significant problem. For these reasons, we decided to use the state-concept to extend AutAT to test dynamic web-pages.

Alt.	Concept	Advantages	Problems
1	The Input-Output Element with several outputs on a separate dynamic page	Normal ”static” view is kept short when several outputs are modelled.	Difficult to link from i.e. a form in an output in the ”dynamic” view, to the next page in the ”static” view.
2	The Input-Output Element with a table on a separate dynamic page	Good overview of different input-output value combinations	Only one input type and -value possible. Only one output type and -value for each input.
3	The Dynamic Element with several outputs on the same page	Can model that elements in an output-area can lead to a new page.	Several outputs result in a large AutAT-page that is difficult to follow. Unclear how to represent the input values.
4	State-concept	Dynamic behavior can be modelled in a way that is easy to follow. Users do not decide when a new page is necessary.	Can be difficult to understand states for non-technical users.
5	Page-on-page concept	Can model that there are two browsers-windows open at the same time.	Users have to know when to model a new state instead of a new page.

Table 5.1: Summary of concepts

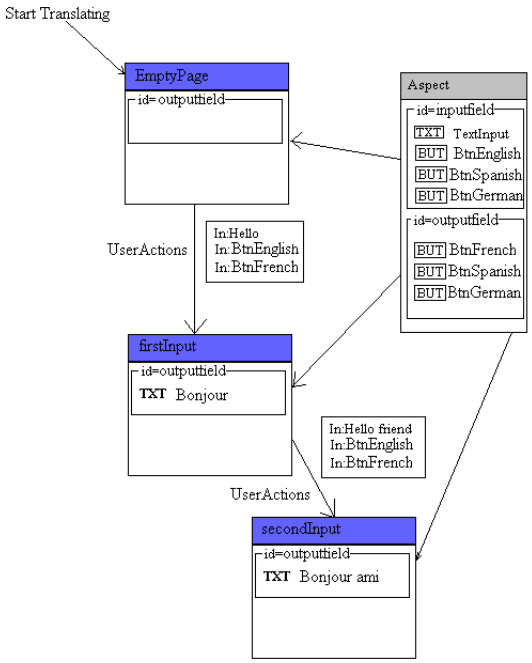

Figure 5.7: Example of the State-concept modelling the AjaxTrans web-site

5.3 The Solution

The solution is based on the ”state”-concept, presented in section 5.2.4, but contains some improvements. These improvements include;

- Changing the element name; ”DIV-box” to ”Context”.
- Changing the name of the transition from ”events” to ”UserActions”.
- Enable that several actions can be performed in one UserActions-Transition, such as inserting different text and then clicking on a button in one UserActions-transition.

5.3.1 Example: AjaxTrans

We will present the solution through an example of a test for the Ajax translator, AjaxTrans[3], see figure 5.7. An aspect containing two contexts is connected to the first, second and third state. The two contexts are “inputfield” and ”outputfield”, wich have the same properties as the DIV-box previously described. The outputfield-context is also in the ”emptyInput”-state, and contains the elements from the union of the context with the same id.

[3]http://ajax.parish.ath.cx/translator/

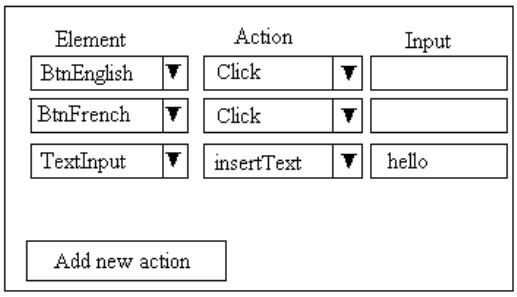

Figure 5.8: ”UserActions”-Transtion

The connection between the first and the second state is a UserActions-transition, which contains a set of user actions that will be performed on the page. When a UserActions-transition is created, a popup-window will appear as seen in figure 5.8, where the user can specify (from top to bottom) the actions that are performed on elements in the state where the transition starts. In this example, the user can select between the elements in the “emptyPage”-state, when creating the transition between the first and second state. This is done in the ”Element”-column in the popup-window. Similarly, an action on the chosen element can be specified in the ”Action”-column, and input can be inserted if needed.

The results from performing the first UserActions-transition in the web-page, is modelled in the Outputfield-context in the ”firstInput” state, where it is written; “bonjour”. To add a second word into the translator, a user need to create a third state; the ”secondInput”-state in the example. Then, a new UserActions-transition must be created, which will be similar to figure 5.8. However, it must be added ”friend” to the string “hello”, which will result in the “secondInput”-state in the “Outputfield”-context, where the translation is: “bonjour ami”.

5.3.2 How The Solution Support Typical Scripts

For AutAT to be taken into use, it is important that users can model common dynamic web-pages. Hence, we will discuss how the State-concept can be used to test the common scripts presented in section 4.6.

The example in the previous section, shows how instantly updating a web-page based on written input from the user, is modelled with states. The test in the example, models that a sentence inserted by a user is translated, which is in the browser done without reloading the web-page. Validating form-input values, can be modelled similarly, because input inserted into a form, can result in that warnings are written in the same web-page. Tests in AutAT will look the same, independent of if the validation is performed

at the server or in the client browser, as when selecting a ”username” in Gmail and JavaScript FormValidator, see section 4.6. This shows that the the users of AutAT only have to concern about the functionality in the web-page, and not if it should be implemented with Ajax, JavaScript or static HTML. Server validation can also be implemented with static HTML, but this would require sending the same web-page back to the client with an additional warning. For these reasons, the implementation decisions can be done by developers. For the same reasons, AutAT can not be used to model that a page lazily loads hierarchical content. It is an implementation decision if content is loaded lazily, or if hidden content is loaded at when the web-page loads. However, the user can model that he content to be hidden when a page loads into the browser, such as sub-menus that are hidden, and later appears, on a web-page.

Content on a web-page can be changed based on user actions other than inserting text seen in the previous example. Studying typical scripts showed that common actions were clicking on a web-element, holding the mouse over a web-element, and pressing a key. These action can be modelled with the State-concept in the UserActions-transition. Another action used in the typical scripts were drag-and-drop, however we chose to not support drag-and-drop in action in this version of AutAT due to time-constraints. Other actions not supported, are actions not possible to perform on web-elements in web-pages, such as inserting text in a button.

Ajax allows web-applications where the client browser is dynamically updated because of changes in the server state, seen i.e. in browser-based chat applications such as Free Chat, and the mailing-list in Gmail. When the server-state is changed, i.e. when a mail is received at a mail-server, there is no particular page-element in the client application that is related to this event. We suggest that the changes in the clients' application can be modelled in a ”ServerEvent”-transition between two states, as seen in figure 5.9 as an arrow with an attached text-box. A description of the event that occurred on the server, can be written in the text-box. The figure shows an example of how a web-based chat application can be tested.

An important question before adding support for server-state events, is if modelling this feature in AutAT requires that the user has knowledge about servers, or the communication between the servers and the client-application. If this is the case, it can not be used by most customers. In addition, how to create Watir tests for this feature, depends on how it is implemented. It can be implemented with i.e. polling needed data from the client-application, or sending data from the server when its state is changed. This makes it complex to create Watir script to test this feature in AutAT. To reduce the scope of this thesis, we will not explore this feature any further.

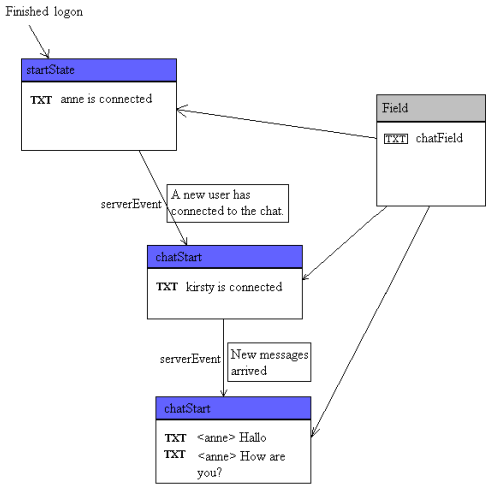

Figure 5.9: Example of the State-concept modelling Free Chat

Chapter 6

Design and Implementation

This chapter describes how we have changed the design of the initial version of AutAT, which was created by Skytteren and Øvstetun [30]. It also describes how these changes have been implemented. In addition, it presents the new version of AutAT by showing how users can create tests.

6.1 Design

We will, as stated in 2.1, create an extension of AutAT. The design of AutAT is therefore partly given by the initial design. We therefore only present the changes to the design that were needed when developing new functionality. To read more about the initial design, see section 4.1 or the master thesis in [30]. The new version of AutAT fulfill the requirements FR.1 to FR.9, specified in section 5.1.

6.1.1 Domain Model

When developing a software system, a *domain model* is used to describe and model the problem domain. The problem domain for AutAT is to model tests for dynamic web-application projects. The domain model is shown in figure 6.1, where a project represents the web-application project that is being developed for a customer. This development project has a set of requirements, usually written as user stories in Test Driven Development. For each user story, one or several tests are created. Each test has a starting point representing the web-page where the test starts. Each test consists of a set of states, each representing a ”snapshot” of the web-application. A test can also contain aspects, which are similar to states. However, an aspect is connected to several states and contains the page-elements common in those states. A state can be connected to ”UserActionTransitions”, containing a set of ”UserActions”. A ”UserAction” represents an action performed on an element on the web-page in that state. Executing the ”UserActions in a ”UserActionsTransition”, leads to a new state. Each state and aspect contains web-page elements, such as text, links, form elements, etc.

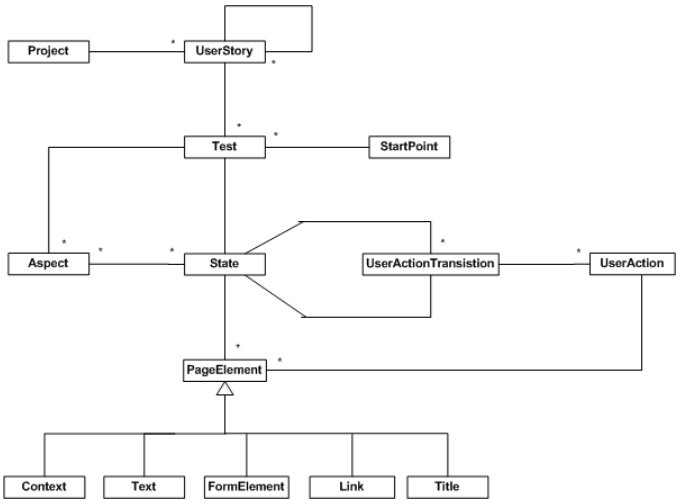

Figure 6.1: The new Domain Model

From a non-technical user's perspective, a form is just a set of form-elements. It is only in the HTML-code where a "form" that contains form-elements, exists. We have therefore removed the "form"-element from the domain model of the initial version of AutAT, see section 4.1.

6.2 Implementation

We have kept the package structure and the architecture of AutAT as described in 4.1, and in more detail in the master thesis written by Skytteren and Øvstetun [30]. However, we have changed some of the classes and methods. This section presents how we have implemented these changes, concerning the packages; *AutAT Model*, *AutAT Exporter*, *AutAT Persistence* and *AutAT UI*.

6.2.1 AutAT Model

The Model package contains the "data-value" objects. The changes we have made to the domain model reflects the changes made in this package, such as modelling states instead of pages, removing the form-element, and changing the transition between states. Figure 6.2 shows a new class diagram.

State
We have not changed the name on the "Page" class, but we have changed the name in the graphical user interface of AutAT. The class is used by several other classes, and therefore, changing the name would result in unnecessary

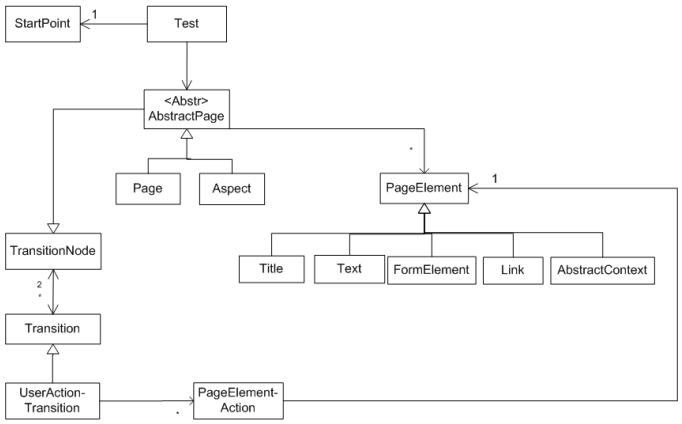

Figure 6.2: The Main Classes in the New Version of AutAT

problems. The content of the class is kept the same.

Form
We have removed the ”Form” object from this package. This for reasons discussed previously in this chapter. To enable form elements to be put on states and aspects, we changed their policy by adding ”PageElementComponentEditPolicy” to the ”FormElementEditPart” class.

Transition
We have removed the ’Connection” class and the ”FormInputConnection” class, and replaced them with the ”UserActionsTransition” class. In addition, we implemented changes needed when showing and handling the ”UserActionTransition” in the graphical user interface.

6.2.2 AutAT Persistence

The AutAT Persistence package is, as stated in section 4.1, responsible for reading tests and startpoints to and from XML files. The changes presented in the previous section, resulted in the need for changing this package. The main changes of the code in the AutAT Persistence package, has been to enable saving the new ”UserActionTransition” to file. In addition, it must be possible to save a form-input element without a form element related to it.

We have not changed the design of the Persistence package, but only its methods. The main classes in the package are TestConverter, PageConverter, AspectConverter TestElementConverter and TransitionConverter which are responsible for converting tests, pages, test-elements and transitions to and from XML, respectively. The changes we made in the graphical user-interface of AutAT, resulted mainly in changes in PageConverter, TestElementConverter and TransitionConverter.

The changes we have made in the AutAT Persistence class can be seen in the changes to the XML schema of which attributes to save from a test. Necessary changes in the code where made to save tests according to the new schema. The XML schema for elements and transitions is respectively shown in the listings 6.1 and 6.2. The complete schema can be found in appendix F.

Listing 6.1: XML-Schema for the Watir exporter

```
% \caption{\label{fig:schema_elements} The schema for
    saving a page element in
% AutAT}
<!--type for link element:linkType-->
<xs:complexType name='linkType'>
<xs:simpleContent>
<xs:extension base='xs:string'>
<xs:attribute name='not' type='xs:boolean'/>
</xs:extension>
</xs:simpleContent>
</xs:complexType>

<!--type for text element:textType-->
<xs:complexType name='textType'>
<xs:simpleContent>
<xs:extension base='xs:string'>
<xs:attribute name='not' type='xs:boolean'/>
</xs:extension>
</xs:simpleContent>
</xs:complexType>

<!--type for form types:formFieldType-->
<xs:complexType name='formFieldType'>
<xs:attribute name='id' type='xs:string'/>
<xs:attribute name='name' type='xs:string'/>
<xs:attribute name='type' type='xs:string'/>
</xs:complexType>


<!--type for a list of elements:elementsType-->
<xs:complexType name='elementsType'>
<xs:choice minOccurs='0' maxOccurs='unbounded'>
<xs:element name='link' type='linkType'/>
<xs:element name='text' type='textType'/>
<xs:element name='formElement' type='formElementType'/>
</xs:choice>
</xs:complexType>
```

Listing 6.2: XML-Schema for the Watir exporter

```
% \caption{\label{fig:schema_userActions} The schema for
    saving an
%         ''userActionTransition'' in AutAT}
<!--type for user actions values-->
```

```
4  <xs:complexType name='userActionValue'>
5  <xs:attribute name='id' type='xs:string' use='required'/>
6  <xs:attribute name='action' type='xs:string' use='
       required'/>
7  <xs:attribute name='input' type='xs:string'/>
8  </xs:complexType>
9
10 <!--type for userActions transition:
       UserActionsTransitionType-->
11 <xs:complexType name='userActionsTransitionType'>
12 <xs:choice eminOccurs='0' maxOccurs='unbounded'>
13 <xs:element name='userAction' type='userActionValue'/>
14 </xs:choice>
15 <xs:attribute name='from' type='xs:string' use='required'
       />
16 <xs:attribute name='to' type='xs:string' use='required'/>
17 </xs:complexType>
```

6.2.3 AutAT Exporter

Implementing AutAT to automatically build test-scripts to include Ajax-based web-applications, introduce new challenges. The test-scripts created by AutAT must be able to run independent of whether the web-application is implemented with i.e. JavaScript, Ajax or with static-HTML. We therefore discuss the problems and possible solutions for how to implement the Watir exporter in AutAT, and we present the in addition to the resulting implementation.

There are two main solutions for how to write Watir-scripts for Ajax-based web-applications, as presented in section 4.7; inserting a sleep-command, or checking in a while-loop, if an element exists until it actually does exist.

Problems and Possible Solutions

There are two main solutions for how Watir-scripts for Ajax-based web-applications can be written, see section 4.7.2; with ”sleep”- or ”waitForElement” commands. There are advantages and disadvantages with implementing each solution in AutAT.

The easiest solution to implement is the ”sleep” solution, which only comprises of inserting a sleep command after all ”userActions”-transitions. A disadvantage, however, is that the test will sleep even if all elements are present, which will result in unnecessary slow test execution.

The ”waitForElement”-solution is a more complex solution to implement in AutAT. The problem is that AutAT must have information about which elements to wait for after a user-action is performed if all elements are not

present. Only elements requested with Ajax calls can appear after the web-page has loaded, so waiting for other elements will suspend the test. But how the elements in a web-page are requested, is most likely not known by the person who uses AutAT to create the tests. A solution is to add a possibility of defining, in a UserActions-transition, which elements that are requested with Ajax. Such information can then be inserted by developers after a user have created tests. This way, AutAT knows if an element should be present immediately, or if it appears when the Ajax-call returns. The test script could then contain "waitForElement" instruction on the elements requested with Ajax, which suspend the test execution until the element appears.

A better solution is to implement "waitForElement"-statement with a maximum time limit, on all elements that are in an AutAT state, no matter if it is implemented with Ajax or not. This way, if an element is implemented with Ajax in the web-application, the test script execution will wait until it appears. Waiting for an element that is implemented with traditional request-reply architecture, will always reach the maximum limit of time, because such elements do not appear on the page after it is loaded. However, the test will not suspend forever, and the total extra time it takes to wait for missing elements is maximum the number of elements in the tests multiplied with the maximum waiting time. A maximum limit of time to wait, is also a good idea for elements implemented with Ajax in case i.e. the communication with the server is lost and that the element will not appear even if it is supposed to.

Implementation of the Watir Exporter

The general exporter package was implemented in the first version of AutAT, as described in section 15.3 [30]. This includes the *DirectoryWalker class*, the *TestStep class*, the *StepListBuilder*, etc., which handles the traversing of elements, modelled by the user, in the AutAT workarea. The changes in the implementation of the exporter package, concerns writing the AutAT elements, such as the *UserActions element*, *Text element*, etc., to a Watir script-file. This mainly includes changes in *PageElementConverter* and *TransitionConverter*, as presented below.

Wait-for-condition on *traditional* elements
Figure 6.3 shows the method that is used on all AutAT elements, such as *Textfield*, *Button*, *Checkbox*, *Context*, etc., with the exception of Text. It makes sure that if an element in a web-application is not present, the script will wait at least two seconds before it concludes with that the element does not exist.

Wait-for-condition on *Text*
The method in figure 6.3, contains the method "exists" which does not work

```
/**
 * @param element The element that are checked to be present.
 * @param type The type of the value representing the element in Watir (name, value, etc
 * @param value The value representing the element on the wep-page.
 * @return A Watir statement that will wait for the element, maximum 2 secounds.
 */
private String appendWaitStatement(String element, String type, String value) {

    return "count = 0 \n\t\t while not ie." + element + "(:"+ type+ ", \""
    + value + "\").exists? and count < 20 do \n\t\t\t " +
            "sleep 0.1 \n\t\t\t " +
            "count += 1 \n\t\t " +
            "end \n\t\t ";

}
```

Example of Watir script:

```
begin
        count = 0
        while not ie.text_field(:name, "q").exists? and count < 20 do
                sleep 0.1
                count += 1
        end
        assert_equal("", ie.text_field(:name, "q").getContents())
        passedSteps += 1
rescue => e
        puts("Step failed: Check text field present with name = 'q' and value = ''")
        failedSteps += 1
end
```

Figure 6.3: Method that writes a wait statement in a Watir script

```
found = false
count = 0
while found == false and count < 20 do
    begin
        assert(ie.contains_text("ruby on rails"))
        found = true
        passedSteps += 1
    rescue => e
        count += 1
        sleep 0.1
        if ( count == 20 ) then failedSteps += 1
            puts("Step failed: Check text present: 'ruby on rails'")
        end
    end
end
```

Figure 6.4: Watir script that checks if Text is present

on Text objects. The script for waiting on text to appear and therefore needs to be written differently. Figure 6.4 shows an example of test script crated with AutAT. The script tries to assert that the text is present several times, with a limit of two seconds. This way, the web-application have time to load the text into the web-page, if it is not present.

Wait-for-condition on *Context*

In AutAT, a "simple context", as well as an "ordered context", is used to test that a set of elements are placed together on a web-page. In the "simple context" the elements can be in random order on the web-page. In an ordered context, however, the elements must be placed in the same order as in the AutAT test. How to create Watir test-scripts to test, that elements are placed together on a web-page, depend on which HTML-tag that is used to create the context. A context can in a web-page be created by several HTML-tags such as table, span, tr, td, thead, div, and p. Table 6.1 explaines the function of these tags. We decided that contexts in AutAT would be related to Div-tags in HTML since the Div-tags were the most general. All the tags

supported the optional attributes, such as mouseOver, onClick, etc., that we have selected AutAT to support.

Tag	Function
Table	"Defines a table used for tabular data."
Tr	"Table row. tr elements must appear within a table element."
Td	"Table data cell. If the cell contains a header rather than data, th should be used instead. td must be used inside a tr element."
Span	"Used to group in-line HTML. span applies no meaning and is commonly used solely to apply CSS."
Div	"Division. Defines a block of HTML. Commonly used to apply CSS to a chunk of a page."
P	"Paragraph."
Thead	"Table header. Along with tfoot and tbody, thead can be used to group a series of rows. thead can be used just once within a table element and should appear before a tfoot and tbody element."

Table 6.1: HTML-tags that can be used for creating contexts on web-pages [19]

A Watir script can get references to elements in the browser, such as buttons, textfields, text, contexts, etc., through the Internet Explorer web-page object (using the DOM). References to elements inside a Div-tag, except text strings, can similarly be found by querying the Div-tag object for elements it contains. For testing that a specified text is present in a context, all the text in the Div-tag in the web-page is first found, and then a method in Ruby is used to check if the specified text is a substring of the text in the Div-tag.

Contexts in AutAT can be nested. This means that a context can be put inside another context, which can be put inside a third context, etc. This is however, difficult to test with Watir scripts today. Watir was originally not created for testing nested elements [26]. A script can therefore test that the elements are on a web-page, or in a Div-tag on the web-page, but it cannot test that the Div-tags are inside other Div-tags. We believe that this problem can be solved by using a new version of Watir that is claimed to support xPath queries [10]. This version of AutAT, therefore, does not support testing nested contexts.

Watir script for a *UserActions* element
When specifying the *UserActions* in AutAT, the elements upon which actions are preformed are selected from a drop-down menu consisting of the elements in the start state. The Watir tests created from the *UserActions-transition* is run after the testing of the start state (the state where the transition starts). Since wait-statements are executed on all elements not present

```
/**
 * Converts a PageElementAction to a Watir Step.
 *
 * @param fi The FormInput to convert.
 * @return A step representing the input or <code>null</code> if no input is required.
 */
private ITestStep fromPageElementAction(PageElementAction fi) {
    PageElement element= (PageElement)fi.getElement();
    String action = fi.getAction();
    String input = fi.getInput();
    TestStep step = null;

    if (element instanceof Button) {
        if(action.equals(ACTION_CLICK)){
            String label = FormElementUtils.getValue((FormElement)element);
            step = new TestStep( "ie.button(:name, \"" + label + "\").click");
            step.setDescription("Clicking button with label '" + label + "'");
        } else if(action.equals(ACTION_MOUSE_OVER)){
            String label = FormElementUtils.getValue((FormElement)element);
            step = new TestStep( "ie.button(:name, \"" + label + "\").fireEvent(\"onmouseover\")");
            step.setDescription("FireEvent 'onmouseover' on button with label '" + label + "'");
        } else {
            throw new IllegalStateException("UserAction was not possible to preform");
        }
    } else if ( element instanceof TextField) {
        if(action.equals(ACTION_INSERT_TEXT)){
            FormElement fe = (FormElement) element;
            String name = FormElementUtils.getNameCamelled(fe);

            step = new TestStep("ie.text_field(:name, \"" + name + "\").set(\"" + input +"\")");
            step.setDescription("Inserting into input field with name '" + name + "', value '" + input + "'");
        } else if (action.equals(ACTION_MOUSE_OVER)){
```

Figure 6.5: The ”fromPageUserAction”-method which exports an user’s action to a Watir script test step

in the start state of every transition, it is sufficient to perform actions on these elements and assume that they are present in the web-application. If they are not, they will not appear later since the test script has already waited for them to appear. Exceptions can happen if i.e. a Ajax-request for an element is delayed after it is sent to the server, for more than the maximum waiting time on an element. However, we believe that this will most likely not occur often.

In the implementation of AutAT a UserActions-transition object consists of a set of ”PageElementAction”-objects. A ”PageElementAction”-object contains the element, action and input, which is used to model an action on a web-page. A ”TransitionConverter”-class goes through each ”PageElementAction” in a ”UserActions”-transition, and builds a ”TestStep”-object containing the Watir script code. Figure 6.5 contains the first part of the ”fromPageElementAction” which performes this task. The ”TestSteps” created in this method are used to build a complete Watir script.

The challenge with writing ”UserActions”-transitions to a Watir script, is to create Watir statements for each action possible on the AutAT-elements. Figure 6.6, shows the actions that is supported by the different elements in

	Mouse over	Click	Insert text	Key pressed
Link	X	X		
Button	X	X		
Checkbox	X	X		
TextField	X		X	X
TextArea	X		X	X
Context	X	X		

Figure 6.6: Actions possible on AutAT Elements

AutAT. A cell is colored black if the action is not possible to perform on such element in a web-page. An ”X” shows that the functionality is implemented. We have focused on modelling the most common user actions to limit the scope of this thesis. However, several actions will be supported at a later time, if assessment indicates that this version of AutAT is useful in software development projects.

6.2.4 AutAT UI

The AutAT UI package, as explained in subsection 4.1, is responsible for the Graphical User Interface (GUI) in the AutAt workarea. Changes were done according to the graphical changes presented in 2.3. These changes included, i.e., creating a new wizard so that the user can specify the user actions in a ”UserActionTransition”.

6.3 User Documentation

The installation procedure for installing AutAT in Eclipse and how to create an empty test in AutAT, are explained in appendix B. To model states in AutAT, it is important that a user understands the state-concept, which is therefore described. An example will also be shown of how to model a test in AutAT.

6.3.1 Explanation of States

AutAT enables users to create tests for dynamic web-pages by using states. A dynamic web-page is a page where some of its content can change, without the white page flicker given by a ”refresh command”. Changes of the content in a dynamic web-page, can be the result of actions performed by a user. An action such as changing the position of a mouse so that it is over an item in a top-menu can result in that a sub-menu appears. A state is a “snapshot” of the content on a web-page at a give time. For the above example, this

means that one state of the web-application is where it contained only the top-menu, and another state is where it also contained the sub-menu.

6.3.2 How to Create Tests in AutAT

To illustrate how a dynamic web-page can be modelled, we will present how to create a test for the Google Suggest[1] web-page. Google Suggest is a search engine that shows suitable suggestions for the complete search word as the user inserts letters into the search field. Before the test is created, it must have a Start-Point. How to provide a Start-Point for the test can be found in appendix B. When the Start-Point has been defined, an empty test appears in the Editor-window of Eclipse. Is it now possible to start modelling the example page with the needed elements, such as states, aspects, elements and UserActions-transition. A test in AutAT can be created by following the below steps.
The test in figure 6.9,

1. Drag a new state into the workarea to add it to the test, and call it ”Start”. Insert a name into the pop-up box that appears on the state. When this is done, an arrow connecting the Start-Point and the first state will appear.
2. Drag a ”title” to the state, click on it and insert the text ”Google Suggest”. It will then test that the title is ”Google Suggest” in the actual web-page.
3. Add a textfield with the text “q” and a button called “btnG” to the state. These first steps will result a finished first state, as shown in figure 6.7
4. Drag a new state into the workarea. If you want to test that the elements in the first state are still present in the second, these can be added in an ”aspect” instead of the first state, and connected to both states.
5. Drag a new state into the workarea, and call it ”Suggestion”.
6. Drag a UserActions-transition into the test and click on both the start-state and the end-state. A wizard will appear, and in the wizard do the following steps:
   (a) Click on the cell that contains the text ”select” in the row named ”element”, and select the element ”q” in the drop-down list.
   (b) In the same row, select ”InsertText” in the action-column, and write ”Ruby” in the input column, as shown in figure 6.8.
   (c) To model several user inputs, press the button ”Add New User Input”.

[1] http://www.google.com/webhp?complete=1&hl=en

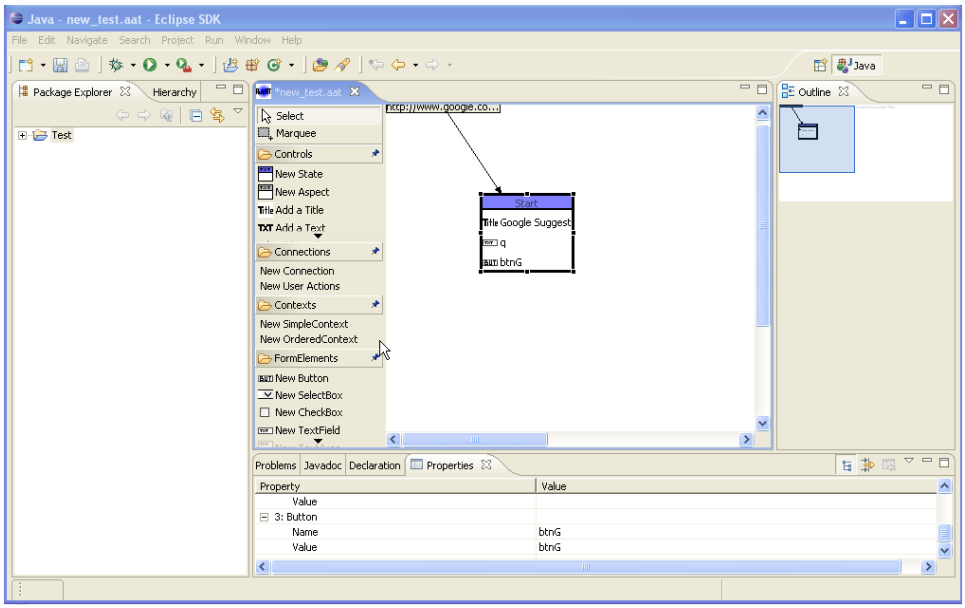

Figure 6.7: First state in Google Suggest modelled in AutAT

(d) Press ”Finish” to complete the wizard.

7. Add a text into the ”Suggestion”-state and type ”Ruby on rails”. The result of the test is shown in figure 6.9.

6.3.3 How to run Watir test in AutAT

To create a Watir test in AutAT, right-click when the mouse is over the name of the test, and choose ”AutAT -> Create Watir Tests”. Then AutAT will create a ”Ruby”-file with the Watir tests. The tests are run by right-clicking on the Ruby-file and choosing ”Open with -> System Editor” with Ruby and Watir installed on the computer.

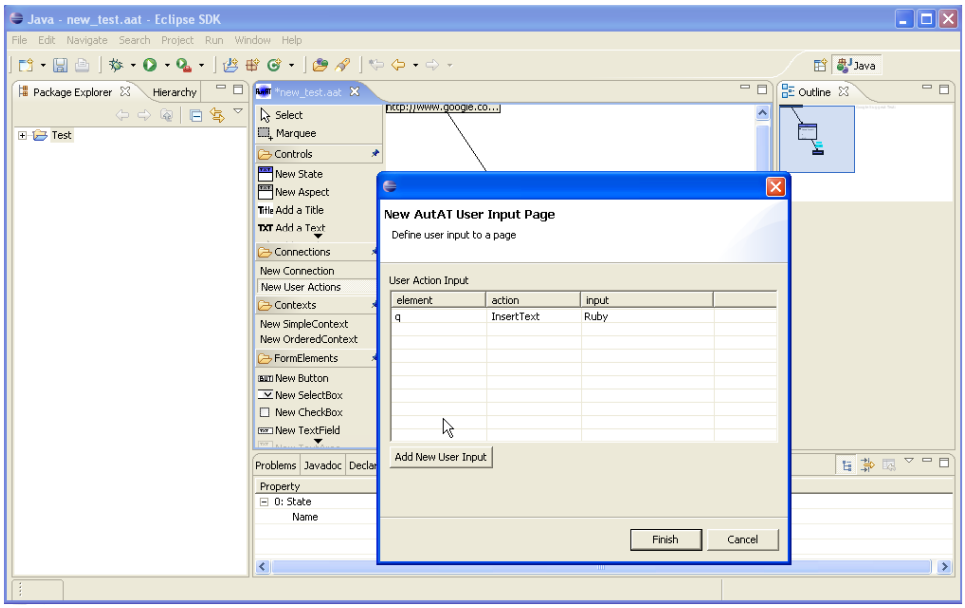

Figure 6.8: UserActions in Google Suggest modelled in AutAT

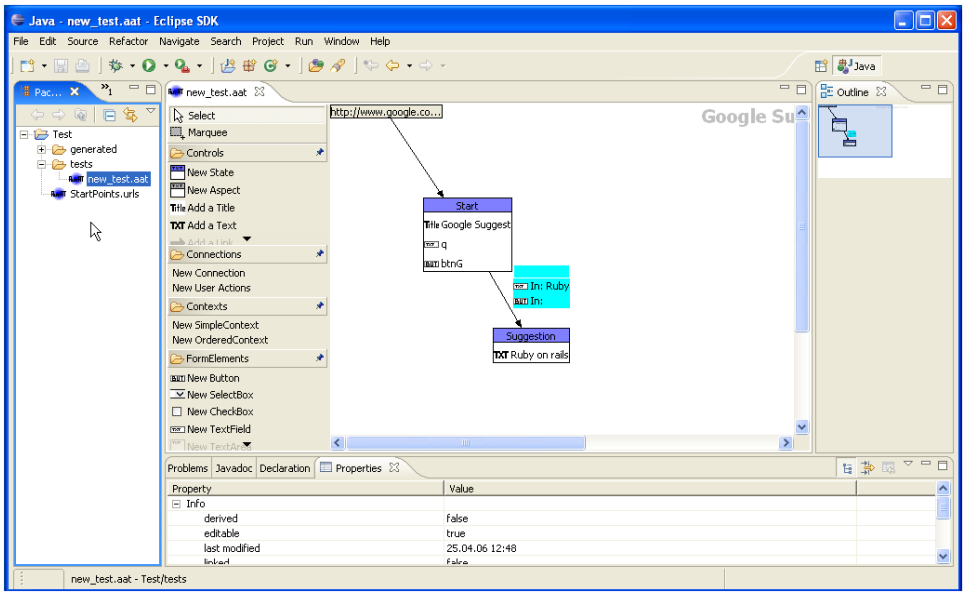

Figure 6.9: The Final Result of Google Suggest modelled in AutAT

Chapter 7

User Testing and Results

If both customers and developers could use AutAT to create respectively acceptance- and system tests, these tests could be used and modified by both parties since they both tests the functionality of an application. We will therefore evaluate the usability of AutAT for non-technical users, as well as developers. In addition, we investigated if AutAT is useful for customers and developers in software development projects.

7.1 Testing with Non-Technical Users

To test the usability of the new version of AutAT for non-technical users, we tested the tool on eight persons with a non-technical background. These persons, from now called subjects, had different age and occupation, as can be seen in table 7.1.

Subject nb	**Age**	**Occupation**
Subject 1	23	Social Worker
Subject 2	23	Master Student in Chemistry
Subject 3	23	Master Student in Petroleum Engineering
Subject 4	24	Master Student in Petroleum Engineering
Subject 5	25	Master Student in Civil Engineering
Subject 6	26	HSEQ (Health, Safety, Environment and Quality) Manger
Subject 7	48	Teacher
Subject 8	57	Bachelor of Economics

Table 7.1: Non-technical subjects

7.1.1 Test process

We created a fictitious software development case, for which we wanted the subjects to create tests. The test sessions were performed by first giving the subjects a written presentation of the case, which also explained acceptance testing and how this can be done manually or with automated test scripts. The presentation also contained an introduction to AutAT, and an example of how to create an acceptance test for the Google Suggest[1] web-page. The presentation can be found in appendix C, which also includes the tasks given to the subjects.

The subjects were allowed to ask questions about AutAT during the test session. When they had created tests for the fictitious case, we asked them about their opinions on the usability of the tool. The questions included how easy they thought it was to understand the concepts in AutAT, and to create the tests. We also asked question about what they liked and disliked about the tool to capture their general opinions about AutAT.

7.1.2 Results

The results presented here is a summary of the comments given by the subjects during the test session. The comments can be found in appendix E.

The subjects had trouble using AutAT after they had read the presentation because they were not familiar to the technical terms ”state” and ”user action”, and they therefore had difficulties understanding the concepts ”State” and ”UserActions”-transition. A subject stated that the reason for his trouble was that he did not have anything familiar to relate to the concepts. Especially, the term ”state” was difficult, because it was not something tangible, like a web-page that they could look at. The general opinion was that AutAT would be easier to use if they could model web-pages instead of states. However, the subjects said that they understood the concepts in AutAT after we had guided them through an example.

For the majority of subjects, it was easier to understand UserActions-transitions than ”States”. It was logical that the user of a web-application inserted some input into a web-page, and that the page changed as a response. However, it was not logical that changes on a web-page, as a result of actions performed on the same web-page, should be modelled in a new state. When the subjects were creating tests, a few added only one action into each UserActions-transition, which resulted in several states and UserActions-transitions. However, the general opinion was that UserActions-transition was simple because it resembled actions they normally performed on web-

[1] http://www.google.com/webhp?complete=1&hl=en

pages.

In addition to the technical terms not familiar to the subjects, there were terms they had not heard of, such as textfields, textarea, etc. The subjects therefore asked several questions during the test session, which resulted in that their work was not efficient. Their general opinion was that AutAT would have been easier to use if a test could be related to an actual page i.e. on a picture in the tool, which would show them how the AutAT elements looked like in an actual web-page. The picture did not have to include a whole web-page, only the functionality in the model. The subjects thought that they would create tests more efficiently in cooperation with a person that knows how to use AutAT. However, they would only need such cooperation the first times they used AutAT, because it was easy to learn with its tidy appearance and drag-and-drop feature. When they had learned how to use it, they believed they could use it efficiently by themselves. A subject thought that a user manual with step-by-step instructions explaining all elements in AutAT would help him to create tests.

A subject stated that AutAT is better to use than drawing a web-page by hand, because tests in AutAT were easy to manage and edit. AutAT allows a user to easily add and delete elements, as well as modify an element's properties. However, the fact that changing an element's properties could be done directly in the Editor as well as in the Property Editor, was confusing for some of the subjects. Some subjects therefore meant that it should only be possible to change these in the Editor. Four of the five subjects needed to modify the UserActions-transition they had created, and therefore also missed the possibility of changing the UserActions-transition from the Editor, which is not yet implemented.

7.2 Testing on Developers Familiar with Watir

AutAT will not be used by developers if it lacks important functionality or if it is more efficient to write test manually, even if the AutAT tests can be used and updated by customers. It is therefore important to investigate if test-scripts can be created more efficiently with AutAT than manually. This was done by testing AutAT on developers that have used Watir in software development projects. Two of the developers had been responsible for creating Watir-tests for a web-application in a project that lasted about a year. The three other developers had also created tests with Watir in one project, but these project lasted less than six months.

7.2.1 Test process

Eeach test session was performed by giving the developer a short presentation of AutAT. We then let the developer use the tool to create system-tests

similar to those he/she normally created in development projects. During the test sessions, we were present to answer questions about AutAT. After the developers had tested the tool, they gave their opinions about using AutAT to create test scripts. We were interested in information about if they thought AutAT limited what test scripts a user is able to write. In addition, we wanted to know their opinion on the usefulness and usability of AutAT.

7.2.2 Results

These results are a summary of the main opinions from the test sessions with the developers. The developers understood the concepts; State, UserActions-Transition, etc., and they had no problems with using AutAT. The graphical user interface (GUI) was easy to understand, and they thought the drag-and-drop functionality made the tool efficient to use.

The general opinion from the developers was that AutAT could enable them to create automated tests more efficient than writing scripts manually. However, before they could know this for certain, it must be tested on a real project. An advantage with using AutAT was said to be that spelling errors would not occur, which often happened when making scripts manually. In addition, developers would only need one development environment for writing code, tests, integrating the application, etc., which can all be done on the Eclipse platform. For these reasons, the developers meant that AutAT could be useful for creating tests when developing web-applications.

A developer stated that it was easier to present tests in AutAT than Watir scripts, to customers and other stakeholders since AutAT has a graphical user interface which makes the tests visual, and therefore easier to understand for non-technical users. One developer stated that it was a minority of developers that knew how to create tests in Watir, and these developers had to create all the tests scripts for an web-application. It would therefore be an advantage if developers could create automated tests without needing to learn the syntax for how to write tests in a test framework.

A developer wanted additional test information, such as the coverage of how much of the code that have been tested. With coverage information, it would be easy to see which lines of code that were executed after a test has been run. The developer could then create better- or more tests to increase the coverage of testing.

Two of the developers interviewed, worked in a project where they created their own Watir-methods for custom-made HTML elements. Hence, they could not have used AutAT in their project since AutAT requires that the application is implemented according to the exporter. However, these devel-

opers believed that it was not common to use custom-made HTML-elements in most projects.

7.3 Testing with Project Managers

We have interviewed project managers in five companies in Trondheim and Oslo, to investigate if and how AutAT can be useful in software development projects. The project managers have experience from several projects and are hence familiar with projects' needs.

7.3.1 Test process

In each interview we presented the general concepts of AutAT, and its intentional use. The presentation included movies of AutAT showing how to model tests of the Google Suggest web-page. We then asked questions about the usefulness of AutAT in software development projects, and emphasized that it was not restricted to the intended use. We wanted to know if AutAT could be used in any of the companies' projects, in addition to opinions about how AutAT could be changed to become more useful. The presentation of AutAT, and the questions asked in the interviews, can be found in appendix D. The example movies can be seen in the files that follow this master thesis.

7.3.2 Results

The results presented here gives a summary of thoughts and opinions from the interviews in companies, without relating them to any of the participating companies. AutAT can be used either by customers, developers or in cooperation between both, and each use has different needs and hence, different usefulness of AutAT. We will therefore present the results from each user's point of view.

Used by customers

Several project managers meant that AutAT could be useful for customers since it enabled them to create automated acceptance tests, which can be run without much effort. With automated tests several parts of a web-application would be more easily tested after changes and updates in the application, than if testing was performed manually. These tests can in addition, be run during development by the developers.

A developer meant that another advantage with using AutAT, was that an AutAT-test modelled the workflow of applications. Modelling the workflow of an application should be done by the customers since they are usually

more familiar to the workflow of the intended users, than the developers. The project managers agreed on that the main advantage with AutAT was that it increased customers involvement in the development process, because it enable them to create automated acceptance tests.

Even if the project managers saw several advantages with AutAT when used by customers, they had some concerns. This was mainly that it would be too time-consuming for customers to use AutAT, especially to create tests for large applications. It was however also stated that it was too time-consuming for customers to create and maintain automated tests in general. If customers used AutAT to create acceptance tests early in the development process, it would often result in that changes and updates were needed. Changes and updates often occur because customers' requirements usually change during the development process. Customers are often uncertain about what they want, and what it is possible to create. If the acceptance tests are created after the application is developed, the effort of updating the tests is avoided. Another project manager therefore meant that AutAT would be useful for customers to create automated acceptance tests only after an application is developed. Another reason for this is because he meant that in addition to create automated AutAT-tests, the tests also needed to be documented. This because often, and especially in large projects, it is necessary to give the documentation to other organizations involved in the project. It will then be necessary to update both the automated test and the test-document if an application was changed, which they often did according to the developer.

A different opinion was that customers would not use AutAT because they often preferred to perform the acceptance tests manually by going through written test specification. This because it gives them an impression of how easy it is to use the web-application, which the customer will not get when using AutAT to create automated tests. This because in tests created with AutAT, the elements that are tested are specified by i.e. id, and the location and appearance of the elements are therefore insignificant.

For a web-application, a test in AutAT defines i.e. the actions a user is be able to perform, the text a user will see, etc. For this reason, it was said that the user decided the functionality on each web-page in the web-application, and therefore decided parts of the design. Another project manager stated that customers would try to decide the layout of the web-application when using AutAT. Developers can design web-applications better than the customers, and for that reason they claimed that AutAT should not be used by customers.

Used by developers

The project managers that used automated system- and acceptance tests in their projects, were unsure if AutAT would be useful for developers. This because they they were not convinced that developers would save time on creating test-scripts with AutAT, instead of writing the scripts manually. Developers have a technical background, which makes it easier for them to learn the syntax used in test frameworks. However, the five project managers interviewed did not agree upon if automated tests in general, reduced the time and effort used on testing. They were therefore unsure if AutAT would be useful when used by developers alone.

Project managers that did not use automated system- and acceptance tests in their projects, believed that AutAT would be useful for developers. This because it enabled them to write automated test-scripts without learning a new test-framework. Without automated tests, it was said that testing was often performed on only the most necessary parts of the application to limit the effort used on testing. Testing only the most necessary parts of the application, was not a solution they preferred.

AutAT has not been tested on a real project. The project managers were therefore critical to how much of the functionality in their web-applications AutAT-tests could test. This especially concerned exceptions from the normal execution path, which is usually tested by the developers. It was stated that it might be better to write test-specifications since exceptions from the normal execution path are easier and faster to describe with words than in AutAT.

Used in cooperation between customers and developers

The general feedback from the interviews was that AutAT would be most useful when it is used in cooperation between developers and customers. This would be the case even if AutAT was simple enough to use for customers to model tests without help. It was said that the best result was accomplished when developers and customers cooperate. A suggestion was that during the specification phase, the developer and the customer could have a workshop and agree upon a web-application to develop. This way, the customer would not need a requirement specification. Instead, the customer and developer could express the requirements by modelling tests in AutAT, showing the workflow and the possible solutions.

Modelling a real test-specification

To investigate if AutAT can be used to test the functionality in common web-applications in a company, we were given a medium-complex test-specification

document from a project manager. This test specification document contained functionality for registering projects and checking that project information was presented correctly. Together with the project manager, we went through the test specification document and created tests in AutAT. This process can be seen as a proof that AutAT can create test scripts for this company's web-applications with medium complexity, which the project manager meant was useful in their projects.

When creating tests for the test-specification document, there was some missing functionality in AutAT to completely test the web-application. The test specification included checking the order of elements in a list. In AutAT, lists can be specified by putting elements a contexts, but to test that the elements are presented in a given order requires that the elements are specified in that order in the context. When tests are run repeatedly with elements that are stored in a database, the test-data depends on the state of the database for each test-iteration. In example, the test specification document required that a list of projects was updated when a new project was registered. When running such tests in AutAT, the tests must either be updated or the state of the database must be kept constant for the AutAT-test to correctly check the order of the elements in a list. This functionality was also requested from other companies.

Another missing functionality, is the possibility of modelling "if/else"-clauses. In the given test specification document, if/else-clauses would be used to model that if a project exists, check its data. If a project does not exist, then create it. In addition, it was said in another interview that AutAT lacked the possibility to insert descriptions in the transitions between states. This was needed to be able to specify i.e. dependencies between states. It was also said in an interview that AutAT supported only limited types of input. I.e. AutAT does not support dates, which makes it impossible to tests i.e. that a set of data can not be changes after a particular date.

7.4 Threats to Validity

We will look at the most important threats to validity, with basis in the book of Wohlin et. al. [41].

Typical customer

In the interviews with non-technical persons, we wanted to interview persons that could represent typical customers to evaluate the tool from a customer's point of view. However, the subjects we have chosen, does not necessarily match the customers in experience and knowledge. This means that there is a threat to validity of the generalization of these results. Most customers do not have a technical background, but we cannot say that there is a standard background or gained knowledge for customers. None of the subjects

interviewed had technical knowledge, but to reduce the threat that they still could not represent customers, we interviewed people of different age and occupation, and hence different experiences. We believe that the scattering of our subjects reduced this risk, but it is still a relevant threat for the evaluation of the test-results.

Developers and Project Managers

When we interviewed developers and project managers, we wanted results that we could generalize to all companies that develop web-applications for customers. We therefore interviewed developers and project managers from several companies, which have mainly been working on different projects. However, two of the developers were currently working on the same project, which may have influenced their answers. In the interviews with project managers, they gave answers based on what they though customers and developers meant about the usefulness of AutAT. The project managers we interviewed have much experience from working with developers and customers, and hence, they are familiar with their needs. We see these threats as present, but believe they have minor impact on the test results.

Sample size

The sample size used in the interviews with non-technical users, developers, and project managers, consist of respectively eight, five and five persons, thus the sample size is small. People understand problems differently, and answers are given based on their understanding of the problem. With a small sample size, it is less likely that the persons interviewed have the same understanding of the problem and that the answers express similar opinions. However, we have tried to avoid this threat by giving all persons within each group (customer, developer or project manager) the same introduction to AutAT, and the same example of how to model the tests. They have in addition been allowed to questions if anything in AutAT or any tasks were hard to understand. We believe that this limits the effect from misunderstandings and short explanations, and therefore has this threat has been reduced to only having minor effect on the test results.

Tasks and questions

The tasks given and questions asked in the interviews were decided by us. These were specific towards the areas of AutAT that we wanted to evaluate. It is therefore a threat that there were other opinions about AutAT which were not revealed. We limited this threat by also asking for opinions about AutAT not covered by the questions. If the tasks and questions had been more general, the answers from the interviews would be difficult to compare and analyze.

Another threat is that the persons interviewed want to be ”nice” and give us the answers they believe we want to hear. To avoid this, we explained to

the persons interviewed that we were interested in their own opinions, and that an honest review was important. We, however, believe that this is the substantial threat to our results.

Summary of threats

We think that the majority of threats influencing our results have been reduced to an acceptable level. The threat that subjects answer what they think we want to hear is a considerable threat. We take all threats into consideration when evaluating our results, but we mainly believe that only the latter threat can significantly influence our results.

Chapter 8

Evaluation and Discussion

To assess if we have reached the main goal of our thesis, this chapter discuss and evaluates the new version of AutAT based on the tests and the interviews presented in chapter 7.

8.1 The New Version of AutAT

There are several differences between AutAT and other state-of-the-art test tools. The most important is the graphical user interface (GUI) in AutAT, which makes it easier for non-technical users to create tests, than with text based tools. AutAT can be used for creating automated acceptance tests before the actual software is developed. To our knowledge, this is not possible with other GUI based tools, such as capture-replay tools. Tests in capture-replay tools are created by traversing a web-application, which requires that the web-application is created. Tests in capture-replay tools need to be updated after the GUI of an application has been changed. AutAT is not as sensitive to changes as capture replay tools, because it reference elements in the browser based on the elements' name. This results in that elements can be moved to another location in the browser, in addition to that the visible text on elements can be changed.

Because AutAT is only a graphical layer on top of an existing test framework, it is restricted by the functionality in the test framework. The new version of AutAT can therefore only test what is supported by Watir. However, AutAT is an open-source tool which can be customized to fulfill additional needs. This can be done by adding several exporters, which i.e. support testing custom-made HTML elements.

Today, Ajax is a relative new technique, and we therefore believe that it has not been fully taken into use in software development projects. Customers of development projects are usually not updated on new web-technologies, since most do not have a technical background. AutAT does not support

testing dynamic changes on a web-application as a result of changes in the server's state, which is a feature enabled by Ajax, see section 5.3.2. We do not know if this feature is used in web-applications and, in case, how important it is for customers of software development projects to automatically test this feature. This should be further researched when Ajax becomes more widespread.

8.2 The usability of AutAT

The goal of our thesis was to create a tool that can be used by customers and make acceptance testing simpler and more efficient. Research of how testing is done in companies, shows that the acceptance tests sometimes are created by developers. In addition, acceptance tests can be used in system testing performed by the developers. We will therefore evaluate the usability for non-technical users, as well as developers.

8.2.1 Usability for the non-technical subjects

The test-results indicate that AutAT was difficult to use for non-technical users, mainly due to the technical terms on elements and trouble with understanding States. The subjects had trouble with relating elements in AutAT to elements on a web-page. For example, they could not relate the term textfield to the visual representation of a textfield in an actual web-page. We believe that providing the users with a visual representation of the elements in AutAT, could solve this problem since most users are familiar with web-pages. The persons we interviewed were Norwegian, and therefore not native English-speaking, which also can affected their understanding of the terms used in AutAT. This problem could be solved by providing a User Manual that explained each term used in AutAT and mapped them to the corresponding element in a web-page. We could in addition have more informative pop-up messages for the elements in the palette in AutAT. For example, the message "Creates a new TextArea" could be changed to "Used for inserting a large amount of text in a form".

States were difficult to understand for the subjects since they did not have anything definite to relate to it. Our subjects were not familiar with other software diagrams, such as State- and Activity diagrams [40] that are used to describe the behavior of a software system. However, our subjects understood States after a thorough explanation with examples. Subjects stated that they would prefer modelling web-pages instead of States. Modelling pages would result in that users must understand Ajax and JavaScript to know if a page is reloaded or if it is only dynamically changed, without the need for reloading the whole page. This is avoided with States, which is crucial for non-technical users to enable them to use AutAT for modelling dynamic behavior correctly.

The non-technical subjects interviewed, thought it was difficult to understand AutAT in the beginning of the test session. However, all the subjects were able to create test in AutAT during the test session. This shows that AutAT is easy to learn, which were also confirmed by the majority of the subjects.

A functionality that increased the usability of AutAT, was the ease of changing the modelled tests, even if the Property-Editor was confusing for some of the subjects. We did not explain the Property-Editor to the subjects, and they had therefore no qualifications to understand it. Users are able to create tests in AutAT without using the Property-Editor because it is used to specify additional properties to elements, than properties that can be defined in the general Editor.

As a summary the test-results show that the subjects needed proper training before they could use AutAT, but it was easy to learn. There are some improvements that can be implemented to increase the usability for non-technical users. However, the test-results indicates that AutAT can be used by non-technical users.

8.2.2 Usability for developers

All the developers interviewed meant that the usability of AutAT was good. This because they did not have problems with understanding the concepts or creating tests. The names on the elements used in AutAT were familiar to the developers, and the graphical user interface was well arranged. In addition, the drag-and-drop feature enabled the developer to quickly add elements to a test. For these reasons, developers could efficiently create tests when using AutAT. These results indicates that the usability of AutAT for the interviewed developers, is good.

8.3 Is using AutAT designing the application?

A project manager stated that a reason for why AutAT could not be used by customers, was because a web-application was designed when creating tests for it in AutAT. If this is the case, AutAT should not be used by customers since developers have more knowledge about design. There are two issues when discussing if using AutAT implies designing the web-application. First, using AutAT can be seen as design since a user selects which elements that the web-application will consist of. Second, when a user have added i.e. a context to the AutAT-test, the application must be implemented with Div-tags because the Watir exporter creates tests based on that assumption.

An argument for why using AutAT implies designing the application, was that the user of AutAT decides the functionality on each web-page in the

web-application. However, it is only possible to model states and not web-pages in AutAT. A State in AutAT is mapped to a state in the web-application that is developed. When a user models states in AutAT, he decides the functionality in the corresponding state in the web-application, i.e. which actions that can be performed. It is the developers that decides which web-page that will contain the State, as well as the UserActions-transitions from that state.

If a user wants to test that a text in a web-application can be changed from English to Norwegian, he must decide how to enable this action. This can in example be done by pressing a button named ”Norwegian”, or having the text ”Norwegian” with an ”onClick”-event attached to it. Both can be modelled in AutAT. However, the user must decide which option he think is best for his application, which can be seen as designing the application.

A project manager said that customers would try to decide the layout of the web-application when using AutAT since the visual content on a web-page is important for customers. It is not possible in AutAT to decide the location of elements in the browser, nor how each web-page in the web-application will appear. This can be confusing for customers. To solve this problem, it was suggested that the concepts in AutAT where changed so that the user associates a test in AutAT with the workflow of an application. We do not know if this is a good suggestion. However, we believe that customers understand that they shall not create the layout of the application, after they have learned how to use AutAT.

AutAT puts restrictions on how a web-application can be implemented. This because an element in AutAT does not always map to an element in HTML. For example, a texfield in AutAT can be implemented with a textfield-tag in HTML. A context however, as discussed in 6.2.3, can be implemented with several tags such as Div, thead, P, Span, etc. The Div-tag is the most general tag and was therefore selected to be used to represent contexts in AutAT, when writing the Watir exporter. This means that AutAT restricts the developers to use Div-tags for implementing contexts if the tests shall run without errors. The restriction can be avoided by adding more specified elements to AutAT, such as tables and paragraphs. However, this will increase the number of necessary design decisions for customers, when creating tests in AutAT.

Another example where AutAT elements does not map to elements in HTML, is when custom-made HTML-elements are created and used in a web-application. This shows that AutAT puts restrictions on how a web-application can be implemented, and therefore can be seen as designing the application code. In example, if a customer had used AutAT to model an application, custom-made elements could not be used to develop the application. In the test

session it was said that custom-made elements in HTML are not common. AutAT is not meant to be used in all projects, and we therefore believe that AutAT will be used even if it cannot create tests for custom-made HTML-elements.

We have shown several arguments for why using AutAT can be seen as designing web-applications. The most important reason is because the user of AutAT must decide how a functionality should be implemented, i.e. with a button or ”on-click”-event. However, the user cannot specify the layout or where elements are placed in a web-page. We conclude that a user of AutAT partly designs a web-page when using AutAT, but we still believe that AutAT can be used by customers.

8.4 The usefulness of AutAT for customers

Test-results show that there are several advantages for customers when using AutAT. The main advantage is that it enables customers to create automated acceptance tests, which were seen as useful by all project managers in the test session. However, the test-results also indicates that customers will not take AutAT into use, because creating tests with AutAT is seen as time-consuming for customers when it is performed early in the development process. Customers' requirements often change during development, which results in updates and changes in the application, as well as the automated tests. The results from the assessment show that it is not common to change the elements in the graphical user interface (GUI) of a web-application, but only their location and visible names in the browser. Such changes do not require updating tests in AutAT. In addition, if changes are required, the results from the usability testing shows that changing tests in AutAT is easy. For these reasons, we do not believe that the effort used on updating AutAT-tests are too significant for the tool to be taken into use.

The results from researching state-of-the-practice of testing, shows that customers traditionally perform acceptance testing manually, by clicking on links, buttons, etc., which gives them an impression of how easy the web-application is to use. However, this impression is provided by using the application once, if the graphical user interface has not been significantly changed. Significantly changing the GUI of an web-application, is not performed often according to the test-results. Automated acceptance tests can not provide customers with information about how easy it is to use a web-application, which was another reason for why the project managers thought customers would not take AutAT into use. We do not mean that usability testing should not be performed by customers. However, we believe that the effort used on usability testing and creating automated tests in AutAT, is less that the effort gained by using automated tests, especially when using Test-Driven Development (TDD). This because when web-applications are

developed in several modules or iterations, which is the case in TDD, the test-results show that a substantial amount of testing is needed.

Even if customers are critical to AutAT, most project managers believe that customers will benefit from using AutAT since automated tests can be run often (also during development) to make sure that the whole web-application still works correctly after changes in the application code. Testing the whole application instead of the most important parts, increases the confidence in the application. However, most project managers did not believe that it would be beneficial for customers to create automated acceptance tests for the whole web-application. Their general opinion was that around 70/80 percent of the application could be tested with automated acceptance tests. A higher percent would not be beneficial due to the costs of creating and updating the tests.

Based on the test-results, we believe that it would be useful for customers to use AutAT to create automated acceptance tests for web-applications. However, we also believe that AutAT will not be taken into use by customers in the near future. This because the test-results indicates that customers are not convinced that they will benefit from using automated acceptance tests.

8.5 The usefulness of AutAT for developers

The test-results indicate that there are different opinions on the usefulness of AutAT for developers, even if the usability of AutAT was good and allowed developers to efficiently create automated tests. Developers that not commonly used automated test frameworks, saw AutAT as an advantage because it enabled them to create automated tests without learning a new test framework. AutAT-tests were also seen as easy to present to non-technical persons. However, this would be more valuable if AutAT had provided additional test information, such as test coverage of application code.

Companies that did not use automated system- and acceptance tests in their projects, had in general a different opinion on the usefulness of AutAT than companies that used such tests. Companies that commonly used automated tests were sceptical to if test-scripts could be created more efficiently with AutAT than by writing them manually, even if AutAT enabled them to create test-scripts without spelling errors. Developers in these companies already used a test framework to create automated system tests, which they though was adequate. Companies that did not use automated system- and acceptance tests in their projects, meant that AutAT would be useful for developers. This because AutAT enabled them to create automated tests without learning the syntax for how create tests in other test-frameworks.

The test-results show that AutAT could be used to test a medium-complex

test-specification document in a company that do not commonly use automated system- and acceptance tests in their projects. The tests that were created, could not cover the whole application since AutAT lacks functionality for testing lists. However, the project manager still believed that automated tests created with AutAT would be valuable in their development projects. This because common tests often contained inserting values into forms, and checking that elements and text are represented correctly in the browser, which is time-consuming to test manually. We therefore believe that this test specification is similar to what is developed in common web-application development projects. For this reason we think that AutAT can be useful for developers in other companies that develop web-applications.

8.6 Generalization and recommended use of AutAT

We have evaluated the usability of AutAT as sufficient for non-technical users. The subjects showed during the test session that they understood the concept of State and UserActions-transition, by modelling tests in AutAT. This shows that the results is not based on that the subjects were ”nice”, which we identified as a threat in section 7.4. There is also a threat to the generalization of our test-results that the subjects do not represent customers of software development projects. However, the probability of that the subjects have more technical knowledge about i.e. States, than typical customers, is small. Evaluation shows that there are several reasons for why AutAT is seen as useful for customers. Evaluation also shows that some design decisions need to be taken when creating tests with AutAT, but that the tool can still be used by customers. We believe that AutAT presents a challenge for customers, and that they therefore will not take it into use. For customers it can i.e. be a challenge to install AutAT in Eclipse. In addition, it is a challenge for customers to become confident with using automated tests. For these reasons, we believe that customers will most likely not take AutAT into use.

For the interviewed developers, the usability of AutAT is evaluated as good. In addition, there are several advantages with AutAT that can be useful for developers. We do not believe that the threat identified as; developers providing positive answers to be ”nice”, will affect the generalization of the usability and usefulness to most developers. However, it might have influenced the answers concerning the usefulness of AutAT. We believe that AutAT will not be taken into use by developers because there were uncertainties about the disadvantages with AutAT, and because developers in general do not have problems with learning how to create test scripts. Further tests of AutAT can assess if the possible disadvantages will affect the usefulness of AutAT for developers.

The results from testing indicates that it is an advantage to use AutAT

in cooperation between customers and developers. AutAT can then be used during the Requirement specification phase, for example in a workshop where the workflow and user-interactions for an application, can be modelled with tests in AutAT. When this phase is finished, both the customer and developers have automated tests that they can use for testing the application during development. In addition, the customers do not need to create a full requirements specification in the process, because the AutAT-tests represent the main functionality of the application.

An advantage when using AutAT-tests as a part of the Requirement specification document, is that the developers cannot skip reading any of the Requirements. This because if Requirements are written as tests, it will result in that a test fails if the corresponding functionality is not implemented. Another advantage when using AutAT, is that the tests can be used as documentation for the implemented functionality. Since automated tests are run often and must run without errors, they must always be updated. Hence, the documentation is always up-to-date.

The test results show that the best use of AutAT is when it is used in cooperation between developers and customers. This makes the development process more efficient for both developers and customers, by i.e. limiting the effort used on creating and reading a large Requirement specification. This process enables, in addition, the developer to advise the customer when it comes to issues that concern design of the application. We believe that customers are more likely to take AutAT into use if they could do this in cooperation with a developer since the technical challenge of using a new tool is decreased.

Chapter 9

Conclusion

We have in this thesis created an extension of AutAT, which makes a user able to create automated acceptance tests for common dynamic web-applications. Common dynamic web-applications do not include applications that are created with i.e. custom-made HTML elements. Our goal was to create a tool that was easy to use for non-technical users, and make acceptance testing of dynamic web-pages simpler and more efficient. We performed tests with non-technical users, to see if AutAT was easy to use. In addition, we performed interviews in five companies in Trondheim and Oslo, to investigate if AutAT could be used by customers in their development projects. We also interviewed software developers familiar with Watir, to investigate if AutAT enables developers to create Watir test scripts more efficiently.

In this new version of AutAT, dynamic functionality in web-applications is modelled with states. We have shown that the State-concept can be used to model common dynamic web-application implemented with Ajax- and JavaScript. However, state-events are not supported, but we believe that this will not affect the usefulness of AutAT. The State-concept allows users to specify functionality without deciding if it must be implemented with Ajax, JavaScript or static HTML. In addition, it allows a user to easily model user-interactions with the web-application.

Test results show that the usability of AutAT is sufficient for customers to use the tool with proper training. In addition, that there are advantages for customers when using AutAT to model automated acceptance tests. However, customers can be skeptical to use automated tests and that they usually do not have time to create them. Most customers also preferred to perform acceptance tests manually since this gives them an impression of how easy the application is to use. We therefore conclude that AutAT can be useful for customers, but that they will not take it into use before automated acceptance tests, and other tools to create automated acceptance tests, are more commonly used among customers.

Developers have a technical background, and it is therefore easier for them to write automated test-scripts manually. Tests results show that developers are concerned that AutAT lacks important functionality to be used for testing their web-applications. In addition, they were not convinced that test-scripts was created more efficiently in AutAT than written manually. These issues need to be further researched before we can conclude whether AutAT is useful for developers or not.

Based on the evaluation, we conclude that AutAT is most useful when used in cooperation between developers and customers, in i.e. the specification phase of a development project. The design and requirements of an application can be decided in a workshop with developers and customers. In this process they could use AutAT to model the requirements by modelling the workflow and user-interactions with the application. When this process is finished, the developers and customers would have automated acceptance tests that can be used during development. After AutAT is used to model requirements in cooperation with developers, it is more likely that customers use AutAT to i.e. update tests to specify changes, model user actions that have resulted in exceptions, etc. For these reasons we conclude that AutAT can contribute to the development process of dynamic web-applications.

Chapter 10

Further Work

The scope of this master thesis did not include testing AutAT in a real software development project. The conclusion is thus based on testing AutAT in a smaller context. To increase the credibility of our conclusions, AutAT needs to be tested in a real project.

In the new version of AutAT, it is not possible to test contexts that are nested since it is not supported in the Watir-exporter, see section 6.2. A new version of Watir that supports xPath expressions is under development. When that version is released, the exporter should be implemented to also support testing nested contexts. An exporter could also be created to run tests in the AutAT-console, which provides visual feedback showing if a test-step in the test has failed or succeeded. This will make the usability of AutAT better for non-technical users.

A visual presentation of how the elements in AutAT look in an actual webpage can be added to increase the usability of AutAT for customers, see section 8.2.1. This could be done i.e. by showing the visual presentation of a web-element when a user selects the corresponding element in AutAT, or the visual presentation of all web-elements in a State when it is marked by the user, etc. Creating such a visual representation can be an improvement of AutAT, which is possible further work.

As discussed in section 8.5, AutAT would have been more useful for developers if it provided feedback about test-coverage after the execution of the AutAT tests. Useful information could be which lines of code that have been executed during testing. Such information could be provided by i.e. integrating AutAT with other development tools. We therefore recommend for further work, to investigate the possibility for enabling AutAT to provide test execution feedback.

As a result of the asynchronous communication used in Ajax, web-pages

can dynamically change without user interaction, see section 8.1. This is not supported in this version of AutAT. More research is necessary to understand how this feature is used in web-applications. This to create a concept that can model events not triggered by a user, without requiring that users need technical knowledge about servers, or the communication between server and client-application. In addition, it is necessary to study test frameworks that can be used in an exporter, to create test-scripts for this feature.

Appendix A

Companies

We have interviewed project managers in five companies in Trondheim and Oslo. The questions asked in the interviews are presented in section A.1, and the answers and main opinions from the interviews are summarized for each company.

A.1 Questions

These questions mainly concerns projects that uses Test-Driven Development (TDD), and develops web-applications that can be tested with Watir.

These are the general questions asked about such a project in the company:

How is testing performed in the project?

Is TDD used in the project?
Why/Why not?

Do you user test-script to run automated tests in the project?

Do you use a tool for writing such automated tests?
Why/Why not?

How are acceptance tests performed by customers?

- Are automated test-script created i.e. in Watir, etc., used?
- When in the development process are acceptance tests created?
- Who writes the acceptance tests? The developers or the customsers?

A.2 Bekk

Bekk[1] is a Norwegian consultant-company that delivers development- and consulting-services within three areas: Technology, management and design. Company Details:

Representative: Jan Thoresen

Location: Oslo

Employees: ca 130

In a common web-application project in Bekk, the project team tries to run test-driven development (TDD) to the ability that they find is best. They do not run it as "in theory", because it is a distance between best in theory and what is possible in practice. When using TDD, an ambition for them is to write acceptance tests and acceptance criteria before starting writing code and to write unit tests before writing the correlating code. A web-project with TDD, usually have short development iterations. In Bekk these iterations are based on user stories, proposed by developers based on a requirements specification. For each iteration the customer decides which user story(ies) should be implemented. Then acceptance tests are written to cover the user story(ies), and when the system meats the acceptance criteria the customer approves that iteration. Since a customer only approves some of the functionality in the system in each iteration, regression testing is very important. The developers are constantly adding user stories to the system, and every part needs to function together as a whole system. Therefore is regression testing performed after each new functionality is added.

Developers in Bekk run acceptance tests before delivering an iteration to the customer. These acceptance tests are automatic and are written in a testing tool called Selenium. It is a wish that the customer also should use Selenium to write acceptance test, but it has not been time enough to teach them the tool, yet. Therefore is new functionality after one iteration, demonstrated to the customer. When the whole system has been implemented the customer runs a formal acceptance test. This acceptance test is based on how a user can work with the system, and is written in text in a word document. The formal acceptance test is performed manually by the customer, which goes through the written text and performs the tasks on the document.

A.3 Proxycom

Proxycom[2] is a IT-consulting company that sells software- services and solutions to private and public sector. They use several platforms, technologies,

[1] http://www.Bekk.no/
[2] http://www.proxycom.no/

and programming lanuages, such as Windows, Linux, IBM / CICS, Web, PDA, SMS, ASP, J2EE, Microsoft .NET, C, C++, C#, VB, Java, Cobol, Fortran, Prolog og Lisp. Most of their software development projects are web-based.
Company Details:

Representative: Trond Johansen

Location: Trondheim

Employees: 10

Testing in a common project in Proxycom, includes unit testing of modules, integration testing, system testing against the Requirement specification, user evaluation, and acceptance/approval tests. They have been trying to find a tool for creating automated system tests. This because automated tests can be run and test the whole system "at night", when no one is working on it. In addition, automated tests can easily be run to discover errors in the whole system resulting from changes in a module.

Project teams in Proxycom had evaluated several tools. Among these was Rational Robots evaluated as the best, and it was tested on a system development project. The Rational Robot tool has a capture-replay function which they used to create tests. The tool also enabled the user to program test scripts, but this was not used. They thought, however, that it was not efficient to use the capture-replay tool to create tests. They highlighted that a disadvantage with the tool was that it was sensitive for changes in names and changes in where elements were placed on a web-page. In addition, it put a restriction on the test-data database to contain the same test-data each time the test was run.

Johansen stated that the customer is responsible for the acceptance tests or the approval tests as he called it. The customer however, sometimes used the system tests and the test data used for running tests when the system was developed. The customers of Proxycom usually performed testing manually by going through the written test specifications.

A.4 Kantega

Kantega[3] is a Norwegian IT-consulting company with competence within system development, integration and management. Their employees design and develop software systems within banking and financial-, industrial-, trade- and public sector.

Company Details:

[3] http://www.kantega.no/

Representative: Skule Johansen

Location: Oslo and Trondheim

Employees: 55

Software development projects in the Kantega, starts with a Requirements specification, and Use Cases based on these requirements. To test the application, the company performs unit tests, walk-throughs of code, system tests, technical tests, regression tests and acceptance tests. These tests are usually written after the source code is finished. However, test-cases can be written in the start of projects, but they usually have to be changed because of changes in the applications. Kantega does not use Test Driven Development (TDD), because it is difficult to integrate into the company.

Developers in Kantega write system tests to test the systems normal workpath as well as, exceptions from that path. The system tests are performed by the developers since they concern testing the whole system, including the exceptions that customers are not interested in. The acceptance tests are performed with the customers, as well as with members of the project team. The acceptance tests include functionality tests and non-functionality tests of the system as a whole. Most of these tests are performed manually to cover more of the application than if performed automatically. In addition, in the acceptance tests session, the customer goes through the test reports from other tests such as system tests.

Some acceptance tests that test the functionality of the system are automatically performed. If the company uses automatically tests, there is also a written description of the tests. This way, it is possible for others to verify and approve the tests. The company has tried different tools to write automatically runned test, such as LoadRunner, QuickTestPro and Open STA. LoadRunner was used to automate functionality tests. QuickTestPro was used less often and it tested that web-pages appear as expected. Open STA was easy to learn, and provided good test-results that gave information about testdata-errors or system-errors. The project manager in Kantega thinks that automatic tests offers better testing than when a person perform testing by clicking on in the browser. However, they are not used in the company, because they require more maintenance and therefore become more expensive than manually testing. An experience in the company is that the interest in tools for writing automated tests lies with only one man. He learns how to use the tool, but no one is there to acquire his knowledge.

A.5 Fundator

Fundator[4] is a consulting company in Trondheim, that was formerly a part of EDB Business Consulting in Trondheim. They deliver IT-solutions and carries out consultant-services within the areas of system development and system integration.

Company Details:

Representative: Knut Bliksås

Location: Trondheim

Employees: 19

Fundator does not use Test-Driven Development (TDD). Some tests can be written before code, but this is never the main goal because of the lack of knowledge early in a project. However, Knut Bliksås states that it is easier to create tests early in a development project at the browser level, because these can be drawn early in the process with the customer, in comparison to i.e. unit tests. Regression testing is an important activity in the development process in the company, because their experience is that results are seldom correct the first time. If developers have automated regression tests, i.e. acceptance tests, these could run continually during development and verify that there were no errors in the GUI. The company uses tools for unit-testing that builds and runs the tests both automatically and manually. The tools often used for unit-testing are JUnit, NUnit and CPPUnit. Automatically tests are not used for testing the GUI on web applications, since they have not found a good enough tool to create such tests. Therefore, much time spent is testing web-pages and functionality by clicking on web-pages.

Acceptance tests are not run automatically. A development team formulates a test document based on the customer requirements, which contains steps and expected results of the tests. The document is approved by the customer. If the system is large, developers help the customer when executing the acceptance tests. However, usually with most systems it is the customer that executes the acceptance test. The customer often wants to test the system by clicking on the pages, which gives him a feeling of how easy the system is to use. For this reason, the acceptance tests are often performed manually by the customer. In Fundator, automatically runned test are not created because it takes to long to write them. However, they find that an advantage with automated tests is they can be run often. The code is never perfect and, hence, running more tests can result in finding more errors. In addition, frequent changes in the GUI and functionality are reasons to run automatically tests. It is not obvious that a system works after changing

[4] http://www.fundator.no/

the source code. Running an automated test is, therefore, more safe than relying on that the code works.

A.6 Abeo

Abeo[5] is specialized in deliveries within the areas of service-oriented architecture, identity-handling and trading-solutions.

Representative: Odd Martin Solem

Location: Trondheim and Oslo

Employees: ca 85

The company only uses Test Driven Development (TDD) in some of its projects, depending on who is project leader, project architect and which organization they work for. Solem uses TDD on the project where he currently is the project manager. However, TDD is only used on unit and integration tests. The project has focus on running unit tests, integration tests, system tests and smoke tests[6] while they are developing. They use NUnit to run unit tests on the source code. For system tests, they use a tool called TargetProcess. Their goal is to run these tests without human interaction. In this company, they believe that automated tests are more useful for developers when verifying functionality than for customers.

In Abeo's projects, the acceptance tests are often written after the system is delivered. The tests are not written by the customers, but by the project-team. It is desirable that customers could write the acceptance test, but it is a far step before that is possible. Customers relate to what they see in a GUI, but most test-tools does not have any relationship to a GUI. The acceptance tests are performed by a customer and the company only offers support for the tests. This organization is based on documents when testing, therefore the acceptance tests are run manually.

[5] http://www.abeo.no/

[6] Smoke test are run to tests the system as a whole and its functionality, after it has been deployed onto virtual servers.

Appendix B

Documentation

This documentation is taken from the master thesis [30], and describes how to install AutAT in Eclipse and how to create a start point of a test.

Chapter 17

User Documentation

This part contains the user documentation for AutAT. First we show a detailed installation procedure, before we describe how to create a test in AutAT. The test we will create is the same as used as an example in Chapter 9, a test for a simple CD database system.

17.1 Installation

The installation procedure presented here is rather detailed. For those familiar with Eclipse the short installation is to install Eclipse, GEF and then AutAT.

First *install the Eclipse Platform* (preferably 3.1 that is used in this manual). Eclipse can be downloaded from *http://www.eclipse.org/downloads/index.php*. Extract the downloaded file. In the eclipse catalog that you just extracted, open the eclipse executable.

GEF is required for running AutAT, as AutAT uses features provided by the GEF plugin. Installing GEF is done through the update manager in Eclipse which found by clicking *Help − > Software Updates − > Find and Install....* Select *Search for new features to install* and click *Next >*. Select *Eclipse.org update site* and click *Finish*. Select the *Eclipse.org update site*. Click *OK* and wait for the Update Manager to complete its search (this might take some time). Expand the *Eclipse update site*, and select *Graphical Editing Framework* after expanding the GEF catalog. Note that the version of GEF should be the same version as the Eclipse Platform. Click *Next >* and accept the terms in the licence agreement. Click *Next >* again and then *Finish*. Wait and click *Install*. When the installation is complete, you will be prompted to restart Eclipse.

The process of *installing AutAT* is similar as installing GEF. It uses an update site, just as GEF. Click *Help − > Software Updates − > Find and Install...* before selecting *Search for new features to install*. Click *New Remote Site...* and type *AutAT* for the name and *http://autat.sourceforge.net/update/* as shown in Figure 17.1. Click *OK*. Select *AutAT* and click *Finish*.

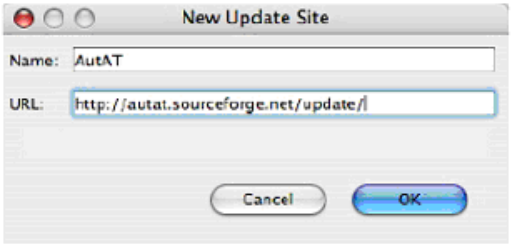

Figure 17.1: Installing a new New Update Site

Select AutAT from the search results as shown in Figure 17.2 and click *Next* >.

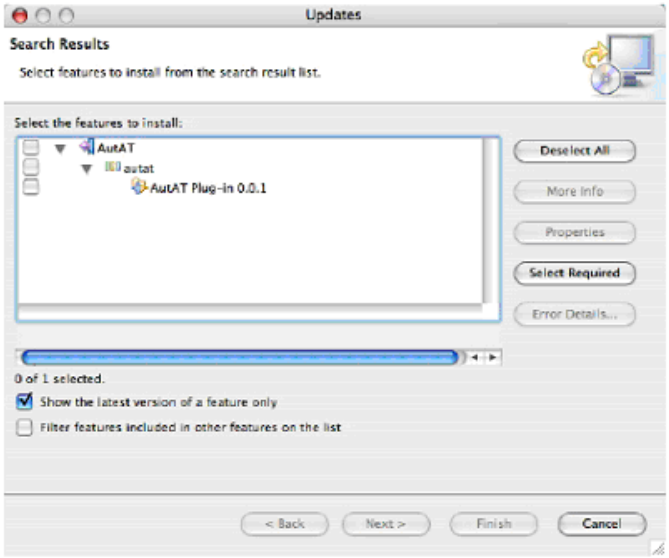

Figure 17.2: Select AutAT

Accept the licence agreement and click *Next* >, *Finish* and *Install*. Restart Eclipse after the installation.

After Eclipse has been restarted, AutAT is ready to be used.

17.2 Usage

Here we will show a step-by-step guide to create a test in AutAT. The test will first check the welcome page of the system, then register a new artist. We assume the user is at least a little experienced with using Eclipse.

The first interaction with AutAT is to open the AutAT *Perspective*. The AutAT perspective is shown in Figure 17.3.

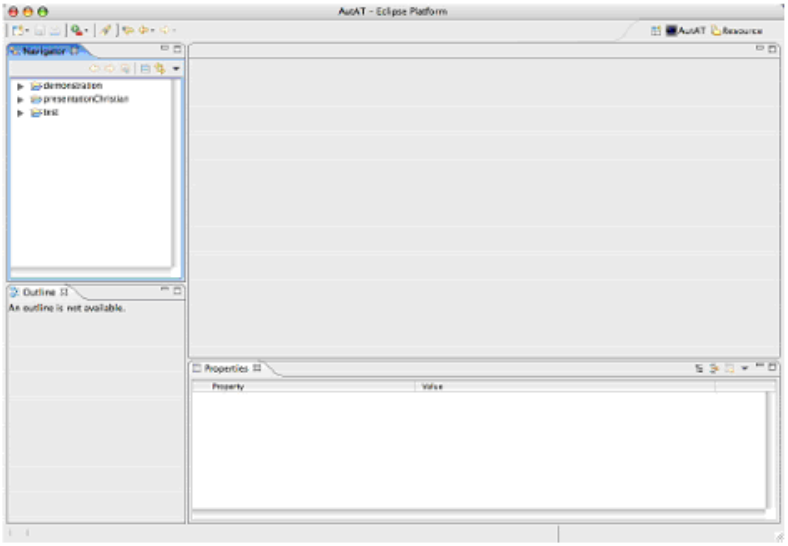

Figure 17.3: The AutAT Perspective

Create a *New AutAT Project* by right-clicking in the navigator view to the left in the perspective, selecting *New Project*. Select *AutAT* − > *New AutAT Project* and click *Next* >. Give the project the name "CDDB", and keep the default location. Click *Next* >. Type "http://cddb.ovstetun.no" as the Base URL as shown in Figure 17.4 and click finish.

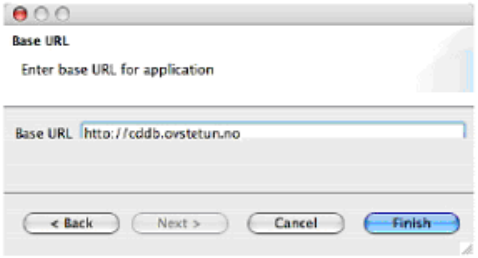

Figure 17.4: Provide a Base URL

A new project will appear in the navigator view, as shown in Figure 17.5. The "tests" folder will contain the user stories and tests for the web application.

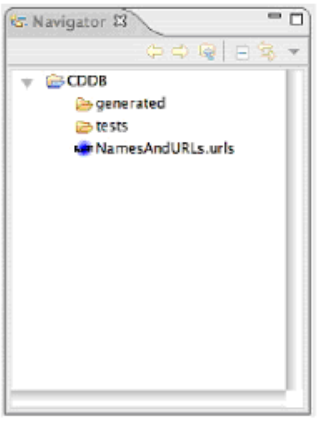

Figure 17.5: The AutAT Navigator

Create a *New Test* by right-clicking on the tests-folder and selecting *New* − > *Other....* Select *AutAT* − > *New AutAT-test* and click *Next* >. Choose a file name, a name for the test and a description as shown in Figure 17.6. Click *Next* >.

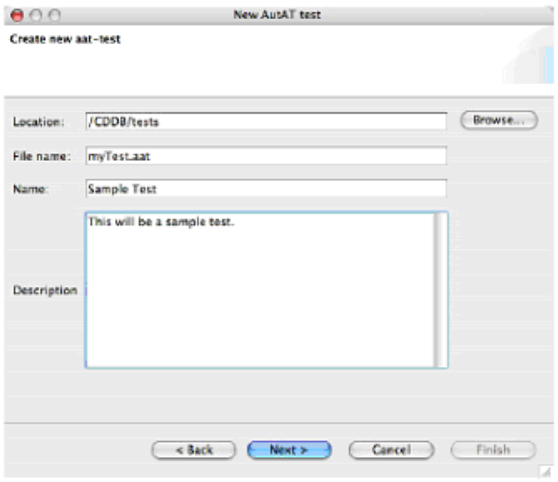

Figure 17.6: Create a new test

Now you must provide a *Start Point* for the test. None have been created, so the list of available points is empty. Click *Add startpoint* to add one. Type the values shown in Figure 17.7 and click OK. Click *Finish* to create the test.

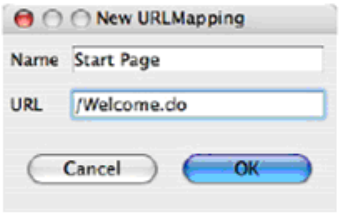

Figure 17.7: Create a new Start Point

Appendix C

Non-Technincal Users

This chapter will describe the information presented to non-technical user under the testing session. The information is written in Norwegian, because the tests were performed in Norwegian.

C.1 Introduksjon til testing

Som hovedoppgave har vi utvidet er verktøy, ved navn AutAT, brukt til automatisk akseptansetesting. Akseptansetesting blir presentert som et case og deretter blir verktøyet presenter ved hjelp av et eksempel.

Case: NTNU

Det Norsk Teknisk Naturvitenskapelig Universitetet (NTNU) ønsker seg en ny nettside da den gamle mangler litt funksjonalitet og er vanskelig å bruke. De lager en kravspesifikasjon der det står blandt annet alt hva en bruker skal kunne gjøre på siden. Siden det finnes så mange smarte it-studenter på NTNU, gir NTNU jobben med å lage siden til to it-studenter. It-studentene er dermed utviklerne av siden, mens NTNU er kunden. Etter at it-studenen har laget web-siden og testet at funksjonaliteten virker, blir siden levert til kunden (NTNU). Det er nå kunden sitt ansvar å teste at en bruker faktisk kan gjøre alt de definerte i kravspesifikasjonen at en bruker skulle kunne gjøre på web-siden. En slik test kalles en akseptanse test. Siden kunden er den som er ansvarlig for akseptanse testen, er det viktig at kunden kan lage testen.

Det er i hovedsak to måter kunden kan teste siden på:

1: Han kan skrive ned hva en bruker skal kunne se og gjøre på siden til enhver tid. I tillegg spesifisere hvilke knapper, linker, felter, tekster, osv., som må være til stede og trykkes på/fylles ut for å teste at brukeren faktisk skal kunne gjøre det som er ment. Deretter kan kunden manuelt gå igjennom siden og teste at den ser ut slik som de ønsker.

2: Kunden kan lage, ved å programmere, et test script som gjør akkurat de samme handlingene som han ville ha gjort manuelt, og til enhver tid sjekke at han ser det han skal se på siden. Dette kalles en automatisk test og er mye enklere dersom testen må utføres flere ganger.

Ideen med AutAT er at at kunden skal kunne lage automatiske test script ved hjelp av bokser og piler. Dermed trenger ikke kunden ha programmeringskunnskap for å lage automatiske tester.

AutAT
AutAT er et verktøy som kan brukes til å modellere dynamiske internettsider, og endringer i internettsidene som et resultat av brukerhandlinger. Innholdet i en dynamisk internettside kan endres, uten at en bruker må laste inn hele siden på nytt. Dermed når for eksempel en bruker skriver inn en bokstav i et tekstfelt, kan en nedtrekksboks med forslag til nettsider dukke opp på siden uten at den lastes inn på nytt. Andre brukerhandlinger som kan føre til en dynamisk endring på siden er: klikke på en link, dra et bilde fra en plass på siden til en annen plass. På grunn av dette dynamiske innholdet på en nettside som kan forandres uten at hele siden oppdateres, holder det ikke å modellere nettsider. Isteden kan man snakke om ”tilstander”, hvor en tilstand er et ”bilde” av en nettside i et bestemt tidspunkt.

Google Suggest er et eksempel på en dynamisk side, som kan vises som et eksempel på hvordan en test kan modelleres i AutAT. For å vise AutAT og eksempler i AutAT uten å måtte åpne programmet har vi laget film av tre utførte eksempler i verktøyet. Disse filmene ligger på en nettside[1] for at de skal være enkle å få tilgang til.

Når du skal lage en test i AutAT, må et startpunkt for testen bestemmes. Dette skrives inn idet testen lages. **Eksempel 1:** Eksempel på hvordan brukergrensesnittet til AutAT ser ut, og hvordan et nytt AutAT prosjekt med en test kan åpnes. I dette eksempelet, er startpunktet addressen til Google Suggest siden. Når startpunktet er gitt, åpnes en tom test i Editoren i Eclipse, kun startpunktet synes. Den første tilstanden som legges til blir koblet til startpunktet, uten at en bruker trenger å lage en kobling mellom disse.

Eksempel 2: Eksempel på modellering av Google Suggest, hvor du starter med å legge til en ny tilstand, med tittelen ”Google Suggest”. Deretter kan du legge til et tekstfelt med teksten ”q” og en knapp som kalles ”btnG”. Når denne tilstanden er laget, kan en ny tilstand legges til i testen. Den neste tilstanden legges til med tittelen, ”Google Suggest New”. I denne tilstanden skal det også være et tekstefelt med teksten ”q” og knappen ”btnG”. Når

[1] http://www.idi.ntnu.no/ stinelil/

den andre tilstanden er lagt til kan forbindelsen mellom den første og den andre tilstanden opprettes ved hjelp av en userActions. I en userActions, elementene fra første tilstanden kan bli valgt i en nedtrekksliste. Når et element er valgt, kan du legge til en ”action” og en ”input” til det elementet. Hvis du ønsker å legge til flere elementer med ”input” i en userActions, er dette mulig ved å trykke på en ”Add New User Input”-knapp. Deretter kommer det opp en ny nedtrekksliste under det første elementet i userActions-lista. I dette eksempelet, velger du ”q” fra nedtrekkslista, og gir det verdiene ”inserText” i ”action” og ”Ruby” i ”input”. Når input-verdiene i koblingen er valgt, går det an å gjøre ferdig ”Google Suggest New”-tilstanden, ved å legge til teksten ”Ruby on rails”. Sluttresultatet og hvordan en test kjøres i AutAT, vises i **Eksempel 3.** Vise andre noen eksempler på dynamiske sider til testpersonen.

C.2 Oppgave som skal utføres

Oppgavene baserer seg på *NTNU*-caset beskrevet ovenfor. Figur C.1 viser hvilke skjermbilder som ønskes og hva som ønskes på de, fra kunden. Skjermbildene er nummerert for å vise rekkefølgen deres, gitt at en bruker gjør handlinger på siden. Oppgaven er å lage en test i AutAT som tester skjermbildene. Det skal lages tilstandene til de forskjellige skjermbildene og userActions mellom tilstandene. Oppgaven beskriver en mulig vei å gå for testen, men det finnes også andre veier. Det skal modellere minst en alternativ vei til veien vist i figur C.1.

C.3 Spørsmål om AutAT

- Etter introduksjonen til AutAT, er det enkelt å forstå hva en tilstand er i AutAT?
- Er det enkelt å forstå koblingen mellom tilstandene?
- Hva mener du om denne måten å lage tester for de handlinger du ønsker å kunne utføre på en nettside? (Fordeler, ulemper, muligheter)
- Hva synes du om AutAT? (Fordeler, ulemper, muligheter)
- Hva synes du om hvordan AutAT ser ut (perspektivet)? (Fordeler, ulemper, muligheter)
- Hva synes du om hvordan du kan gjøre endringer i testen i AutAT? (Fordeler, ulemper, muligheter)

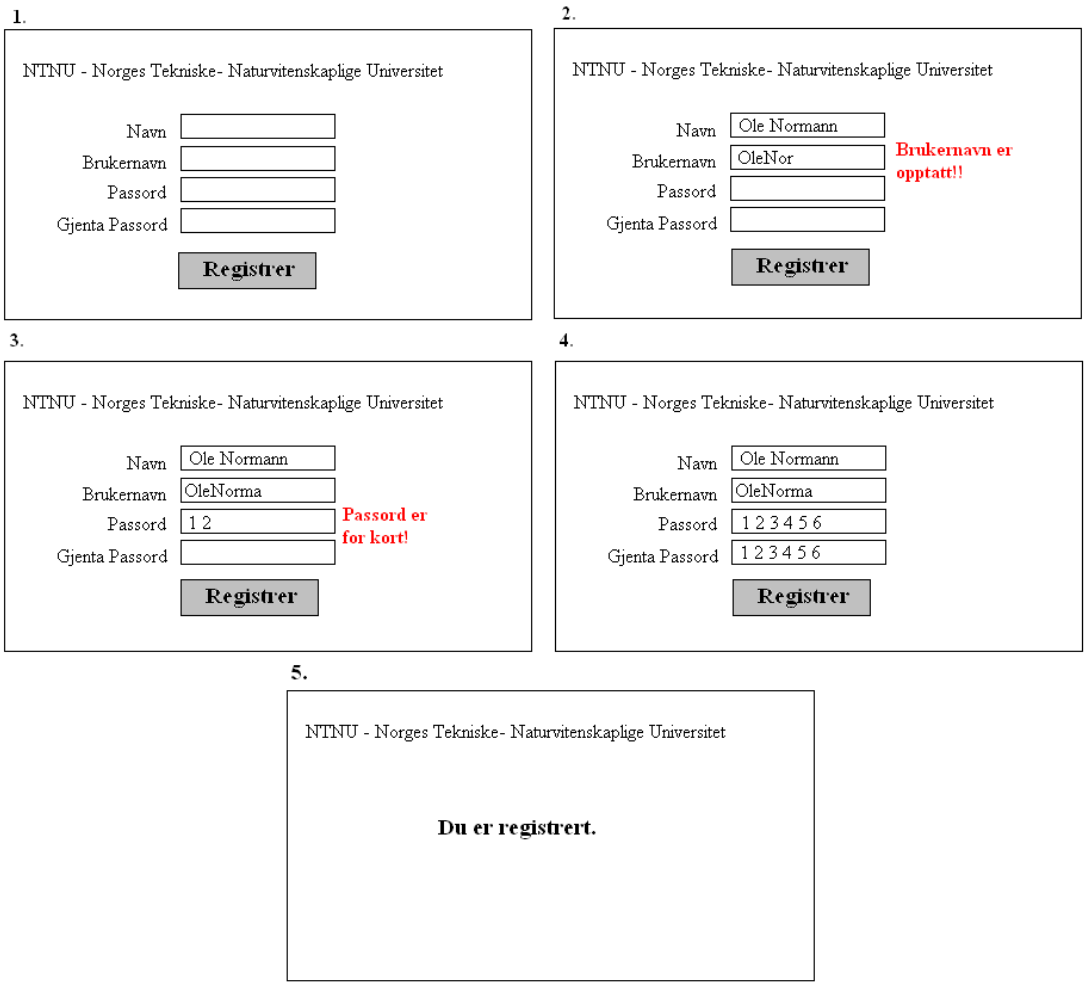

Figure C.1: NTNU Case

Appendix D

Developers and Project Managers

This chapter will describe the information presented to project managers and developers during the test and interviews. These test and interviews were not a formal testing session, but more a conversation about AutAT. For this reason, we have not any actual comments to present.

The information is written in Norwegian, because the tests were performed in Norwegian.

D.1 Introduksjon til AutAT

AutAT er et GUI-basert verktøy som kan brukes til å modellere internettsider og endringer i internettsider som et resultat fra modellerte bruker-input. Ut fra denne modellen kan AutAT generere test-script til eksisterende test-verktøy som f.eks. Watir. Watir-scripts skrives i "Ruby" for å automatisere funksjonell testing av nettsider. AutAT kan ses på som et gui-basert verktøy som ligger som et lag på toppen av eksisterende test-verktøy for å unngå at brukeren må ha programmeringskunnskap.

AutAT ble opprinnelig laget av Stein Kåre Skytteren og Trond øvstetun i deres hovedoppgave våren 2005, for å teste statiske nettsider. Vi har modifisert deres versjon av AutAT til å kunne modellere dynamiske nettsider som inkluderer Ajax (Asynchronouse JavaScript and XML) og JavaScript. Ajax er ikke en ny teknologi, men en ny måte å bruke eksisterende teknologi som XML, Cascading Style Sheet (CSS), Document Obejct Model (DOM) og XMLHttpRequest object.

Ajax er en måte å endre en nettside på uten å laste inn hele siden på nytt. En Ajax-motor, laget i JavaScript, kommuniserer med serveren og henter endringer på siden samtidig som brukeren kan fortsette å kommunisere med

nettsiden. Når nettleseren mottar svaret fra serveren blir kun de nye elementene lastet inn i nettsiden. På grunn av dette holder det ikke å modellere nettsider. Isteden, kan man snakke om "tilstander", hvor en tilstand er et "snapshot" av en nettside i et bestemt tidspunkt. En brukerhandling som fører til at en tilstand endres må også kunne modelleres.

For å vise AutAT og eksempler i AutAT uten å måtte åpne programmet har vi laget film av tre utførte eksempler i verktøyet. Disse filmene ligger på en nettside[1] for at de skal være enkle å få tilgang til.

Eksempel 1: Eksempel på hvordan brukergrensesnittet til AutAT ser ut, og hvordan et nytt AutAT prosjekt med en test kan åpnes.

D.2 Eksempel på modellering av Google Suggest

Google Suggest henter inn passende forslag til søkeord etterhvert som brukeren fyller inn bokstaver i søkefeltet.

Eksempel 2: Eksempel på modellering av Google Suggest.

P.S. Testen feiler på å tittelen "Google" fordi den er lagt inn på siden som et bilde og ikke som tekst.

For å lage watir-tester høyreklikker man på testen og velger "AutAT -> Create Watir Tests". Denne kjører ved å høyreklikke på Ruby-fila og velge "Open with -> System Editor" med Watir innstallert på maskinen.

Eksempel 3: Eksempel på hvordan man lager og kjører Watir tester.

D.3 Spørsmål brukervennlighet og bruk av AutAT stilt til utviklere:

- Er det enkelt å forstå hva en tilstand er i AutAT?
- Er det enkelt å forstå hva en "UserActions"-kobling er?
- Hva mener du om denne måten å lage test-skript for nettsider? (Hva er bra med denne måten? Hva kunne vært bedre?)
- Ville du brukt AutAT? (Hvorfor/hvorfor ikke?)

D.4 Spørsmål om nytteverdien av AutAT stilt til prosjektledere:

- Hva mener du er bra med AutAT?
- Ser konseptene i AutAT ut som de setter tekniske/logiske begrensinger i applikasjonen?

[1] http://www.idi.ntnu.no/ stinelil/

- Tror du at AutAT kunne bli brukt i ditt prosjekt? (Hvorfor/hvorfor ikke?)
- Hvordan mener du at AutAT kan bli endret slik at det blir mer nyttig verktøy?
- Tror du at AutAT kan brukes til å skrive akseptansetester som kan brukes som en del av en kravspesifikasjon? (Hvorfor/hvorfor ikke?)

Appendix E

Feedback

This chapter present the collected feedback during the testing session of persons with non-technical background.

E.1 After the introduction to AutAT, was it easy to understand the state concept in AutAT?

- It was difficult to learn by reading about the State concept.
- I needed the example, an explanation and ask questions, before I understood the state concept.
- It was not simple to understand the State-concept, because I mean that you need an object to have a state.
- States were difficult, user actions were easier.
- The state term was not something tangible and hard to understand.
- I understood the State-concept after reading the introduction.

E.2 Is it easy to understand the connection between the states?

- I liked that it was possible to write the text in as input.
- It was hard to understand that a user could do several action in one UserActions.
- The connection was easy to understand because it included actions I perform on a real web-page.
- I understood the concept with the UserActions-transition fairly fast.
- Actions you can do on a web-page in a transition, easy.
- It was clever to be able to model several actions in one UserActions-transition.

E.3 What do you think about this way of creating test for the action you wish to have on a web-page?

- I would have liked a clear picture of the of the web-page.
- I had a better view of functionality the resulting page.
- The test should be more related to an actual web-page in the tool.
- I missed word that I had heard before.
- I would not have used AutAT to create test for web-pages on my own.

E.4 What do you think about AutAT?

- Aspects were difficult to understand.
- It was easy to learn.
- I wanted to see the actual page, in addition to the test.
- It is not hard to use, even if I have only knowledge about using Word and Excel.
- A test was easy to follow and it was easy to get an overview of the test.

E.5 What do you think about how AutAT looks?

- The palette was clear, and it was easy find what I was looking for.
- The information about the elements on the palette were inadequate, for me to understand how to model tests in AutAT.
- I did not understand some of the words used in AutAT.

E.6 What do you think about the editing possibilities in AutAT?

- It was confusing to be able to edit at different places.
- What I missed to edit in the test, I could edit in the properties window.
- I created an UserActions-transition wrong, and I had to delete it and create a new one to do changes.
- I would prefer to change the UserActions-transition directly in the Editor.
- I did not use the Property Editor.

Appendix F

XML-Schema for the Watir-Exporter

Listing F.1: XML-Schema for the Watir exporter

```
<?xml version='1.0'?>
<xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'
    targetNamespace='http://autat.sourceforge.net' xmlns='
    http://autat.sourceforge.net' elementFormDefault='
    qualified'>

<!--type for the mapping elements-->
<xs:complexType name='mappingType'>
<xs:attribute name='id' type='xs:string' use='required'/>
<xs:attribute name='name' type='xs:string' use='required'
    />
<xs:attribute name='url' type='xs:string' use='required'/
    >
</xs:complexType>

<!--type for the collection of mapping elements-->
<xs:element name='urlMappings'>
<xs:complexType>
<xs:sequence>
<xs:element name='mapping' type='mappingType' minOccurs='
    0' maxOccurs='unbounded'/>
</xs:sequence>
</xs:complexType>
</xs:element>


<!--type for the startPoint-->
<xs:complexType name='startPointType'>
<xs:attribute name='id' type='xs:string' use='required'/>
</xs:complexType>

<!--type for connectionPoint-->
<xs:complexType name='connectionPointType'>
```

```
28 <xs:sequence>
29 <xs:element name='startPoint' type='startPointType'
      minOccurs='0' maxOccurs='1'/>
30 </xs:sequence>
31 </xs:complexType>
32
33 <!--type for link element:linkType-->
34 <xs:complexType name='linkType'>
35 <xs:simpleContent>
36 <xs:extension base='xs:string'>
37 <xs:attribute name='not' type='xs:boolean'/>
38 </xs:extension>
39 </xs:simpleContent>
40 </xs:complexType>
41
42 <!--type for text element:textType-->
43 <xs:complexType name='textType'>
44 <xs:simpleContent>
45 <xs:extension base='xs:string'>
46 <xs:attribute name='not' type='xs:boolean'/>
47 </xs:extension>
48 </xs:simpleContent>
49 </xs:complexType>
50
51 <!--type for form types:formFieldType-->
52 <xs:complexType name='formFieldType'>
53 <xs:attribute name='id' type='xs:string'/>
54 <xs:attribute name='name' type='xs:string'/>
55 <xs:attribute name='type' type='xs:string'/>
56 </xs:complexType>
57
58
59 <!--type for a list of elements:elementsType-->
60 <xs:complexType name='elementsType'>
61 <xs:choice minOccurs='0' maxOccurs='unbounded'>
62 <xs:element name='link' type='linkType'/>
63 <xs:element name='text' type='textType'/>
64 <xs:element name='formElement' type='formElementType'/>
65 </xs:choice>
66 </xs:complexType>
67
68 <!--type for a single page:pageType-->
69 <xs:complexType name='pageType'>
70 <xs:sequence>
71 <xs:element name='title' type='xs:string'/>
72 <xs:element name='elements' type='elementsType'/>
73 </xs:sequence>
74 <xs:attribute name='id' type='xs:string' use='required'/>
75 <xs:attribute name='xPos' type='xs:integer' use='required
      '/>
76 <xs:attribute name='yPos' type='xs:integer' use='required
      '/>
77 </xs:complexType>
```

```
78
79 <!--type for a list of pages:pagesType-->
80 <xs:complexType name='pagesType'>
81 <xs:sequence>
82 <xs:element name='page' type='pageType' maxOccurs='
       unbounded'/>
83 </xs:sequence>
84 </xs:complexType>
85
86 <!--type for a singleaspect:aspectType-->
87 <xs:complexType name='aspectType'>
88 <xs:sequence>
89 <xs:element name='title' type='xs:string'/>
90 <xs:element name='elements' type='elementsType'/>
91 </xs:sequence>
92 <xs:attribute name='id' type='xs:string' use='required'/>
93 <xs:attribute name='xPos' type='xs:integer' use='required
       '/>
94 <xs:attribute name='yPos' type='xs:integer' use='required
       '/>
95 </xs:complexType>
96
97 <!--type for a list of aspects:aspectsType-->
98 <xs:complexType name='aspectsType'>
99 <xs:sequence>
100 <xs:element name='aspect' type='aspectType' minOccurs='0'
        maxOccurs='unbounded'/>
101 </xs:sequence>
102 </xs:complexType>
103
104 <!--type for simpletransitions:simpleTransitionType-->
105 <xs:complexType name='simpleTransitionType'>
106 <xs:attribute name='from' type='xs:string' use='required'
       />
107 <xs:attribute name='to' type='xs:string' use='required'/>
108 </xs:complexType>
109
110 <!--type for user actions values-->
111 <xs:complexType name='userActionValue'>
112 <xs:attribute name='id' type='xs:string' use='required'/>
113 <xs:attribute name='action' type='xs:string' use='
       required'/>
114 <xs:attribute name='input' type='xs:string'/>
115 </xs:complexType>
116
117 <!--type for userActions transition:
       UserActionsTransitionType-->
118 <xs:complexType name='userActionsTransitionType'>
119 <xs:choice eminOccurs='0' maxOccurs='unbounded'>
120 <xs:element name='userAction' type='userActionValue'/>
121 </xs:choice>
122 <xs:attribute name='from' type='xs:string' use='required'
       />
```

```
123 <xs:attribute name='to' type='xs:string' use='required'/>
124 </xs:complexType>
125
126 <!--type for aspect transition:aspectTransitionType-->
127 <xs:complexType name='aspectTransitionType'>
128 <xs:attribute name='from' type='xs:string' use='required'
    />
129 <xs:attribute name='to' type='xs:string' use='required'/>
130 </xs:complexType>
131
132 <!--type for a list of transitions-->
133 <xs:complexType name='transitionsType'>
134 <xs:choice minOccurs='0' maxOccurs='unbounded'>
135 <xs:element name='simpleTransition' type='
    simpleTransitionType'/>
136 <xs:element name='userActionsTransition' type='
    userActionsTransitionType'/>
137 <xs:element name='aspectTransition' type='
    aspectTransitionType'/>
138 </xs:choice>
139 </xs:complexType>
140
141
142 <!--the test type,base element in the test documents-->
143 <xs:element name='test'>
144 <xs:complexType>
145 <xs:sequence>
146 <xs:element name='description' type='xs:string'/>
147 <xs:element name='connectionPoint' type='
    connectionPointType'/>
148 <xs:element name='pages' type='pagesType'/>
149 <xs:element name='aspects' type='aspectsType'/>
150 <xs:element name='transitions' type='transitionsType'/>
151 </xs:sequence>
152
153 <xs:attribute name='id' type='xs:string' use='required'/>
154 <xs:attribute name='name' type='xs:string'/>
155 </xs:complexType>
156 </xs:element>
157
158 </xs:schema>
```

Bibliography

[1] Canoo Engineering AG. Canoo webtes distribution site. `http://webtest.canoo.com`. Visited:31.05.2006.

[2] Scott W. Ambler. Introduction to test drien development (tdd). `http://www.agiledata.org/essays/tdd.html`. Visited:19.04.2006.

[3] Anneliese A. Andrewa, Jeff Offoutt, and Roger T.Alexander. Testing web applications by modeling with fsms. `http://ise.gmu.edu/~ofut/rsrch/papers/webtest.pdf`.

[4] Joseph Bergin. Using htmlfixture. `http://fitnesse.org/FitNesse.FitNesse.HtmlFixture`. Visited:04.06.2006.

[5] Dave Crane, Eric Pascarello, and Darren James. *AJAX IN ACTION*. Manning Publications Co., 2006.

[6] Creotec. electronic business terms and definitions (glossary). `http://www.creotec.com/index.php?page=e-business_terms`. Visited:11.06.2006.

[7] Cunningham and Cunningham. Fit: Framework for integrated test. `http://fit.c2.com/`. Visited:28.04.2006.

[8] Darkforge. Jsunit and ajax don't mix! `http://books.slashdot.org/article.pl?sid=06/03/01/1356241&from=rss`. Visited:02.06.2006.

[9] DevGuru. Javascript. `http://www.devguru.com/Technologies/ecmascript/quickref/javascript_intro.html`. Visited:09.06.2006.

[10] Ruby Forge. Support for xpath in watir. `http://rubyforge.org/cgi-bin/viewcvs.cgi/watir/doc/?root=wtr`. 20.05.2006.

[11] Juliana Freire. Veriweb: Automatically testing dynamic web sites. `http://www2002.org/CDROM/alternate/654/`. Visited:18.04.2006.

[12] Jesse James Garrett. Ajax: A new approach to web applications. `http://www.adaptivepath.com/publications/essays/archives/000385.php`. Visited:14.02.2006.

[13] Grig Gheorghiu. Ajax testing with selenium using waitforcondition. `http://agiletesting.blogspot.com/2006/03/ajax-testing-with-selenium-using_21.html`. Visited:31.05.2006.

[14] Grig Gheorghiu. Web app testing with python part 2: Selenium and twisted. `http://agiletesting.blogspot.com/2005/03/web-app-testing-with-python-part-2.html`. Visited:31.05.2006.

[15] Grig Gheorghiu. A look at selenium. *BETTER SOFTWARE*, October 2005.

[16] Unmesh Gundecha. Web application testing in ruby. `www2002.org/presentations/freire.pdf`. Visited:18.04.2006.

[17] Mary Jean Harrold. Testing: a roadmap. In *ICSE - Future of SE Track*, pages 61–72, 2000.

[18] Christian Hellsten. Automate acceptance tests with selenium. `http://www-128.ibm.com/developerworks/xml/library/wa-selenium-ajax/index.html`. Visited:31.05.2006.

[19] HTMLDog. Html tags. `http://www.htmldog.com/reference/htmltags/`. Visited:23.05.2006.

[20] A. Russell Jones. Is dhtml dead? `http://www.devx.com/DevX/Article/16377`, December 2001.

[21] Alexander Kellett. Automated website testing with java - httpunit / jwebunit. `http://www.lunatech.com/archives/2005/11/30/automated-website-testing-with-java-httpunit-jwebunit`, November 2005. Visited:12.06.2006.

[22] Jonathan Kohl and Paul Rogers. Watir works. `http://www.kohl.ca/articles/watir_works.pdf`. Visited:18.04.2006.

[23] Sun Developers Network. Applets. `http://java.sun.com/applets/`. Visited:07.03.2006.

[24] Ruby on Rails. Seleniumintegration. `http://wiki.rubyonrails.com/rails/pages/SeleniumIntegration`. Visited:18.04.2006.

[25] Open. Selenium - functional testing framework for web applications. `http://www.openqa.org/selenium/index.html`. Visited:31.05.2006.

[26] Bret Pettichord. element selection from pages/naive(?) watir questions. `http://rubyforge.org/pipermail/wtr-general/2005-May/001927.html`. Visited:13.06.2006.

[27] Extreme Programming. Extreme programming: A gentle introduction. `http://www.extremeprogramming.org/`. Visited:08.03.2006.

[28] Martin R. and Martin M. Fitnesse web site. `http://www.fitnesse.org`. Visited:18.04.2006.

[29] Steve Skidmore. The v model. *Professional Scheme Paper 2.1*, pages 48–49, January 2006.

[30] Stein Kåre Skytteren and Trond Marius Øvstetun. Autat - an eclipse plugin for automatic acceptance testing of web applications. Master's thesis, NTNU, 2005.

[31] SoruceForge. Jsunit. http://www.jsunit.net/. Visited:18.04.2006.

[32] SourceForge. What is jwebunit. `http://jwebunit.sourceforge.net/`. Visited:18.04.2006.

[33] SourceForge.net. xunit - unit testing framework. `http://sourceforge.net/projects/xunit`. Visited:13.06.2006.

[34] Ibma Supreme Tmunotein. Client-side and server-side javascript. `http://www.devarticles.com/c/a/JavaScript/Client-side-and-Server-side-JavaScript/`. Visited:13.02.2006.

[35] Hans Van Vliet. *Software Engineering*. John Wiley & Sons, Baffins Lane, Chichester, second edition edition, 2002.

[36] W3schools. Full web building tutorials - all free. `http://www.w3schools.com/`. Visited:13.03.2006.

[37] WebRef. Core javascript guide. `http://www.webreference.com/javascript/reference/core/index.html`. Visited:13.02.2006.

[38] Webref. Dynamic web sites with xml, xslt and jsp. `http://www.webreference.com/xml/column37/`. Visited:11.06.2006.

[39] Aaron Weiss and Scott J. Walter. The complete idiot's guide to javascript - online version. `http://search.netscape.com/ns/boomframe.jsp?query=javascipt&page=1&offset=0&result_url=redir%3Fsrc%3Dwebsearch%26requestId%3Df52dd01dfa28a0b2%26clickedItemRank%3D2%26userQuery%3Djavascipt%26clickedItemURN%3Dhttp%253A%252F%252Fwww.help4web.net%252Fwebmaster%252FJava%252FNewJS%252FJavaScriptIdiotsGuide%252F%2521start_here.html%26invocationType%3D-%26fromPage%3DNSCPTop%26amp%3BampTest%3D1&remove_url=http%3A%2F%2Fwww.help4web.net%2Fwebmaster%2FJava%2FNewJS%2FJavaScriptIdiotsGuide%2F%21start_here.html`. Visited:14.02.2006.

[40] Martin Fowler with Kendall Scott. *UML Distilled*. Addison-Wesley, second edition, October 2002.

[41] Claes Wohlin, Per Runeson, Martin H´ost, Magnus C. Ohlsson, Bj´orn Regnell, and Anders Wesslén. *Experimentation in Software Engingeering : An introduction*. Kluwer Academic Publishers, 200.