

Abstract

Today, more and more applications are web based. As these systems are getting larger and larger, the need for testing them is increasing. XP and other agile methodologies stress the importance of test driven development and automatically testing at all levels of testing. There exists a few open source automatic testing frameworks for web applications' features. These are, however, rather poor when it comes to usability, efficiency and quality factors.

This project has created a tool for automatic acceptance testing, called AutAT, which aims at being an improvement when compared to the previous tools' testing features. The tool has been empirically tested to verify that it is better when it comes to the parameters usability, efficiency and quality. The results from this test clearly show that AutAT is superior to available state of the art open source tools for acceptance testing.

Keywords: AutAT, Automatic Acceptance Test, Web Application, Feature Testing Tool, Test Driven Development

Preface

This master thesis documents the work done by Stein Kåre Skytteren and Trond Marius Øvstetun from January to July 2005. This thesis is related to the BUCS (BUbusiness-Critical Software) project run by the Department of Computer Science and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU). It is also related to Bekk Consulting AS (BEKK) and focus on open source software through their BEKK's Open Source Software (BOSS) projects.

We would like to thank Professor Tor Stålhane at IDI, NTNU and Trond Arve Wasskog and Christian Schwarz at Bekk Consulting AS for all their guidance and invaluable feedback during our work on this thesis. We really appreciate their time and efforts helping us.

We would also like to thank the participants in our test session: Tore Aabakken, Ola Sætrom, Arne Johan Hestnes, Thor Marius Henriksen, Øyvind Rød, Frode Hauso, Carl-Henrik Wolf Lund and Michael Sars Norum.

Trondheim, July 1, 2005

Stein Kåre Skytteren

Trond Marius Øvstetun

Contents

I	Introduction	1
1	Motivation	3
2	Problem Definition	4
3	Project Context	5
4	Readers Guide	6
5	Project Process	8
6	Research Question and Method	11
6.1	Goal	12
6.2	Questions	13
6.3	Metrics	14
II	Prestudy	25
7	eXtreme Programming – XP	27
7.1	Values	28
7.2	Principles and Practices	29
8	Testing	31
8.1	V-model	31
8.2	Different types of testing	33
8.2.1	Black Box Testing	33
8.2.2	White Box Testing	34
8.2.3	Human Testing	34
8.3	Extreme testing – Test Driven Development	35
8.4	Testing Web Applications	36
8.5	Summary	37
9	State of the Art - Existing Testing Technologies	38
9.1	xUnit	38
9.2	JUnit	39
9.2.1	jWebUnit	40

9.2.2	HTMLUnit	41
9.3	FIT	42
9.3.1	WebFixture	43
9.3.2	HTMLFixture	43
9.4	FitNesse	44
9.5	Selenium	46
9.6	Canoo WebTest	46
9.7	Summary and Comparison	48
10	Technology Platform	50
10.1	Web	50
10.2	Standalone	51
10.3	Eclipse Plugin	51
10.3.1	RCP - Rich Client Platform	52
10.4	Summary - The Choice	52
11	The Eclipse Architecture	53
11.1	The Platform Runtime and Plugin Architecture	54
11.1.1	The Workbench, SWT and JFace	55
11.1.2	Workspaces	56
11.2	GEF	56
11.3	The Team API	57
12	The Graphical Editor Framework	58
12.1	Model-View-Controller	59
12.2	Command	60
12.3	Chain of Responsibility	61
12.4	State	62
12.5	Abstract Factory	62
12.6	Factory Method	63
III	Contribution	65
13	Requirements	67
13.1	Vision	67
13.2	User Stories	68
14	Design	70
14.1	Domain model	70
14.2	Architecture	73
14.2.1	Using the File System	75
14.2.2	AutAT Internals	76
15	Implementation	80
15.1	AutAT Common	80
15.2	AutAT Core	81
15.2.1	XML Schemas	82

15.3	AutAT Exporter	84
15.4	AutAT UI	85
15.5	AutAT Software Metrics	87
16	Testing	90
17	User Documentation	92
17.1	Installation	92
17.2	Usage	93
18	User Testing Session	98
18.1	Background	98
18.2	FitNesse	98
18.3	AutAT	99
18.4	Evaluation	99
IV	Evaluation	101
19	Research Analysis and Goal Attainment	103
19.1	Threats to Validity	103
19.2	Measurements	104
19.3	Answers	108
19.4	Goal Attainment	111
20	Discussion	113
20.1	AutAT Tool	113
20.1.1	Test Framework	113
20.1.2	Eclipse	114
20.1.3	GEF	114
20.1.4	AutAT Itself	114
20.2	Experiment	114
20.3	Personal Experiences	116
21	Conclusion	117
22	Future Work	118
22.1	Future Work with the AutAT Tool	118
22.2	Future Empirical Experiments	119
22.3	Future Work at NTNU and BEKK	119
V	Appendix	121
A	Questionnaire for AutAT testing	123
A.1	Background and Experience	123
A.2	FitNesse	123
A.2.1	Time Usage	123

A.2.2	Statements	124
A.3	AutAT	124
A.3.1	Time Usage	125
A.3.2	Statements	125
A.4	Comparison: FitNesse versus AutAT	126
A.4.1	Statements	126
B	Testing Exercises	128
B.1	Exercise 1: Test the Front Page	128
B.2	Exercise 2: List All Artists	128
B.3	Exercise 3: Register New Artist	129
B.4	Exercise 4: View Info About Artist and Register a New Album	129
B.5	Exercise 5: Repeating Checks	129
C	FitNesse Commands	131
C.1	General commands in FitNesse	131
C.2	Commands for using jWebUnit with FitNesse	131
C.2.1	Checks for text and links	131
C.2.2	Navigation commands	132
C.2.3	Checks for forms	132
C.2.4	Enter values to a form	132
C.3	Example	132
D	Statistical Background	133
D.1	Analysing the Number of Testers	133
D.2	Binary Probability Distribution	134
E	User feedback	135
E.1	M1 - Fit Knowledge	135
E.2	M2 - AutAT Time Usage	136
E.3	M3 - FitNesse Time Usage	136
E.4	Time Analysis	136
E.5	M4 - AutAT's Ease of Learning	138
E.6	M5 - FitNesse's Ease of Learning	139
E.7	M6 - Compared Ease of Learning	139
E.8	Analyzing Ease of Learning	140
E.9	M7 - AutAT's Ease of Use	140
E.10	M8 - FitNesse's Ease of Use	141
E.11	M9 - Compared Ease of Use	141
E.12	Analyzing Ease of Use	141
E.13	M10 - AutAT's Syntax Complexity	142
E.14	M11 - FitNesse's Syntax Complexity	142
E.15	M12 - Compared Syntax Complexity	143
E.16	Analyzing Syntax Complexity	143
E.17	M13 - AutAT's Overview	144
E.18	M14 - FitNesse's Overview	144
E.19	M15 - Compared Overview	145

E.20 Analyzing Overview	145
E.21 M16 - Modifying Tests	146
E.22 Analyzing Modifying Tests	146
E.23 M17 - AutAT Errors	147
E.24 M18 - FitNesse Errors	147
E.25 Error Analysis	147
E.26 M19 - User Feedback	149
E.26.1 FitNesse	149
E.26.2 AutAT	150
E.26.3 Comparison: FitNesse versus AutAT	152

F XML schemas

List of Figures

5.1	Project Process	9
6.1	GQM Method	12
6.2	GQM-Tree Example	12
7.1	Cost of change	27
8.1	V-model of testing	32
9.1	The xUnit Architecture.	39
9.2	FitNesse example using WebFixture	45
11.1	The Eclipse architecture	54
11.2	A sample view of the Eclipse Workbench	55
12.1	GEF Dependencies	58
12.2	An overview of the MVC pattern in GEF.	60
12.3	The most important methods in the GEF Command class.	60
12.4	The AbstractEditPolicy class from GEF	61
12.5	The Tool interface with its most important methods.	62
12.6	The EditPartFactory interface in GEF.	62
12.7	The CreationFactory interface in GEF.	63
14.1	Top-level domain model	71
14.2	The form and form elements.	72
14.3	Transitions	72
14.4	StartPoint	73
14.5	The AutAT plugin and dependencies in Eclipse	74
14.6	Example AutAT project hierarchy	75
14.7	The internal structure of the AutAT plugin	76
14.8	The structure of the UI package	77
14.9	The structure of the GEF package	78
15.1	Central classes in the Common package	81
15.2	A full test shown with the AutAT plugin	86
15.3	The wizard when creating a new test	88
15.4	The start point editor	89
17.1	New Update Site	93

17.2	Select AutAT	93
17.3	The AutAT Perspective	94
17.4	Provide a Base URL	94
17.5	The AutAT Navigator	95
17.6	Create a new test	95
17.7	Create a new Start Point	96
17.8	An empty test	96
17.9	Provide input values to a form	97
17.10	The final test	97
18.1	Test Session	99

List of Tables

6.1	Metric 1 – FIT knowledge	15
6.2	Metric 2 – AutAT time usage	15
6.3	Metric 3 – FitNesse time usage	15
6.4	Metric 4 – AutAT’s ease of learning	16
6.5	Metric 5 – FitNesse’s ease of learning	16
6.6	Metric 6 – Compared ease of learning	17
6.7	Metric 7 – AutAT’s ease of use	17
6.8	Metric 8 – FitNesse’s ease of use	18
6.9	Metric 9 – Compared ease of use	18
6.10	Metric 10 – AutAT’s syntax complexity	19
6.11	Metric 11 – FitNesse’s syntax complexity	19
6.12	Metric 12 – Compared syntax complexity	20
6.13	Metric 13 – AutAT’s overview	20
6.14	Metric 14 – FitNesse’s overview	21
6.15	Metric 15 – Compared overview	21
6.16	Metric 16 – Modifying tests	22
6.17	Metric 17 – AutAT errors	22
6.18	Metric 18 – FitNesse errors	23
6.19	Metric 19 – User feedback	23
6.20	Relationship between Questions and Metrics	24
9.1	Basic FIT example showing division	42
9.2	WebFixture example	43
9.3	HTMLFixture example	44
9.4	Selenium example	47
15.1	Key software metrics	87
D.1	Number of testers analysis	133

List of Listings

9.1	jWebUnit example	40
9.2	HTMLUnit example	41
9.3	Canoo WebTest example	47
15.1	Reading a Test from XML (from TestConverter.java)	82
15.2	Saving a Test to XML (from TestConverter.java)	82
15.3	XML schema for start points	83
15.4	XML schema for tests	84
15.5	The abstract class DirectoryWalker	85
15.6	Implementation of test conversion for WebFixture	85
F.1	XML schema for start points	153
F.2	XML schema for tests	154

Part I

Introduction

This part is the background and foundation for the project. It starts out with a chapter about motivation before it focuses on the problem definition, the project context, the readers guide and the project's process in next chapters. The final chapter is a description of the project's research question and method.

Chapter 1

Motivation

The use of the eXtreme Programming (XP) and other agile methods for software development has now been widely adopted. Most of these methods stress the use of Test Driven Development (TDD). TDD states that the tests for the software system under development are written before the actual software, and as far as possible the tests must be automatic.

The tests to write come in two different categories. The first category is *Acceptance tests*. The Acceptance tests are like a living requirements document for the system, and the customer is responsible for both writing and maintaining these tests. The acceptance tests are tests for the software system as a whole, when all are accepted, the system is considered finished. The second category consist of *Unit and Integration tests*. These tests are the responsibility of the developers, and test smaller parts of the software system and the cooperation between the parts.

Many tools and frameworks exist for creating and running unit and integration tests. With regards to acceptance tests, the collection of tools and frameworks is smaller. Some exist, but are in general difficult for the customers to use, understand and trust. As the goal of XP is to have the customer or end user write, run and maintain the tests this situation is not satisfactory.

Many of the systems developed using XP are web-based systems. Testing of such systems has traditionally been done manually, using hours to input data and check for correct output. Clearly, a method for automating most of these tests can save both money and time for the customer as well as for the developing organization.

Chapter 2

Problem Definition

The aim of this master thesis is to develop a new tool that will serve as proof of concept for creating and maintaining acceptance tests for web-based applications' features. The tool will be called "AutAT – Automatic Acceptance Testing of Web Applications". An important part of this thesis is to explore the domain of testing web applications and address potential improvements in the field of study.

The tool is to be empirically tested to check whether or not it is an improvement with respect to usability, quality and efficiency compared with state of the art tools. The main focus is software developers working with test driven development and open source software as it is an important focus for BEKK. We will use the Goal-Question-Metric method to identify a good set of questions and metrics to support our goal.

This tool as a proof of concept should be a basis for a larger, future tool for developing web acceptance tests and distributed freely as open source software as part of the BEKK Open Source Software (BOSS) site¹.

¹<http://boss.bekk.no>

Chapter 3

Project Context

This project is carried out partly at 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, where it is associated with the BUsiness-Critical Software¹ (BUCS) project. The BUCS project is funded by the Norwegian Research Council. It works closely together with Norwegian IT-industry and do research on how to develop methods to improve support for development, operation and maintenance of business-critical systems/software.

This project is also carried out in cooperation with Bekk Consulting AS (BEKK) which is a leading Norwegian business consulting and technology service company. BOSS² is BEKK's Open Source Software website where they host information about their Open Source projects. BEKK dedicates a lot of their time to create useful open source tools, and BOSS is a way of giving back to the Open Source Software community. Most projects at BEKK create web applications, and they try to use the XP methods. XP requires automated testing, but automatic acceptance testing web applications is difficult as it is poorly supported or has low usability within today's tools. A new and better tool for the purpose will make the process of creating and maintaining tests easier.

¹<http://www.idi.ntnu.no/grupper/su/bucs/>

²<http://boss.bekk.no>

Chapter 4

Readers Guide

This chapter contains a brief overview of the entire report, giving a short description of the issues discussed in each part.

Part I – Introduction

The first part sets the focus for the rest of the document. It contains the motivation, problem definition, context, this readers guide and an overview of the project's process. It ends with an overview of the project's research question and method.

Part II – Prestudy

The prestudy is the background information for this project. The main topics here are eXtreme Programming (XP), testing and state of the art testing technologies. It also evaluates technology platforms for AutAT before it looks at the Eclipse architecture and the Graphical Editing Framework (GEF).

Part III – Contribution

The contribution focuses on the development of AutAT. It starts out with a chapter on requirements, before it goes on with design, implementation and testing. A short user documentation for the system is provided as an introduction to AutAT's use. The final chapter in this part describes the user testing session.

Part IV – Evaluation

This part first addresses the research question and goal attainment which is connected to the research question and method in the first part and the user

testing session described in the final chapter of Part III. This part also discusses this project, before it concludes and presents some suggestions for future work.

Part V – Appendix

The appendix contains the questionnaire for the AutAT test session, a set of testing exercises and FitNesse commands used in the testing session in addition a part on statistical background and user feedback with some analysis. Finally the complete XML schemas used in the AutAT file system are shown.

The CD

The CD that is delivered with this report contains the source code for the AutAT Eclipse plugin. It also contains a folder with JavaDoc created from the source code. For more information, see the `readme.txt` file in the root folder.

Chapter 5

Project Process

The project process has been influenced by eXtreme Programming (XP), even though the graphical overview of the project process shown in Figure 5.1 seems more like a waterfall model. However, the implementation phase will be performed as a series of iterations in accordance with the XP method. The representation of the phases or activities in the figure are not meant to be “correctly” sized and spaced according to their duration. The figure merely shows the activities that have been performed during the project’s lifespan.

The activities in Figure 5.1 are:

- **Start** – The project started out with a set of meetings with the project stakeholders, to set the direction of this project.
- **Problem definition** – Working out the problem definition found in Chapter 2.
- **XP/agile methods** – Getting to know XP and agile methods. An overview of XP is presented in Chapter 7.
- **Testing** – Getting to know the domain of testing better. There is a brief overview of testing in Chapter 8.
- **State of the Art** – A survey looking into and evaluating the most popular open source testing frameworks with an emphasis on acceptance testing web applications. The results are described in Chapter 9.
- **User stories** – This phase looks at the requirements for the application. These are presented as user stories in Chapter 13.
- **Domain analysis** – The domain analysis elaborates on what tests for web applications consist of, and how they can best be represented to the users.
- **User test design** – Doing the basis for how we would like to empirically test the system, presented in Chapter 6.

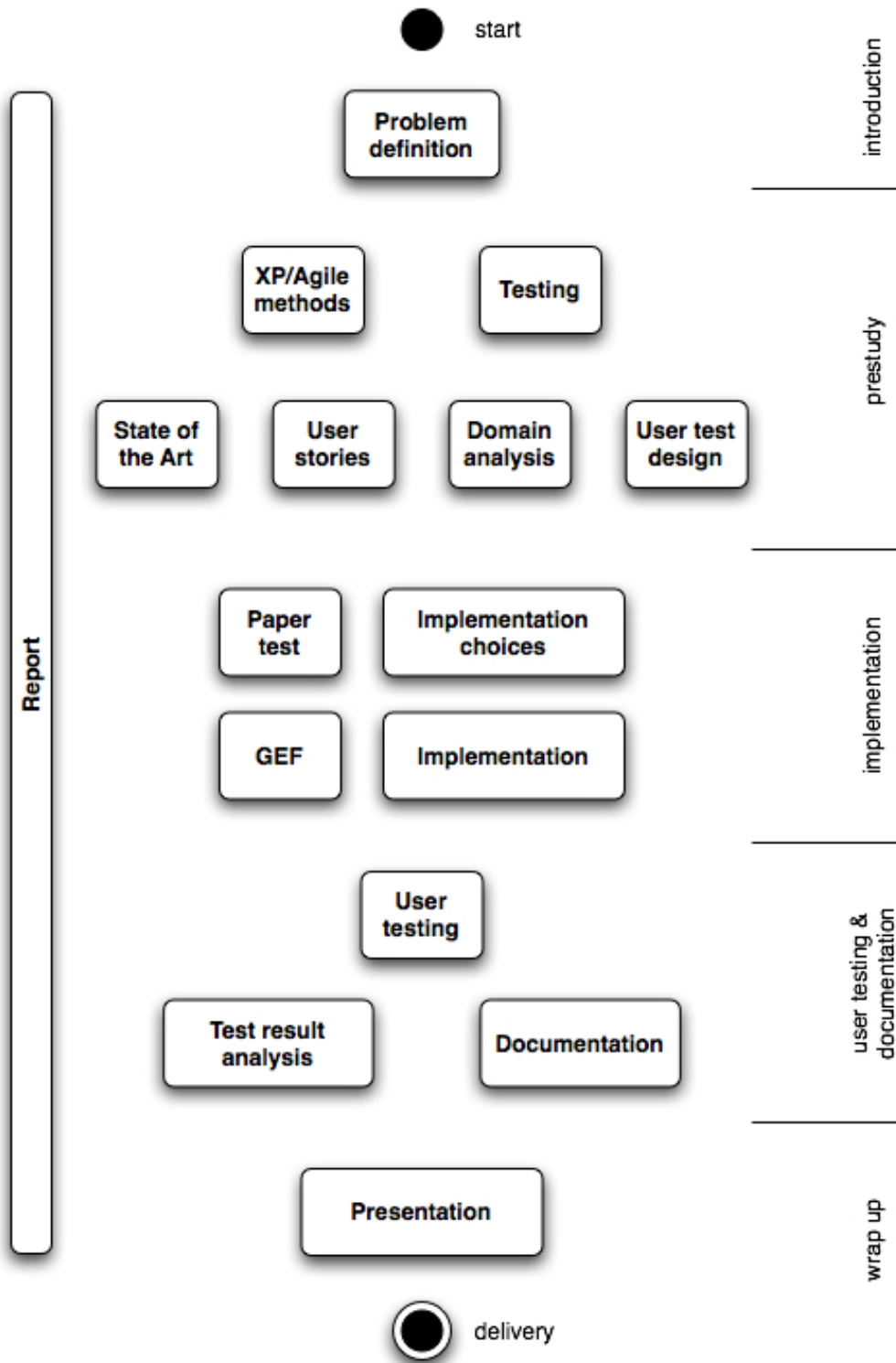


Figure 5.1: The Project Process

- **Paper test** – Paper tests are useful to early communicate and discuss design choices and how the finished system can be. A session with paper prototypes showing the results of the domain analysis was held before the actual implementation started.
- **Implementation choices** – The application can be implemented using several technologies. This phase tries to look at possibilities and finally come up with a design to be implemented.
- **GEF** – The Graphical Editor Framework (GEF) is a complex framework. Getting to understand its workings and how it integrates into Eclipse was quite a process. This activity took place in parallel with the implementation of the application, in a “learning by doing”-fashion. An overview of Eclipse and GEF is presented in Chapter 11 and Chapter 12 respectively.
- **Implementation** – This phase is the actual implementation phase. This phase is done in “cooperation” with many of the nearby phases. It also involves continuous testing of the system.
- **User testing** – This is the data collection phase for the empirical study which is designed in the user test design phase. An overview of this session is given in Chapter 18.
- **Test result analysis** – The user testing creates a lot of data that has to be analyzed and is used for evaluating this project.
- **Documentation** – For this tool to be used for others than its developers it should contain a user documentation (see Chapter 17).
- **Presentation** – It is natural to present the results from the project for representatives from BEKK. They want to see what has been done, and be able to use the results of the project in the future.
- **Report** – Under the whole process the report is always in mind.
- **Delivery** – Finally the report and the software is delivered to BEKK as well as to NTNU. Hopefully, the delivery will not be the end of the project as the intention is for it to be an open source project and live on and be further developed in the future.

Chapter 6

Research Question and Method

GQM (Goal-Question-Metric) is a method that can be used to build research questions in a structured and organized matter. The method was originally developed by V. Basili and D. Weiss, and expanded with concepts from D. Rombach [21]. There are many years of academic research and practical experience behind the method and it is widely used in the field of software process improvement. Here GQM is used for structuring the projects research and the software evaluation.

The GQM-method can be described as shown in Figure 6.1. It contains four phases as described by [21]:

- **Planing** – In this phase the project is selected, defined, characterised, and planned. The result is a project plan.
- **Definition** – The goal, questions, metrics and hypotheses are defined and documented.
- **Data collection** – During this phase the actual data collection is performed. This results in a set of collected data.
- **Interpretation** – The collected data is processed with respect to the defined metrics into measurement results. This provides answers to the defined questions, which ultimately leads to an evaluation of the goal attainment.

The method uses a measurement model called a GQM-tree. An example is shown in Figure 6.2. There are three levels in the GQM-tree [20]:

- **Goals** – Describe the purpose of the measurements.
- **Questions** – Provide the answers supporting the goal when they are answered.

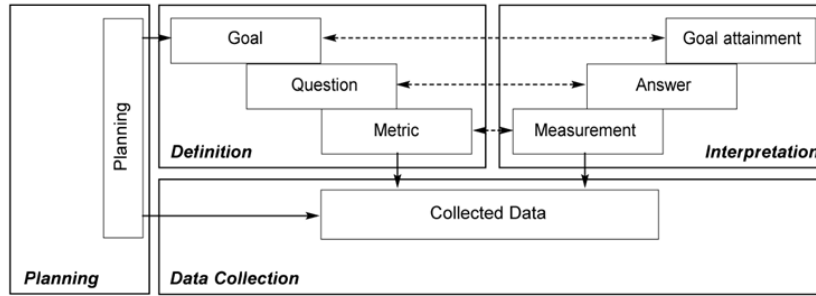


Figure 6.1: An overview of the GQM Method. Source: Solingen et. al. [21]

- **Metrics** – Associated with one or several questions. They are used for answering the questions in a measurable way.

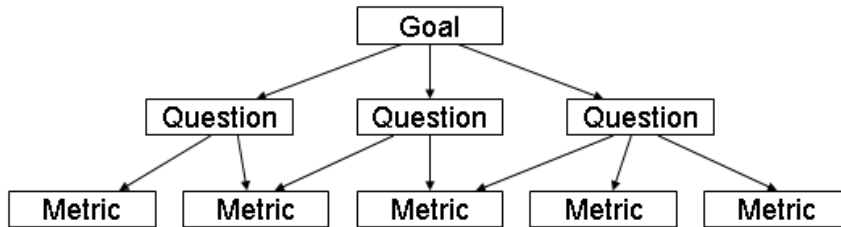


Figure 6.2: GQM-Tree Example.

The next section describes the project goal. Afterwards, we present the questions and metrics, used in the analysis in Chapter 19.

6.1 Goal

According to Stålhane[19], GQM uses a standard template for defining its goal :

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

There might be several goals for one project. However, for this project we have defined only one GQM-goal. It is rather large as its focus has three aspects. It might have been better to split it into three different GQM-goals, but for simplicity we decided to keep it as one.

GOAL:

Analyze **AutAT**
 for the purpose of **evaluating it as a prof of concept**

with respect to **usability, quality and efficiency of writing acceptance tests**
as seen from **software developers'** point of view
in the context of **test driven development of web applications**.

6.2 Questions

The GQM questions must, as mentioned earlier in this chapter, contribute to the goal when answered. We have identified seven questions that support our goal. These questions are associated with one or several metrics in Section 6.3. An overview of these associations is given in Table 6.20.

Question 1 *Quality*

How much does the use of AutAT increase the quality of tests with respect to a state-of-the-art framework like FitNesse?

In this context, quality is measured by the number of errors in the test file. Syntactic quality and the number of syntactic errors will be important as well as semantic errors. By semantic error we check to see if the right checks are done at the right web page and if the test setup which means having the right start page, is correct.

This question will be answered with metrics M17 – AutAT errors, and M18 – FitNesse errors, which will be used to compare AutAT with the state-of-the-art framework FitNesse. We also believe that the number of FitNesse errors are effected by the testers knowledge to FIT which is measured by M1 – FIT knowledge as FitNesse is an extension to FIT.

Question 2 *Efficiency*

What is the increase in efficiency by developing acceptance test with AutAT?

It is natural to think of efficiency as the time used to definine different kinds of tests. By defining the same tests with AutAT and a state-of-the-art framework like FitNesse, differences in time used can be measured. The metrics used for answering this question are M2 – AutAT time usage and M3 – FitNesse time usage. M1 – FIT knowledge will probably affect FitNesse time usage as FitNesse is an extension to FIT.

Question 3 *Usability*

How good it the perceived usability of the AutAT test tool?

The perceived usability is measured by M7 – AutAT's ease of use and M8 – FitNesse's ease of use to be able to compare them. M9 – Compared ease of use is used to see how the testers compare these tools. However, it is not only the ease of use that will affect the usability. We believe that the syntax complexity is important as well. So we use M10 – AutAT's syntax complexity,

M11 – FitNesse’s syntax complexity and M12 – compared syntax complexity for being able to see what the user think about the syntax. The same will be done with the overview as we believe that being able to get an overview of the tests will increase the usability. We use M13 – AutAT’s overview, M14 – FitNesse’s overview and M15 – Compared overview for measuring the overview. In addition the ability to modify tests are important so M16 – Modifying tests, is to be considered when deciding upon the question of usability.

Question 4 *Learning*

How easy does the user feel it is to learn to use AutAT compared to a state-of-the-art framework like FitNesse?

For measuring how easy an user feel it is to learn AutAT we use M4 – AutAT’s ease of learning, and M5 – FitNesse’s ease of learning. We will also compare them by using M6 – Compared ease of learning. The testers prior FIT knowledge will probably affect the results so M1 is importantm because as mentioned earlier FitNesse is an extension to FIT.

Question 5 *GUI*

What are the strengths, weaknesses and possibilities of the AutAT application and especially its graphical interface?

This question addresses some of the basic user feedbacks that are measured by M19 – Users feedback. This is important as it focuses on other aspects than the rest of the questions. It requires a qualitative analysis in order to answer this question.

These questions are linked to the metrics in the next section in Table 6.20 on page 24.

6.3 Metrics

The GQM-metrics are used to give answers to the questions by measuring different areas of interest as mentioned earlier. The metrics are listed in the tables 6.1 to 6.19.

Name	M1: FIT knowledge
Definition	Experience in using FIT. The user evaluates his or hers experience on a 4 point scale: <ul style="list-style-type: none"> • 4 - Using it at a daily basis. • 3 - Tried it. • 2 - Heard about it. • 1 - Unknown.
When to measure	During the user test session after the prototype is finished.
Procedure for measuring	Using a question sheet where the tester uses the scale presented in the definition.
Expected value	Generally expecting none or little knowledge and experience and should as a result average somewhere between 1 and 2.

Table 6.1: Metric 1 – FIT knowledge

Name	M2: AutAT time usage
Definition	Time spent on defining five different exercises in AutAT. The exercises can be found in Appendix B.
When to measure	During the user test session after the prototype is finished.
Procedure for measuring	Measure the time in minutes a user spend on defining five different types of tests in AutAT.
Expected value	AutAT should be quite fast which means no more than 7 minutes per test.

Table 6.2: Metric 2 – AutAT time usage

Name	M3: FitNesse time usage
Definition	Time spent on defining five different exercises in FitNesse. The exercises can be found in Appendix B.
When to measure	During the user test session after the prototype is finished.
Procedure for measuring	Measure the time in minutes a user spend on defining five different types of tests in FitNesse.
Expected value	FitNesse should be quite slow, but no more than 10 minutes per test.

Table 6.3: Metric 3 – FitNesse time usage

Name	M4: AutAT’s ease of learning
Definition	A user’s perceived ease of learning AutAT. It is evaluated by presenting it as the statement “easy to learn” when it comes to AutAT upon which the tester answers on a 4 point scale: <ul style="list-style-type: none"> • 4 - Agree. • 3 - Somewhat agree. • 2 - Somewhat disagree. • 1 - Disagree.
When to measure	During the user test session after the prototype is finished.
Procedure for measuring	Using a question sheet with the statement and the scale presented in the definition above.
Expected value	AutAT should be easy to learn, which means that it should score on average somewhere between 3 and 4.

Table 6.4: Metric 4 – AutAT’s ease of learning

Name	M5: FitNesse’s ease of learning
Definition	A user’s perceived ease of using FitNesse. It is evaluated by presenting it as the statement “easy to learn” when it comes to FitNesse upon which the tester answers on a 4 point scale: <ul style="list-style-type: none"> • 4 - Agree. • 3 - Somewhat agree. • 2 - Somewhat disagree. • 1 - Disagree.
When to measure	During the user test session after the prototype is finished.
Procedure for measuring	Using a question sheet with the statement and the scale presented in the definition above.
Expected value	FitNesse should be quite difficult to learn which means that the result should average somewhere close to 2.

Table 6.5: Metric 5 – FitNesse’s ease of learning

Name	M6: Compared ease of learning
Definition	A user's perceived ease of learning AutAT compared to FitNesse. It is evaluated by presenting it as the statement "AutAT is easier and faster to learn than FitNesse" upon which the tester answers on a 4 point scale: <ul style="list-style-type: none"> • 4 - Agree. • 3 - Somewhat agree. • 2 - Somewhat disagree. • 1 - Disagree.
When to measure	During the user test session after the prototype is finished.
Procedure for measuring	Using a question sheet with a statement and the scale presented in the definition above.
Expected value	AutAT should be easier to learn than FitNesse which means that the result should average somewhere 3 and 4.

Table 6.6: Metric 6 – Compared ease of learning

Name	M7: AutAT's ease of use
Definition	A user's perceived ease of using AutAT. It is evaluated by presenting it as the statement "ease of use" when it comes to AutAT upon which the tester answers on a 4 point scale: <ul style="list-style-type: none"> • 4 - Agree. • 3 - Somewhat agree. • 2 - Somewhat disagree. • 1 - Disagree.
When to measure	During the user test session after the prototype is finished.
Procedure for measuring	Using a question sheet with the statement and the scale presented in the definition above.
Expected value	AutAT should be easy to use, which means that it should score on average somewhere between 3 and 4.

Table 6.7: Metric 7 – AutAT's ease of use

Name	M8: FitNesse’s ease of use
Definition	A user’s perceived ease of using FitNesse. It is evaluated by presenting it as the statement “ease of use” when it comes to FitNesse upon which the tester answers on a 4 point scale: <ul style="list-style-type: none"> • 4 - Agree. • 3 - Somewhat agree. • 2 - Somewhat disagree. • 1 - Disagree.
When to measure	During the user test session after the prototype is finished.
Procedure for measuring	Using a question sheet with the statement and the scale presented in the definition above.
Expected value	FitNesse should be quite difficult to use which means that the result should average somewhere close to 2.

Table 6.8: Metric 8 – FitNesse’s ease of use

Name	M9: Compared ease of use
Definition	A user’s perceived ease of using AutAT compared to FitNesse. It is evaluated by presenting it as the statement ”AutAT is easier to use than FitNesse” upon which the tester answers on a 4 point scale: <ul style="list-style-type: none"> • 4 - Agree. • 3 - Somewhat agree. • 2 - Somewhat disagree. • 1 - Disagree.
When to measure	During the user test session after the prototype is finished.
Procedure for measuring	Using a question sheet with a statement and the scale presented in the definition above.
Expected value	AutAT should be easier to use which means that the result should average somewhere between 3 and 4.

Table 6.9: Metric 9 – Compared ease of use

Name	M10: AutAT’s syntax complexity
Definition	The perceived complexity of syntax of AutAT. It is evaluated by presenting it as the statement “simple syntax” when it comes to AutAT upon which the tester answers on a 4 point scale: <ul style="list-style-type: none"> • 4 - Agree. • 3 - Somewhat agree. • 2 - Somewhat disagree. • 1 - Disagree.
When to measure	During the user test session after the prototype is finished.
Procedure for measuring	Using a question sheet with a statement and the scale presented in the definition above.
Expected value	AutAT should have an easy syntax which means that the answers should be better than 3.

Table 6.10: Metric 10 – AutAT’s syntax complexity

Name	M11: FitNesse’s syntax complexity
Definition	The perceived complexity of syntax of FitNesse. It is evaluated by presenting it as the statement “simple syntax” when it comes to FitNesse upon which the tester answers on a 4 point scale: <ul style="list-style-type: none"> • 4 - Agree. • 3 - Somewhat agree. • 2 - Somewhat disagree. • 1 - Disagree.
When to measure	During the user test session after the prototype is finished.
Procedure for measuring	Using a question sheet with a statement and the scale presented in the definition above.
Expected value	FitNesse is believed to have a rather poor syntax and should on average be rated somewhere close to 2.

Table 6.11: Metric 11 – FitNesse’s syntax complexity

Name	M12: Compared syntax complexity
Definition	The perceived complexity of syntax compared of AutAT compared to FitNesse. It is evaluated by presenting it as the statement “AutAT has simpler syntax than FitNesse” upon which the tester answers on a 4 point scale: <ul style="list-style-type: none"> • 4 - Agree. • 3 - Somewhat agree. • 2 - Somewhat disagree. • 1 - Disagree.
When to measure	During the user test session after the prototype is finished.
Procedure for measuring	Using a question sheet with a statement and the scale presented in the definition above.
Expected value	AutAT should have an easier syntax which mens that the result should average close to 4.

Table 6.12: Metric 12 – Compared syntax complexity

Name	M13: AutAT’s overview
Definition	How well AutAT’s interface is arranged, so it is easily surveyable. It is evaluated by presenting it as the statement “easily surveyable” when it comes to AutAT upon which the tester answers on a 4 point scale: <ul style="list-style-type: none"> • 4 - Agree. • 3 - Somewhat agree. • 2 - Somewhat disagree. • 1 - Disagree.
When to measure	During the user test session after the prototype is finished.
Procedure for measuring	Using a question sheet with a statement and the scale presented in the definition above.
Expected value	AutAT should have a well arranged interface so the result should average close to 4.

Table 6.13: Metric 13 – AutAT’s overview

Name	M14: FitNesse’s overview
Definition	How well FitNesse’s interface is arranged, so it is easily surveyable. It is evaluated by presenting it as the statement “easily surveyable” when it comes to FitNesse upon which the tester answers on a 4 point scale: <ul style="list-style-type: none"> • 4 - Agree. • 3 - Somewhat agree. • 2 - Somewhat disagree. • 1 - Disagree.
When to measure	During the user test session after the prototype is finished.
Procedure for measuring	Using a question sheet with a statement and the scale presented in the definition above.
Expected value	FitNesse should be rated as a poorly arranged interface so the result should average close to 2.

Table 6.14: Metric 14 – FitNesse’s overview

Name	M15: Compared overview
Definition	How well AutAT’s interface is arranged, so it is easily surveyable, compared to FitNesse. It is evaluated by presenting it as the statement “AutAT is easier surveyed than FitNesse” upon which the tester answers on a 4 point scale: <ul style="list-style-type: none"> • 4 - Agree. • 3 - Somewhat agree. • 2 - Somewhat disagree. • 1 - Disagree.
When to measure	During the user test session after the prototype is finished.
Procedure for measuring	Using a question sheet with a statement and the scale presented in the definition above.
Expected value	AutAT should have a better arranged interface so the result should average close to 4.

Table 6.15: Metric 15 – Compared overview

Name	M16: Modifying tests
Definition	AutAT enables easy modification of tests compared to FitNesse. It is evaluated by presenting it as the statement “AutAT tests are easier to modify than FitNesse tests” upon which the tester answers on a 4 point scale: <ul style="list-style-type: none">• 4 - Agree.• 3 - Somewhat agree.• 2 - Somewhat disagree.• 1 - Disagree.
When to measure	During the user test session after the prototype is finished.
Procedure for measuring	Using a question sheet with a statement and the scale presented in the definition above.
Expected value	AutAT test should be perceived as easier to modify which means somewhere between 3 and 4 on the scale.

Table 6.16: Metric 16 – Modifying tests

Name	M17: AutAT errors
Definition	Number of errors in tests defined by using AutAT. Errors counted are semantical and syntactical as explained in Question 1 – Quality.
When to measure	During the user test session after the prototype is finished.
Procedure for measuring	Counting the number of errors made by the user in five tests defined by a user using AutAT.
Expected value	AutAT tests should have almost no errors meaning 2 to 4 error totaly as we believe that the graphical interface prevents most syntactic errors.

Table 6.17: Metric 17 – AutAT errors

Name	M18: FitNesse errors
Definition	Number of errors in tests defined by using FitNesse. Errors counted are semantical and syntactical as explained in Question 1 – Quality.
When to measure	During the user test session after the prototype is finished.
Procedure for measuring	Counting the number of errors errors made by the user in five tests defined by a user using FitNesse.
Expected value	FitNesse tests should have some errors as tying errors and similar will cause some syntactic errors.

Table 6.18: Metric 18 – FitNesse errors

Name	M19: User feedback
Definition	The feedback which the users comes up with after using the tool for the first time. The reason for this is to capture aspects that the other questions do not cover. The emphasis will be on the strengths, weaknesses and opportunities of AutAT.
When to measure	During the user test session after the prototype is finished.
Procedure for measuring	Using a question sheet with open areas for feedback on several subject concerning AutAT and its basic features.
Expected value	There may be some bugs, but the main concept is good. This metric may provide feedback that the other metrics might not capture.

Table 6.19: Metric 19 – User feedback

The metrics can be linked to the questions according to matrix in Table 6.20.

The results from the data collection and the interpretation will be presented in Chapter 18.

Metrics	Questions				
	Q1 - Quality	Q2 - Efficiency	Q3 - Usability	Q4 - Learning	Q5 - GUI
M1 - FIT knowledge	X	X	X	X	
M2 - AutAT time usage		X			
M3 - FitNesse time usage		X			
M4 - AutAT's ease of learning				X	
M5 - FitNesse's ease of learning				X	
M6 - Compared ease of learning				X	
M7 - AutAT's ease of use			X		
M8 - FitNesse's ease of use			X		
M9 - Compared ease of use			X		
M10 - AutAT's syntax complexity			X		
M11 - FitNesse's syntax complexity			X		
M12 - Compared syntax complexity			X		
M13 - AutAT's overview			X		
M14 - FitNesse's overview			X		
M15 - Compared overview			X		X
M16 - Modifying tests			X		
M17 - AutAT errors	X				
M18 - FitNesse errors	X				
M19 - User feedback					X

Table 6.20: The Relationship between Questions and Metrics.

Part II

Prestudy

This part is the background information for this project in general and for the contribution which is the next part in particular. It looks at eXtreme Programming (XP), testing and state of the art testing technologies before it evaluates different technology platforms for AutAT. The two last topics in this part is the Eclipse architecture and the Graphical Editing Framework (GEF).

Chapter 7

eXtreme Programming – XP

eXtreme Programming (XP) is a relatively new software development methodology. It emerged in the middle of the 1990s, as a result of Kent Beck¹ wanting to try an alternative to the standard waterfall model normally used at the time (1996) [16]. XP is now the most popular of the agile methodologies.

The main goal of using XP is to lower the cost of change in a development process. In a normal waterfall model, the cost of change ascends rapidly as time moves forward. The aim of XP is to keep the cost of change lower. An illustration of this is given in Figure 7.1.

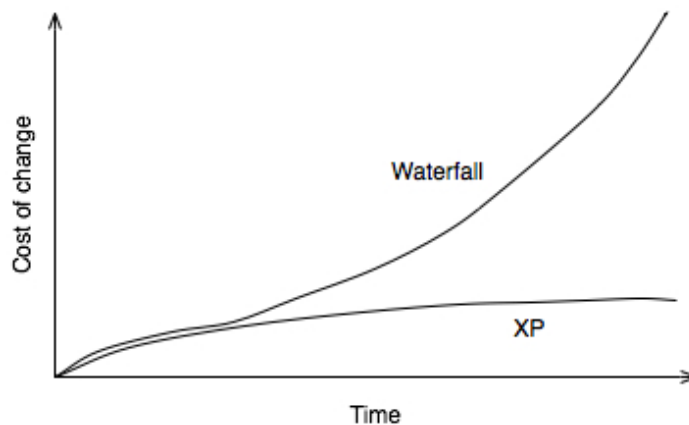


Figure 7.1: Cost of change

While the traditional waterfall process is a sequential model [22], XP is an iterative development methodology. The software is developed one piece at a time, in a series of iterations. Each iteration can be seen as a “mini waterfall project”. The result of an iteration is the release of new functionality for the users that can be used right then without waiting for the next iterations. This leads to each iteration adding business value to the customers. At the end of all iterations, the developed system is complete.

¹http://en.wikipedia.org/wiki/Kent_Beck

XP introduces a set of basic *values*, *principles* and *practices* that tries to make the project better prepared to handle, and indeed embrace change throughout the process.

Note that the description of XP in this chapter is based on the revised second edition of “eXtreme Programming explained” by Kent Beck [23] where and the values, principles and practices have been revised after a few years of experience with XP.

7.1 Values

XP use five values to help a project team focus on what is most important for the team (as opposed to the individual developer) and guide the development process: *Communication*, *Simplicity*, *Feedback*, *Courage* and *Respect*.

Communication. It is important to exchange knowledge between project members. When someone in a team has a problem, it is often likely that some other team member can think of an easy solution from previous experience. If not, it will likely be both easier and faster to solve the problem when more than one person is working on it. When a solution to a problem is shared among all team members similar problems will be solved more quickly in the future.

Simplicity. The phrase “Do the simplest thing that could possibly work” is one of the most used guidelines when dealing with XP. The focus is to do exactly that. The simplest solution that will get the job done. Simple solutions are easier to comprehend, understand and implement for the project team. The goal is not to provide a solution for “what may possibly happen tomorrow”, but design the system for the needs of today and (if need be) change it to accomodate new demands in the future.

Feedback – is vital to the project progress. The development team needs feedback from the customers, the system and the team itself. When feedback is given, the team can adjust to changing circumstances. As circumstances are always changing, feedback must be a continous activity from the very start of the project.

Courage will be manifested in many ways, supporting the other values of XP. One must have the courage to implement the simplest solution, instead of a more complex one that “might be needed later”. One needs courage to do something about a problem. One needs courage to speak the truth about something, in turn enabling good communication. Without a bit of courage (and encouragement to show it) the other values of XP will be hard to achieve.

Respect is a value that supports the four other of XP. In order to make a project work, any project with any development methodology, the participants need to respect one another, what they are doing and the goal they are working towards. To quote Kent Beck [23, p. 21]:

If members don't care about each other and what they are doing, XP won't work. If members of a team don't care about a project, nothing can save it.

7.2 Principles and Practices

The practices of XP are the manifestation of how to behave in a project. They are down-to-earth advice of how to organize a team and the work to be done. Practices alone are not worth much. However, when linked to a set of values they make sense. This linking is done by the set of principles provided by XP. These principles are described in detail in [23, ch. 5], we will not discuss them here.

As for the practices of XP, they are taken to extremes in what they say. Not all practices will suit all teams and organizations, but one of the things about these practices is that a team may choose to use a subset of them, selecting the ones which will likely result in the greatest improvement for the team. The most central practices are described here, for a complete description, see [23, ch. 7].

Sit together in a space big enough for the whole team. This will allow the team to communicate easily, and everyone can get help and discuss opinions and ideas at once.

The Whole Team is all the people, with their individual skills and areas of expertise necessary for the project to succeed. This team includes representatives from the customer, someone who can clarify requirements and make decisions.

Energize work simply means to respect oneself. Don't work more than you can be productive, stay home when you are sick. The rationale for this practice is that a 60 hour week is not necessarily more productive than a 40 hour week. Indeed, in many cases the extra hours are counter-productive, as well as demotivating.

Pair programming is about two minds thinking better than one, and two sets of eyes seeing better than one. The process is a dialog between two people, taking turns at the keyboard. The effects are clear, pair programmers keep each other on track, brainstorm efficiently, clarifies ideas and lowers frustration by taking over when your partner is stuck.

User Stories are short descriptions of functionality visible to the user. "A user story is the smallest amount of information (a step) necessary to allow the customer to define a path through the system" [1]. User stories are the basis for early planning in XP, as opposed to requirements or use cases in more traditional methodologies. By using stories early and estimating how much effort it is needed to implement the story, the customer is forced to consider

which are most important to achieve the business goals the system is meant to meet, and the cost of implementing each of them.

Short planning cycles, release often leads to more detailed and more accurate plans. XP is an iterative methodology. The first plans are high level, stating which user stories are to be included in which release. The detailed planning (and design) of each of these stories are delayed until they are actually needed. This helps keep the focus on what is important right now, not what is to be done in three months.

Test-First Programming (or Test Driven Development (TDD)) refers to the development model in XP. The idea is to write tests for each part of software before the software itself is implemented. This goes all the way from acceptance tests at the top, down to unit tests for each low-level module. More on TDD is given in Section 8.3.

Incremental Design refers to the value of simplicity. We do not make a design today for what *might* come next week. If, when the time comes, the design does not fit, refactor² the software to make the design fit the new constraints.

Continuous integration should be automatic and part of any XP project. An integration build is performed any time the code base has changed. When someone updates a part of the software, the software is rebuilt, all tests are run (unit as well as acceptance tests), and a deployment is performed (if only to a testing environment). The goal is to avoid problems when integrating and deploying the software as a whole later. Using traditional methods, there has often been a problem when integrating the finished modules. When this integration process starts early, these problems should no longer be problems.

²Refactor – rewrite, redesign or reimplement parts of the already implemented software to make it better (faster, easier to understand etc.) while still keeping the same functionality.

Chapter 8

Testing

Testing is an essential part of any software development process. Many definitions of testing have been used over the years. Some of these focus on testing as a means to prove that no errors are present in a program or that the software does what it is supposed to do. However, neither of them is correct. Any non-trivial software system will contain bugs [2]. A test can not prove that an error is not present, it can only prove that an error *is* present. Even though a program does what it is supposed to do, it might still be full of errors if it also does things it is *not* supposed to do. We have adopted the following definition of testing software, from [16].

Testing is the process of executing a program with the intent of finding errors.

This definition of testing leads to the realization of what constitutes a well designed test. A well designed test is one that has a high probability of proving the presence of errors in the code it tests. A successful test is one that finds bugs, not one that run through without any errors. We will not discuss test-case design here.

The testing of a software system is performed at different levels and at different times during the development process. The first section in this chapter takes a look at a traditional model of testing a software system. The next section describes several testing paradigms, before we explain the workings of Test Driven Development, a rapidly growing type of testing. Lastly we will look at some of the characteristics of testing web applications and some of the related key challenges.

8.1 V-model

The V-model of software testing is shown in Figure 8.1. This model shows a traditional view of how to develop an application and how testing relates to development.

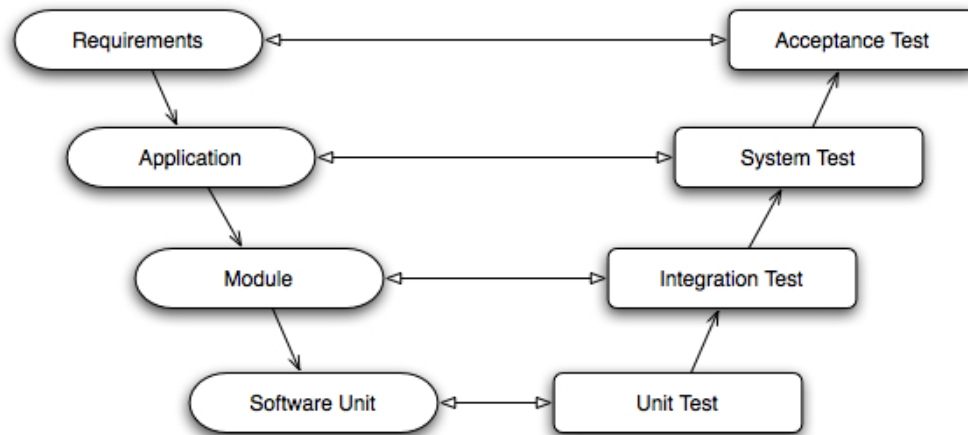


Figure 8.1: V-model of testing

Any software development process starts with a set of requirements from a customer. The requirements describe what the customer wants to achieve by developing the software product. The system requirements describe some business cases that the customer wants to address, and state what the system must achieve in order to successfully address the business case.

In order to make sure that the system does what the customer wants it to and thus meet his business need, a set of Acceptance Tests are created. These tests are based on the requirements for the system, and illustrate how the user will use the system. The acceptance tests are used by the customer to validate that the system actually does what she wants it to do.

The requirements lead to the design of a computer system. This system is delivered to the customer who uses it to meet his business needs. A System Test is created to check that the system meets its specification. Although both the Acceptance Tests and the System Test check the system as a whole, the focus is a bit different. The System Test checks that the system is developed in accordance with the way it was defined in the specification while the Acceptance tests will check that the system is actually what the customer wants and needs.

The purpose of acceptance and system testing is to provide confidence in the functionality of the system. This might seem like a contradiction to the definition of testing given in the introduction to this chapter, but it is not. When a test that is designed to uncover errors fails to uncover errors, the confidence that the system is working as wanted increases.

Most software systems will be divided into modules or subsystems. A module is a part of the system responsible for providing a part of its total functionality. The decomposition of the system into modules lead to a set of Integration Tests. These tests check that a module does what it is supposed to do, and that communication between the module and its surroundings is correct i.e.

that it adheres to the interface descriptions it implements.

A module is composed of still smaller parts. These parts are called components or units. In an object-oriented world classes and their operations will be units. Units are tested with Unit Tests. A unit test will check that the unit performs correctly in *isolation*, i.e. that its operations will return correct results and that its state is correct after operations are executed.

The arrows between the different types of tests in Figure 8.1 indicate a hierarchy of tests. At the bottom we have unit tests for the smallest units of software. When these units are combined into a module, they are tested as a whole by the integration tests. When the modules are put together to form the complete system, the system tests will ensure the system behaves as prescribed. When a customer is handed the finished system, the acceptance tests check that the software performs the way the customer needs it to perform in order to meet the business needs. Accordingly, there does not seem to make much sense to perform any of the tests at a higher level before all the tests at lower levels are declared OK. Surely, a integration test of a module will never be expected to pass when some of the units in the module do not pass their unit tests, and so on.

8.2 Different types of testing

Two common strategies for testing software are black box testing and white box testing. Both of these two strategies involve running software on a computer. A third form of testing is done without using a computer – human testing.

8.2.1 Black Box Testing

Black box, data-driven or input/output testing tests a piece of software from a requirement point of view. Tests consist of a set of input values and expected output values. The results of running the software with the input values are compared to the expected output values. If they match, the test pass, if not it fails.

In order to find *all* errors in a program, a possibly huge number of input and output values must be provided [16]. This includes both valid and invalid input values. As in many cases, the validity of an input value is dependent on previous input, all sequences of input values must also be tested. From this, it is easy to say that an exhaustive test using all input values is virtually impossible in practice. One exception is systems with binary input data (such as a set of sensors) and binary output (such as an alarm going off or not) that are often tested extensively, using all possible combinations of input values. By selecting the right set of input/output values however, one might say a piece of software has been tested enough to believe that it will perform as described in its specification.

8.2.2 White Box Testing

White box testing is another traditional way of testing software. While black box testing tests software from the outside, not caring about how it looks on the inside, white box testing use knowledge of the software's internal logic when designing tests. White box tests are designed to execute all statements in a part of the software. Sometimes white box testing is also referred to as logic-driven testing since the test design is *driven* by knowledge of the software's internal logic when designing the tests.

However, only executing all statements in the program is not enough. One goal when designing white box tests is to achieve exhaustive path testing. This means executing all possible paths through a program. A program will usually have many points where a logic decision (i.e. an if-statement) determines the further execution. Like exhaustive black box testing, this means (in most cases) a huge number of tests [16]. However, one can design tests that will produce both a *true* and a *false* outcome at each branch point. This will lead to all branches in a program being executed at least once.

One problem with white box testing is that it is quite possible to test all paths through the software, but if you ignore the specification, the software might still be erroneous. An exhaustive path coverage test may not show whether the software actually produces the results it should according to its specification.

8.2.3 Human Testing

A form of testing often neglected is human testing. Many feel that a computer program is written solely for a computer to read, not humans, and the only way to test it is to execute it on a computer [16]. However, the human eye is often quite effective in finding errors in source code. Human testing can be divided into two quite similar categories: Program Inspections and Walkthroughs (sometimes called Desk Checking). Human testing can be viewed as a form of white box testing, as it uses the source code of a program directly. However, while the source code is used to design the input set for a white box test it is read by humans during the actual test in a human testing session.

Both *Program Inspections* and *Walkthroughs* [16, ch. 3] involve a team reading a piece of code before meeting to find errors in the code they have read. The purpose of the meeting is not to correct these errors, merely to record their possible existence. The basic difference between an inspection and a walkthrough is that in a walkthrough, the participants will "play computer", manually executing a test case, while in a inspection they will just read the program and use a set of check-lists to look for errors.

8.3 Extreme testing – Test Driven Development

With the advance of eXtreme Programming (see Chapter 7) and other similar agile methodologies for software development, a new form of testing has emerged. Some call it Extreme Testing, some call it Test-First Development and some use the name Test Driven Development (TDD). No matter what name you use, the concept is the same. Tests are written *before* the software it will test. Normally unit tests, integration tests and acceptance tests as described in Section 8.1 are specified. It is not so common to define system tests in a TDD project, but rather use the acceptance test more actively in the development organization.

All the tests must be automatic. This lets a developer easily run the tests to check how she is progressing and that nothing gets “broken” when adding new functionality or changing existing code. It is also easy to integrate with tools for continuous integration (described in Section 7.2).

Unit tests are written right before (a pair of) programmers start coding a new software unit. The tests follow two simple rules: all units must have a set of unit test before the coding begins, and all unit tests must pass before a software unit is released. The reasons for writing the test before the software itself are many:

- assure that the code will meet its specification
- express the result of the code
- provide a tool for telling when the software unit is finished (i.e. when all unit tests pass)
- gives a better understanding of the specification and requirements
- act as a safety net when refactoring designs later

A large software project may have hundreds, or indeed thousands of unit tests. These tests will be a valuable part of the application they test, and must be maintained the same way the application’s source code is maintained, ensuring they show the latest changes in requirements and design decisions.

Just as unit tests, acceptance tests are written before the software. Acceptance tests in XP are based on the user stories (see Section 7.2) of the project. When the development team starts analyzing a new user story in detail as part of a iteration planning process, they expect the customer to come up with a set of acceptance tests for that user story. The team itself may contribute to these tests to help the customer. These acceptance tests act as the specification of what the system must do in order to fulfill the business needs of the customer. XP states that acceptance tests must be automated, just as unit tests. However, in practice it is often difficult to automate *all* acceptance tests. To check the layout and coloring of a user interface for instance, is hard to automate fully. Checking values as displayed to the user, on the other hand, is relatively easy to automate.

Acceptance tests are performed by the customer in order to check the software is performing satisfyingly. However, it may also be (and most often is) used as part of the build process, like unit tests. Then the development team can easily check how many acceptance tests pass and how many fail and be able to tell something about the general progress of the development effort.

8.4 Testing Web Applications

Developing web applications introduces new testing challenges. Some of these are a result of the architecture of web applications, some are a result of users using different web browsers and a high focus on the GUI of the applications and some are a result of the development process.

A web application is based on a client-server architecture. The user interacts with the application through a web browser, sending requests and receiving responses from the server. In most web applications, the server will process all the business logic in the application while the browser (client) just shows the results to the user. The server is (most) often composed of several parts or layers. A typical web application will be deployed in a web container, have some business logic deployed on an application server and use some data stored in a database.

The layering of web applications make testing harder. When a component must be deployed in an application server to work properly, it is difficult to unit test the component. It needs to have a running instance of the application server (often a heavy process) which in turn might need a database connection. This database must be set up with a correct set of data before each test is run. As we see, a lot of dependencies are already present that complicates the test and will not let the component be tested in isolation. New frameworks for developing web applications emerge that focus on making the systems more testable. One example is the Spring framework ¹ that tries to make both the development environment and process more lightweight and suited for unit testing.

Many or most web applications use a database server to store data. The data stored in the database range from the content shown on the web pages to security constraints related to a user. In order to make integration tests that check that a component (that has already been unit tested) performs as expected when connected to a database, the *state* of the database must be consistent across test executions. For instance, in order to test that a user can log on, the user must be registered in the system before the test can be executed. The execution order of two tests also make impact. The state of the system might be changed by the first test, affecting the second. This is of course a problem with all systems that maintain state across different interactions, not only web applications.

¹<http://springframework.org>

So far we have focused on testing on the server side of a web application. However, also the client side must be tested. Traditionally, client side testing has been performed manually, typing values into forms, reading text and clicking on links and buttons and so on. Such testing is time-consuming and error prone. A tester need to manually type all input text, and manually move the mouse around to click on links. He must also wait for the responses from the web server to reach him. When typing values, it is easy to make typing mistakes, and it is easy to miss a typing error when reading text.

The fact that a web application is viewed in a browser is both a good and a bad thing. It removes the need for a user to install the software on his computer. The application can be reached from any computer with a browser and a connection to the internet, if we disregard the use of firewalls, ip-filters and other security measures. However, different browsers (and even different versions of the same browser) have different capabilities. The layout may vary, text may be misplaced and images may be located at different places. In addition, many web applications use JavaScript as part of the interaction. JavaScript capabilities are different in different versions, and is often called a “testing nightmare”. Due to the different capabilities of different browsers, at least some tests must be run using several browsers.

8.5 Summary

It is important to test new software to ensure high quality. Software testing will occur at several levels of detail within the application, ranging from the smallest modules or classes to functional acceptance testing.

Test can be designed either with or without knowledge of the internal logic of the software, and either before or after the application is implemented. The use of TDD in many projects emphasize the creation of a test before the software as a critical part of the design process, using automatic acceptance-, integration- and unit-tests.

Testing of web applications introduces new challenges related to the very nature of the applications. Some tests for web applications can hardly be automated, but many can. If they are automated, they can be executed more often, at a much lower cost. This will in turn help organizations using TDD to develop applications, and will probably increase the quality of the applications they develop.

Chapter 9

State of the Art - Existing Testing Technologies

This chapter will examine some of the open source solutions that are available for testing web-applications' features today. Some of these solutions are general for testing software systems, while others are created especially for testing web-applications.

We will show how a sample test of a web system may look using different solutions. The sample system is a simple CD database. It will let a user register artists, register CDs to these artists, and browse the registered artists. Our sample test will check that the welcome-page of the system is correct (its title is correct, a welcome message is displayed and links to listing and registering artists). Then we want to register a new artist. This will include checking that a form is correct (has a field for registering the artists name and in which genre it belongs) and submitting values for these fields. After submitting, a confirmation page shall be displayed to the user.

9.1 xUnit

The xUnit testing family started with Kent Beck publishing a unit test framework in 1999 [10]. The framework was written in the Smalltalk programming language under the name SUnit ¹. It was later ported to Java by Erich Gamma calling it JUnit. JUnit will be discussed in Section 9.2. Ports for other languages have later emerged, among others C++ has CppUnit, .Net has NUnit and Python has PyUnit. These are all free, open source software tools.

In addition to the basic xUnit family members there are several extensions that add to the functionality, targeting specialized domains instead of working as standalone tools [10]. Some examples of these extensions are XMLUnit that is an extension to JUnit and NUnit for testing XML, JUnitPerf which

¹<http://sunit.sourceforge.net/>

is an extension that tests code performance and scalability and HTMLUnit and jWebUnit that tests web-based applications on top of JUnit. HTTPUnit also tests web-based applications, but at a lower level, as it deals with HTTP requests and response objects.

In [10], Hamill states that the basic architecture of all xUnits is the same. The central classes of any xUnit framework are shown in Figure 9.1. The TestCase class is the base class for an unit test, all unit tests extends this class. A TestRunner class simplifies running tests and viewing details about the results, a GUI TestRunner is included in most of the xUnits to increase the visibility of the feedback. The TestRunner most often use a form of naming convention to decide which operations it will run on a TestCase implementation.

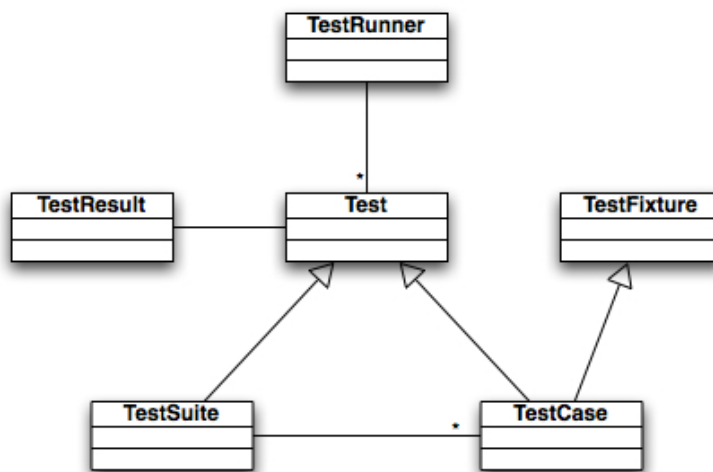


Figure 9.1: The xUnit Architecture.

The TestFixture class has the responsibility to ensure that the tests run in isolation and the state of the system is correct when a test is run. This is done in the two operations setUp() and tearDown() which are invoked just before a test and just after a test, respectively.

A TestSuite maintains a collection of TestCases. When the TestRunner executes a TestSuite, this will in turn run all the test cases it contains. The final base class is the TestResult class, which sole purpose is to report how the tests run. It collects any errors and failures as well as counting the tests that pass. These results can then be reported by the framework.

9.2 JUnit

The xUnit testing framework for Java is JUnit² which was create by Eric Gamma [10]. It is the de facto standard for Java testing [4]. The architec-

²<http://www.junit.org>

ture of JUnit follows that of the general xUnit as described in the previous section.

The framework has a GUI version that can run the tests. However, most Interactive Development Environments (IDEs) like Eclipse³ and IntelliJ⁴ have added support for JUnit tests. The advantage of this integration is that it is much easier to handle tests and run them when a developer always has the tool for the job at hand.

JUnit is the basis for jUnit extensions such as jWebUnit and HTMLUnit as mentioned earlier. These two extensions are described next.

9.2.1 jWebUnit

jWebUnit⁵ is an extension to jUnit for testing web pages. jWebUnit aims to make testing web pages and applications easy, providing a simple-to-use API⁶. This API provides methods for asserting the presence and correctness of links, general text, form input elements etc. jWebUnit also lets the user navigate simply by pressing links or buttons or by form entry and submission.

Our example test is shown in Listing 9.1. The *WebTester* class is responsible for the actual interaction with the web application and performing the checks.

Listing 9.1: jWebUnit example

```
1 public void testCDDbJWebUnit() {
2     WebTester tester = new WebTester();
3     tester.getTestContext().setBaseUrl("http://www.cddb.com");
4     tester.beginAt("/");
5
6     tester.assertTitleEquals("CDDb");
7     tester.assertTextPresent("Welcome_to_the_CDDb");
8     tester.assertLinkPresentWithText("List_all_artists");
9     tester.assertLinkPresentWithText("Register_new_artist");
10
11     tester.clickLinkWithText("Register_new_artist");
12
13     tester.assertTitleEquals("Register_new_artist");
14     tester.assertFormElementPresentWithLabel("Name");
15     tester.assertFormElementPresentWithLabel("Genre");
16     tester.assertSubmitButtonPresent("save", "Save_artist");
17     tester.setFormElementWithLabel("name", "Pink_Floyd");
18     tester.setFormElementWithLabel("Genre", "Rock");
19     tester.submit();
20
21     tester.assertTitleEquals("Artist_registered:_Pink_Floyd");
22 }
```

³<http://www.eclipse.org>

⁴<http://intellij.com>

⁵<http://jwebunit.sourceforge.net>

⁶Application Programming Interface

9.2.2 HTMLUnit

HTMLUnit⁷ is another jUnit extension for testing web pages. HTMLUnit will parse the HTML pages being tested, transforming it into a DOM-tree⁸. A DOM-tree (Document Object Model) is a object oriented view of a structured document. It consists of a root object that holds references to a hierarchy of sub objects. The test can then traverse this DOM-tree, performing checks or setting values on each element in the tree.

The example test using HTMLUnit is shown in Listing 9.2. As we see, HTMLUnit differs from jWebUnit in that it cannot directly perform many of the assertions that jWebUnit can. As an example, with jWebUnit we can check the presence of a link directly by calling the operation *assertLinkPresentWithText()* as seen on line 7 in Listing 9.1. The same check with HTMLUnit first must find a link by its *name* before we can check the actual text with a normal jUnit assertion, as seen on lines 8–9 in listing 9.2.

Listing 9.2: HTMLUnit example

```

1 public void testCddbhtmlUnit() throws Exception {
2     WebClient webclient = new WebClient();
3     URL url = new URL("http://www.cddb.com");
4
5     HtmlPage frontPage = (HtmlPage) webclient.getPage(url);
6     assertEquals("Cddb", frontPage.getTitleText());
7
8     HtmlAnchor listAnchor = (HtmlAnchor) frontPage.getAnchorByName(
9         "listArtists");
10    assertEquals("List_all_artists", listAnchor.asText());
11    HtmlAnchor registerAnchor = (HtmlAnchor) frontPage.
12        getAnchorByName("registerArtist");
13    assertEquals("Register_new_artist", registerAnchor.asText());
14
15    HtmlPage registerPage = (HtmlPage) registerAnchor.click();
16    assertEquals("Register_new_artist", registerPage.getTitleText()
17        );
18
19    HtmlForm form = registerPage.getFormByName("registerForm");
20    HtmlTextInput name = (HtmlTextInput) form.getInputByName("name"
21        );
22    HtmlTextInput genre = (HtmlTextInput) form.getInputByName("
23        genre");
24    name.setValueAttribute("Pink_Floyd");
25    genre.setValueAttribute("Rock");
26
27    HtmlPage confirm = (HtmlPage) form.submit();
28    assertEquals("Artist_registered:_Pink_Floyd", confirm.
29        getTitleText());
30 }

```

⁷<http://htmlunit.sourceforge.net>

⁸<http://www.w3.org/DOM/>

9.3 FIT

FIT (Framework for Integrated Test)⁹ is a general framework for acceptance testing created by Ward Cunningham.

Tests are defined using HTML tables in normal HTML documents. These documents can be edited in any editor, including those most familiar to customers, such as Microsoft Word and Excel. The first row of the table defines which Fixture (to be explained shortly) to use. Then, the following rows of the table are tests or actions performed with this fixture.

A central concept in FIT is the *Fixture*. A fixture is a special class or module to which the FIT testing engine will delegate the tests. A fixture can do just about anything, but in most cases they simply act as mediators. A mediator will transform the input from the table to a form suitable as input for a component of the software. It will then invoke an operation with the given input and transform the result back to a form that FIT can verify against the expected result in the table.

A basic example of a test table in FIT is given in Table 9.1. Here a Fixture called *eg.Division* is used. This fixture is an example where each row in the table is a single test. The two first columns, numerator and denominator are parameters while the last column, quotient() is the expected result. This particular example will first divide 1000 by 10 and check that the result equals 100, then divide 50 by 10 and check this result equals 5.

eg.Division		
numerator	denominator	quotient()
1000	10	100
50	10	5

Table 9.1: Basic FIT example showing division

FIT tests can be executed in several ways. A *FIT runner* will parse the input HTML page, execute the Fixtures within it and produce an output file. The output file will be a copy of the input file, with some cells in the tables colored green or red, telling whether the test passed or failed. Different flavors of such runners exist. The standard runner processes a single HTML page and is executed from a command line. Other command line runners take a folder as parameter and will process all HTML files in the folder and any subfolders. Yet other runners are created for use with build tools such as ANT¹⁰ or Maven¹¹, and can process a set of files or folders as part of the overall build of the system.

A Fixture must be executed by a runner in its own language, and can only test applications written in the same language. However, FIT runners are created for many different programming languages, including Java, C++, .Net, ruby

⁹<http://fit.c2.com>

¹⁰<http://ant.apache.org>

¹¹<http://maven.apache.org>

and Python, allowing developers to use FIT to test applications developed in any of the supported languages.

In most cases, the development team is expected to extend basic Fixtures, adapting them to the system under test. However, some fixtures for common actions and tests exist. Most of these specific Fixtures are created for the Java implementation of FIT. These include JdbcFixture for testing simple database access and WebFixture and HTMLFixture for testing web pages.

9.3.1 WebFixture

WebFixture is an extension to jWebUnit for use with FIT. It provides the same functionality as described in Section 9.2.1 for interacting with a web application. However, it use the FIT form of writing tests and the FIT testing engine for executing them. The cd database example test is shown in Table 9.2.

As we see from the example, the syntax use user-friendly keywords for performing *checks*, *pressing* links or submit buttons and *entering* values.

net.sourceforge.jwebunit.fit.WebFixture		
base url	http://www.cdadb.com	
begin	/	
check	title equals	CDDB
check	text present	Welcome to the CDDB
check	link	List all artists
check	link	Register new artist
press	link	Register new artist
check	title equals	Register new artist
check	form element present with label	Name
check	form element present with label	Genre
check	submit button present	Save artist
enter with label	Name	Pink Floyd
enter with label	Genre	Rock
press	submit	
check	title equals	Artist registered: Pink Floyd

Table 9.2: WebFixture example

9.3.2 HTMLFixture

HTMLFixture is the HTMLUnit extension to jUnit. HTMLFixture is for HTMLUnit what WebFixture is for jWebUnit. The same practice of dealing with full DOM representations of the tested web pages apply.

The first part of our example test is shown in Table 9.3.

com.jbergin.HtmlFixture			
http://www.cddb.com	frontPage		
Element	1	head	cddbHead
Element	2	body	cddbBody
Focus	cddbHead		
Type Focus	title	1	titleText
Text	CDDB		
Focus	cddbBody		
Has Text	Welcome to the CDDB		
Type	a	1	listLink
Type	a	2	registerLink
Focus	listLink		
Text	List all artists		
Focus	registerLink		
Text	Register new artist		
Click	registerPage		

Table 9.3: HTMLFixture example


9.4 FitNesse

FitNesse is an extension to the original FIT framework that was described in Section 9.3. The FitNesse project has decided to integrate the FIT framework with a *wiki*. A wiki is a web based editing system. It lets users edit all the web pages it manages, and provides a way to add new pages in a simple manner. The wiki framework consists of a set of templates controlling the layout of the pages, the users only have to contribute the actual content on the pages.

A special syntax is required for the wiki system to function. One part of this syntax deals with the creation of new pages and links to these pages from existing ones. When a word in a page is written using *CamelCase* (several word put together as one, with the first letter of each word capitalized), a new page is created for editing, and a link to this new page is placed at the page where the word is used.

The actual tests are, as in the original framework, written as HTML tables. The writing of tables in FitNesse is simplified by the wiki syntax. When editing a page, a user can add a table by starting a new line with a *pipe* symbol, |. These tables have the same meaning as when using FIT, and the same Fixtures can be used. In Figure 9.2, the sample test of the CD database is shown in FitNesse, first the normally displayed version, then the view when editing the test.

The same rules for running tests apply for FitNesse as for FIT. As discussed in Section 9.3, a runner can only run Fixtures developed in the same programming language. However, tests are arranged differently in FitNesse than in FIT. Tests pages are organized together in a TestSuite in a hierarchical manner. A TestSuite will consist of Tests and possibly new TestSuites. FitNesse runners




TrondMarius.

TestCddb

Test	net.sourceforge.jewbunit.fit.WebFixture
Edit	base url http://www.cddb.com
Versions	begin /
Properties	check title equals CDDB
Refactor	check text present Welcome to the CDDB
Where Used	check link List all artists
RecentChanges	check link Register new artist
Files	press link Register new artist
Search	check title equals Register new artist
	check form element present with label Name
	check form element present with label Genre
	check submit button present Save artist
	enter with label Name Pink Floyd
	enter with label Genre Rock
	press submit
	check title equals Artist registered: Pink Floyd

[\[.FrontPage\]](#) [\[.RecentChanges\]](#)



TrondMarius.

TestCddb

EDIT PAGE

```
!-net.sourceforge.jewbunit.fit.WebFixture-!  
base url | http://www.cddb.com |  
begin | / |  
check | title equals | CDDB |  
check | text present | Welcome to the CDDB |  
check | link | List all artists |  
check | link | Register new artist |  
press | link | Register new artist |  
check | title equals | Register new artist |  
check | form element present with label | Name |  
check | form element present with label | Genre |  
check | submit button present | Save artist |  
enter with label | Name | Pink Floyd |  
enter with label | Genre | Rock |  
press | submit |  
check | title equals | Artist registered: Pink Floyd |
```

Save | Spreadsheet to FitNesse | FitNesse to Spreadsheet | - Insert Fixture Table - ▾

Figure 9.2: FitNesse example using WebFixture

will execute all tests on the page if invoked on a single Test page. If invoked on a TestSuite page it will execute all tests on the TestSuite page as well as all tests on sub pages.

To execute a Test or TestSuite in FitNesse, the user can navigate to the web page and click a link to start the test and view the results right away. This corresponds to the normal way of running FIT test from a command line tool. In addition, there also exist FitNesse runners that can be integrated into build tools such as ANT and Maven, just as for FIT.

9.5 Selenium

The Selenium test system is in many ways similar to FIT using WebFixture. Selenium tests are written as HTML tables, as in FIT. The commands used in Selenium are similar to those of WebFixture, providing navigation and checks to validate pages, but the similarities end there.

While WebFixture is based on Java *simulating* a web browser, Selenium is based on JavaScript¹² and is executed using a *real* web browser. When the user want to execute a Selenium test she must use a web browser to connect to a web server running the Selenium installation. The server will provide the tests to the browser, the test is parsed in the browser, and the browser will execute the test. The user can then watch as the test progress, as well as view all results when the test is finished.

One minor limitation in Selenium is the fact that it can only test web applications located at the same server as the Selenium installation itself. This limitation is due to security limitations in JavaScript. The effect is that the tests and Selenium must be deployed at the same server as the system being tested. This also makes it harder to execute Selenium tests as part of a system build, but it is possible and work is well underway to make this integration easier.

If we assume that Selenium and the tests are deployed in a subfolder of our CD database, our example test will look as shown in Table 9.4.

9.6 Canoo WebTest

Canoo WebTest represents different way to write tests for web applications. While the previously discussed systems FIT, FitNesse and Selenium use HTML tables to define the tests, Canoo has chosen to use XML (eXtended Markup Language)¹³ for representing the tests and ANT to run the tests.

¹²<http://research.nihonsoft.org/javascript/CoreReferenceJS15/>

¹³<http://www.w3.org/XML/>

CD Database Test		
open	/index.html	
verifyTitle	CDDDB	
verifyTextPresent	Welcome to the CDDDB	
verifyText	link=listAllArtists	List all artists
verifyText	link=registerNewArtist	Register new artist
clickAndWait	registerNewArtist	
verifyTitle	Register new artist	
verifyElementPresent	name	
verifyElementPresent	genre	
verifyElementPresent	saveArtist	
type	name	Pink Floyd
type	genre	Rock
clickAndWait	saveArtist	
verifyTitle	Artist registered: Pink Floyd	

Table 9.4: Selenium example

Canoo defines an ANT task which consist of a series of *steps* to be performed when testing a web application. Each step can represent a check for a link, a check for text, form entry or submission or clicking a link.

The sample test of the CD database is shown as a Canoo WebTest in Listing 9.3.

Listing 9.3: Canoo WebTest example

```

1 <target name="testCDDDB">
2   <testSpec>
3     <config
4       host="www.cddb.com"
5       port="80"
6       protocol="http"
7       basepath="" />
8     <steps>
9       <invoke url="/" />
10      <verifyTitle text="CDDDB" />
11      <verifyText text="Welcome_to_the_CDDDB" />
12      <verifyElementText type="A" text="List_all_artists" />
13      <verifyElementText type="A" text="Register_new_artist"
14        />
15      <clickLink label="Register_new_artist" />
16
17      <verifyTitle text="Register_new_artist" />
18      <verifyElement type="FORM" text="registerForm" />
19      <selectForm name="registerForm" />
20      <verifyElement type="INPUT" text="name" />
21      <verifyElement type="INPUT" text="genre" />
22      <setInputField name="name" value="Pink_Floyd" />
23      <setInputField name="genre" value="Rock" />
24      <clickButton name="saveArtist" />
25
26      <verifyTitle text="Artist_registered:_Pink_Floyd" />

```

```
26         </steps>
27     </testSpec>
28 </target>
```

9.7 Summary and Comparison

In this chapter, we have looked at some of today's most commonly used systems for testing web applications. These systems are in many ways similar. They provide means for navigating web applications by clicking links and submitting values to forms. They also allow the tester to validate pages with regards to their title and content such as links and text.

The simplest systems are those based on jWebUnit. Both when using jWebUnit and when using WebFixture, the user can validate the presence of a link with a given text by using one simple command. When using HTMLUnit, Canoo and Selenium, this check is more complicated. All these systems require you to supply the HTML *name* attribute as well as the actual text you want to be displayed to the user as a link. HTMLUnit complicates this check even further as you must select the link before you can perform the actual check.

Another benefit of jWebUnit and WebFixture is the ability to validate input elements in HTML forms and enter values to these elements by using the text label displayed to the user on the web page. None of the other systems have this ability, but use the name attribute here as well. When using jWebUnit, the user can choose to use the name attribute if she wants to use this option instead.

The way in which a user defines tests is the most differentiating attribute of the testing systems. For a customer or user to be able to write and understand the tests, they must be defined in a "form" the user easily can relate to.

One fundamental difficulty with all the systems discussed in this chapter is to understand all the commands or keywords used in a test. They can be learned, memorized or written on a piece of paper for reference, but they may still seem foreign to customers.

Using JUnit (with either jWebUnit or HTMLUnit) involves writing Java code. While this would probably be the most natural choice for a programmer, for a non-programmer writing tests as JUnit tests will seem like a pretty impossible task. Most modern IDEs¹⁴ will give much assistance to writing tests correctly, providing syntax highlighting and code-completion. Still, the user needs to understand how a programming language works to make sense of such tests.

As for Canoo, which use XML for defining tests, the case is much the same as for the JUnit-based ones. XML is not a format any user or customer is used to reading or writing. As with Java code, modern tools can help by providing

¹⁴Integrated Development Environments

highlighting, auto-completion and validation, but a test will still be difficult to understand.

When using FIT, the tests are HTML tables. These tables can be created and modified with any tool that can save a file as HTML, including Microsoft Word, that may be familiar to customers. The problem is to understand how to write a test. Tests are defined by naming a fixture and then giving it commands. The customer must remember the commands, how they work and what parameters they will expect.

One additional challenge that presents itself with FIT, Canoo and Selenium is the problem of distributed editing. How can a development team and a customer synchronize their efforts with regards to test management? Often, several people working from separate locations will manage the tests together. A normal way to synchronize might include using e-mail to communicate the latest changes between all the people of the team. As tests often are quite dynamic, changes might occur often. These changes should be recorded to reflect how requirements change through the course of the project. Using e-mail for keeping track of the different versions and in particular the latest version will be a challenging task.

This is one of the problems FitNesse tries to solve. FitNesse will keep track of changes to the tests, and will keep all files at a centralized location. Then, all team members (both customers and developers) can view them and modify them. However, FitNesse still requires the user to know commands and Fixture names, as well as the wiki syntax used to define tests and testsuites.

In conclusion, we see that all the systems described in this chapter have challenges, in particular for non-programmers. What is needed is a system which allows the customer to easily create, modify and view tests and share these with the development team while at the same time providing the same functionality as the existing solutions.

Chapter 10

Technology Platform

This chapter describes three technology platform alternatives that can be used for creating the AutAT application. To make the selection we made a list with each platforms advantages and disadvantages. We consider this more than sufficient for both our supervisors and for us to decide which platform to use.

The first section will look at Web as the technology platform. The next section looks at AutAT as a standalone application, before Eclipse and its Plugin features are considered. Finally, will we summarize and decide which platform we are going to use.

10.1 Web

There are several arguments that can be used for and against using a Web Application as the platform for AutAT. The main reasons for selecting Web as the platform are:

- Easy to control when several users are using the tool at the same time so everyone in the project team will see the same version of a test. Providing version control of the tests will also be quite easy.
- Easy to make sure that all tests are stored in the same place.
- Everybody can use it with only a web browser installed, which makes it a lot easier to work with on different operating systems.

The main reasons for not choosing web is:

- Poor support for rich graphical functionality like drag and drop.
- Lots of clicking on links and buttons will result in a rather poor user experience due to loading time.
- Cut and paste functionality will be difficult to apply.

- The Web has poor support for creating a graphical solution like the one that we would want.

10.2 Standalone

There are many ways of building a “fat” client. The choice of underlying programming language does not affect the reasons for developing this kind of application. The advantages of building a “fat” standalone client are:

- Keyboard shortcuts for navigating.
- Enables Cut and Paste.
- Graphical layout which support drag and drop functionality for changing the “flow” on the pages.
- Table/property view of the tests where one can do changes directly.
- It can be designed in a way that makes it easy for software customers to define tests.
- There are few constraints restricting possible realizations of the application.

The disadvantages of creating a standalone application are:

- The software must be installed on the user’s computer.
- Can be difficult to handle distributed test modifications.
- A lot of functionality that is not core functionality requires services of other applications like support for distributed development and version control.

10.3 Eclipse Plugin

The main reasons for choosing to build a “fat” client which is integrated into the Eclipse Platform as a plugin is as follows:

- Integrated in an IDE (Interactive Development Environment) which many developers use.
- Keyboard shortcuts for navigating.
- Graphical layout which support drag and drop functionality for changing the “flow” on the pages.
- Table/property view of the tests where one can change do changes directly.
- Enables integration with other tools that are available as plugins such as the Team API

- There are many Eclipse plugins already available like GEF (Graphical Editor Framework)¹, that can be used when developing AutAT to provide parts of the functionality.

The reasons for not using Eclipse as a platform are:

- Might be confusing for a software customer with all the other Eclipse functionality or plugins.
- The Eclipse framework lay some restricts on the different application realizations that can be implemented.

By using the RCP possibility that is described next, one can turn an Eclipse plugin into an standalone application.

10.3.1 RCP - Rich Client Platform

The Eclipse Rich Client Platform² is the minimum set of plugins that are needed to build a rich application. This is possible because Eclipse, according to [17], is architected so its components can be used to build almost any client application.

As described in [17], RCP applications use the dynamic plugin model and the user interface is built with the same toolkits and extensions points. This approach enables us as application developers to customize the layout and functionality at a fine-grained level. Many applications can also benefit from using a wide range of the plugins that are available as they provide a lot of functionality. Examples are the Help UI and the Team API plugins.

10.4 Summary - The Choice

All of these platforms are probably good choices for this kind of application. However, the choice is to create an Eclipse plugin as it has most of the features that the standalone application provides. It can also be used with other plugins as a mean to proved a wide range of functionality within the same application. It can be transformed into a standalone application through RCP if the use of a plugin later is found inappropriate. This is an advantage as it provides flexibility and therefore can reduce some of the risks associated with the AutAT application concept. We also believe that the software customer should write tests together with the developers which means that an user interface that most developers can relate to is preferred.

¹<http://www.eclipse.org/gef>

²<http://www.eclipse.org/rcp/>

Chapter 11

The Eclipse Architecture

This chapter will give an introduction to the Eclipse Platform (or simply Eclipse), its core concepts, key components, central plugins and how they all fit together. As Eclipse is chosen to be the platform for AutAT, it is useful to understand how Eclipse is organized and how the plugin system works.

The Eclipse Platform is designed to be a foundation onto which it is possible to build and integrate a rich set of tools. Originally it was designed to be a tool for application development, but since its birth it has grown into something more. As the Eclipse Foundation states on its homepage¹:

The Eclipse Platform is an IDE for anything, and for nothing in particular.

The Platform architecture can be visualized as shown in Figure 11.1. The basic foundation is the *Eclipse Platform Runtime*. This part is responsible for actually running the Platform, interacting with the local file system and handling all the plugins that are available, and is described in more detail in Section 11.1.

A *workspace* is the set of items a user is currently working with. The workspace concept is discussed in Section 11.1.2.

For most users, Eclipse is synonymous with the *Workbench*. The workbench is the user interface (UI) of a running instance of the Eclipse Platform, and is what the user will interact with. The Workbench and the concepts related to it is discussed in Section 11.1.1.

Another key design decision fundamental to the Platform is the choice to make it platform independent. This choice will make it available to the widest possible user group. Today, the Platform is running on most operating systems, including MS Windows, Linux, Mac OS X and different flavours of Unix.

¹<http://eclipse.org>

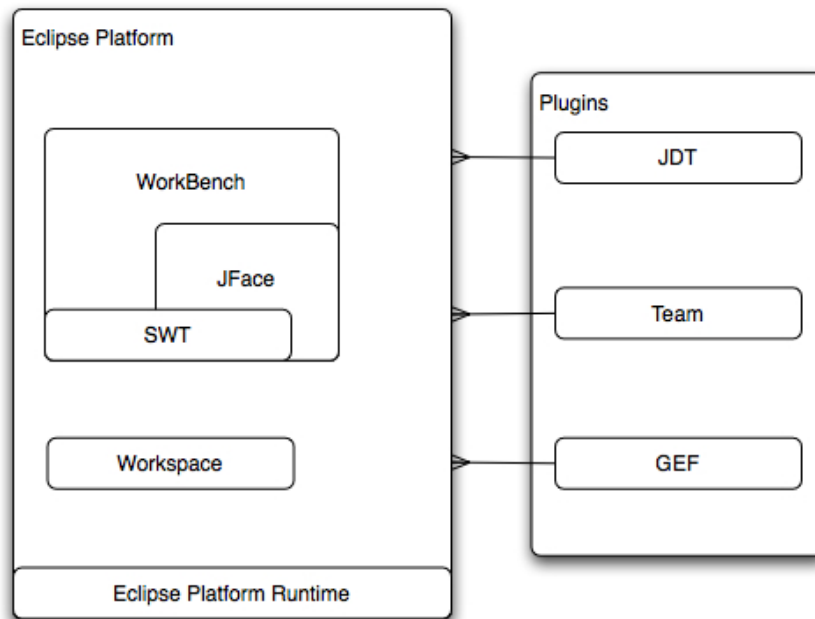


Figure 11.1: The Eclipse architecture

11.1 The Platform Runtime and Plugin Architecture

The Platform Runtime is the core of the Eclipse Platform. When a user starts Eclipse, an instance of the Runtime is created. This instance is then responsible for starting the other parts of Eclipse. All functionality in Eclipse is provided by plugins. A plugin is the smallest unit of functionality that can be delivered separately. The Runtime will keep track of all installed plugins, how they are configured and dependencies between them.

A basic set of *extension points* is provided by the Platform Runtime. An extension point is a point where a plugin can connect to provide its functionality. The Runtime keeps track of which plugins connect to which of its basic extension points. In addition to the basic points provided by the Runtime, a plugin may contribute extension points of its own. In this way a new plugin may use or extend functionality of plugins that are already installed in Eclipse. As stated, *all* functionality in Eclipse is provided by plugins. This has led to the fact that Eclipse itself is implemented as a set of plugins, where some plugins use the extension points provided by the Runtime, while others again use points provided by the first plugins. The Runtime is responsible for keeping track of all extension points, what they mean and how they are used.

While all installed plugins are listed and inspected at startup, they are not started before they are needed. When a plugin is activated, it will stay active for the lifetime of the Runtime instance. A plugin includes a XML file containing the definition of the plugin, the extension points it provides and which extension

points it provides extensions for. This is what lets the Runtime figure out when to activate a particular plugin, and allows “passive” activation of the plugins.

11.1.1 The Workbench, SWT and JFace

The User Interface (UI) of Eclipse is provided by a plugin called the *Workbench*. The workbench provides the overall structure of Eclipse, and provides many common extension points for new plugins. A typical example of how a running instance of Eclipse will look like is shown in Figure 11.2. This example shows a set of projects in the active workspace (described in Section 11.1.2) in the upper left part. The largest part is an active *Editor* showing the contents of a file. Files are normally manipulated using an Editor, and editors can specialize their behaviour to the content of a certain file type. Examples of editors are the Java source editor and XML editor. These will provide syntax highlighting and automatic completion. The two last parts of the sample are *views*, that show a document outline of the currently active editor (down left) and a list of tasks and problems (down right). Both of these are commonly used.

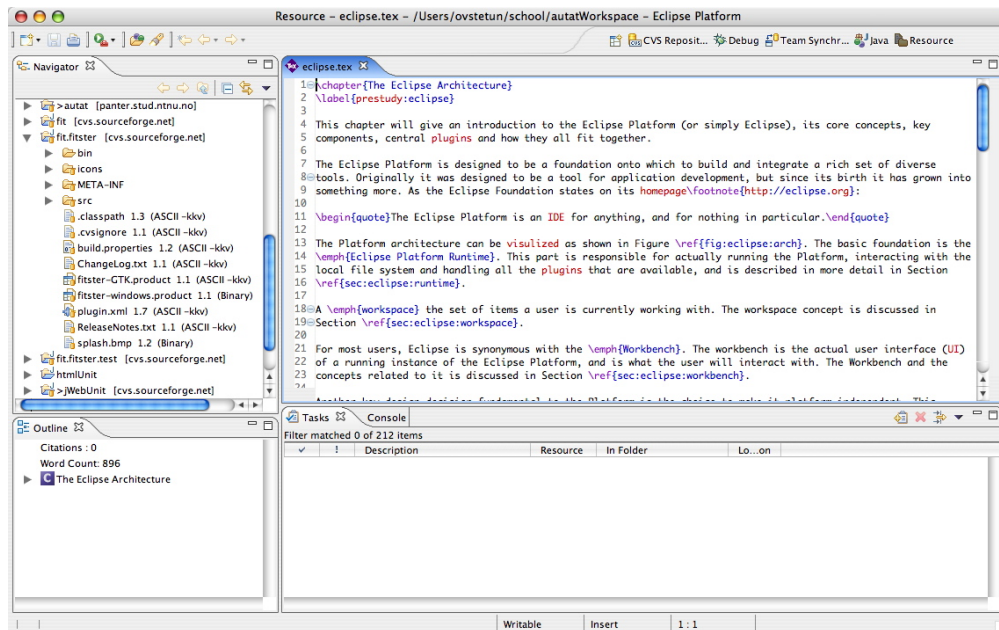


Figure 11.2: A sample view of the Eclipse Workbench

A plugin that provides functionality to Eclipse will often create its own editor for manipulating files. They may also provide one or more views for use. These can be integrated into a *perspective* that creates an initial layout, opening the views and editor as needed. Editors, views and other parts of the UI are developed using the SWT and JFace frameworks provided by Eclipse.

SWT (Standard Widget Toolkit) is a framework providing a common API for graphics and widgets in an OS (Operating System) independent manner. It will use the underlying window system as much as possible, making applications look native to the OS the user is running. SWT provides standard widgets a developer can use, such as windows, lists, tables and plain text areas.

JFace is a UI toolkit built upon SWT. It provides a simple way to perform common tasks such as showing images, providing dialogs for the user, guiding the user through a wizard and showing progress of longer operations. JFace also lets the developer define *actions* that the user can execute from anywhere in the UI. An action can be linked to a menu element, a button or to an item in a tool bar.

11.1.2 Workspaces

A workspace is a collection of directories and files the user is working with. The workspace is organized as one or more projects, a project maps directly to a directory in the file system. The plugins installed in Eclipse can manipulate the directory structure and the content of the files in a project. A plugin will use abstractions provided by the Platform Runtime to actually manipulate the file system.

A project may be tagged with one or more project natures. A project nature gives the project certain properties plugins can use to function properly. Such natures can define which plugins are applicable to the project as well as parameters for how to use a tool provided by this plugin, allowing for a greater opportunity for customization. As an example, the standard Java project nature will include properties such as a classpath, which compiler to use and dependencies to external libraries and other projects.

When a plugin is started, it may register itself as a *Listener* to the workspace. If it does, it will be notified when changes occur. These changes may be the addition of new files or directories or changes made to existing files. This will allow the plugin to act to these changes, i.e. to update a view of the changed resources.

11.2 GEF

The Graphical Editor Framework (GEF) is a plugin allowing developers to easily create graphical editors for Eclipse. The principles and patterns used by GEF are described in detail in Chapter 12.

11.3 The Team API

The Team API is a central Eclipse plugin for any working on a shared project. This plugin provides the means for placing a project in a central repository, which may again provide version control and configuration management. A repository provider just needs to use the extension points provided by the Team API to provide access to a new type of repository through a new plugin. Since all these plugins provide the same functionality according to the Team API, the user will interact with them in similar ways. This means she does not need to learn a whole new system for every type of repository.

Today, implementations for the most common team repositories exist, including CVS (Concurrent Versions System)², Subversion³ and ClearCase⁴.

²<http://www.gnu.org/software/cvs/>

³<http://subversion.tigris.org/>

⁴<http://www-306.ibm.com/software/awdtools/clearcase/>

Chapter 12

The Graphical Editor Framework

The Graphical Editor Framework(GEF)¹ which is the result of an Eclipse project², makes it easier for developers to create a rich graphical editor from an existing application model as described in [9]. It can be used to build all kinds of applications but is generally used for applications like activity diagrams, class diagram editors and state machines. This fits well with what we want AutAT to be able to do and it is therefore a natural technical choice.

This chapter will describe the structure of GEF and the patterns that are the foundation of the framework. However, the purpose is not to give an detailed overview of the GEF API or the patterns, but rather a short introduction to some of the major design decisions in GEF that are important for developers wanting to use the framework.

GEF is related to other Eclipse components as shown in Figure 12.1 [11].

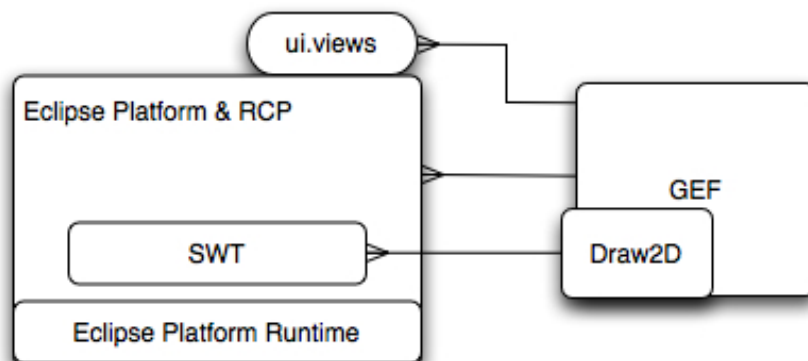


Figure 12.1: GEF Dependencies. Source: [11]

¹<http://www.eclipse.org/gef/>

²<http://www.eclipse.org/projects/index.html>

As shown in the figure GEF is composed of two main components. One is Draw2d which is used to draw the figures in the GEF framework. Draw2d has an efficient layout and rendering support, provides several figures and layout implementations including connection anchoring and routing and provides an overview window for viewing an outline of larger figures [9].

The other component is GEF itself. GEF provides simple tools for single and multiple selection, creation and connection which can be displayed on the included palette. It has a controller framework that is part of its implementation of the MVC pattern described in Section 12.1. GEF relies on *policies* that can be “installed” into different parts of the controller for translating interaction with the graphical view to changes in the model. The policies are part of the Chain of Responsibility pattern (see Section 12.3) and will do its part in deciding what to do when a user event occurs in the view. Undo and Redo are built-in tools that are realized with the Command pattern discussed in Section 12.2 and GEF’s CommandStack.

Patterns and Pattern Languages are ways to describe best practices, good designs, and capture experience in a way that it is possible for others to reuse this experience.

*Pattern Library - Hillside.net*³

According to Gamma et. al. [7] there are four essential elements to a pattern: its name; the problem that it tries to solve; the solution it suggest; and consequences and trade-offs that are results of using the pattern. [7] also describe a formal way of describing patterns and categorize them according to their scope or purpose.

According to [15] there are six patterns that are commonly encountered in GEF and it is the use of these patterns that gives this framework its flexibility. These patterns are presented in the following sections.

12.1 Model-View-Controller

The Model-View-Controller(MVC) pattern was conceived in 1978 [18]. It is a common way of organizing an application into layers of abstractions, with each layer responsible for its own part. The MVC pattern as it is used in GEF is shown in Figure 12.2.

According to [7] the Model of the MVC pattern is an application object. In the case of GEF, this can be any Java Object. The View is the screen representation of the model, as visible to the user. In GEF, the view is implemented by a set of figures implementing the IFigure interface. Finally, the Controller is an EditPart type [15], and is responsible for reacting to the user’s input. It will use its edit policies (see Section 12.3) together with the tools described in Section 12.4 and the commands described in the Section 12.2, for changing the model.

³<http://hillside.net/patterns>

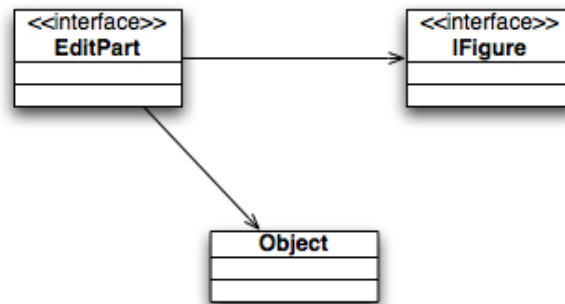


Figure 12.2: An overview of the MVC pattern in GEF.

By applying the Observer pattern [7] with the EditPart as a subject listening for changes to the model, the view can be updated with the changes to keep the view and model consistent. This pattern is outside the GEF architecture and will not be discussed here, but it can be noted that it is often used in GEF based applications.

12.2 Command

The main purpose of the Command pattern, which is a behavioral pattern, is to let “requests” be objects[7]. This enables logging, queuing and support for undoable operations, which can be problematic when modifying objects directly. The solution is according to Larmen [14] to let a task be a class that extends a common interface.

In GEF, the common interface is the abstract Command class. The most important methods are displayed in Figure 12.3. For an overview of all the methods with a better description and the default subclasses look in the GEF API [8].

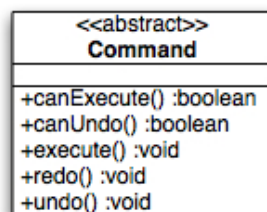


Figure 12.3: The most important methods in the GEF Command class.

The CommandStack class is an implementation of a command stack. It manages executing, undoing, and redoing of Commands, as well as keeping track of whether the data is saved or not.

12.3 Chain of Responsibility

In [3], Braude writes that the purpose of this behavioral pattern is to let a collection of objects provide functionality together rather than just a single object.

The consequences of the Chain of Responsibility pattern is that it reduces coupling, as an object does not need to know which other objects will handle a request [7]. It also adds flexibility in distributing responsibility to different objects. However, it is not guaranteed that an object that will handle a particular request exists.

In GEF the edit polices are responsible for handling the requests. The edit polices extends the abstract `AbstractEditPolicy` class or any of its subclasses. They are added to an `EditPart` which is the controller in the GEF MVC architecture described in Section 12.1. The installed edit polices may all respond to a request with `Commands` (see the previous section) which then are chained together to provide the collective functionality [15]. By using the edit policies instead of making an `EditPart` implement this behaviour itself increases code reusability and makes code management and readability easier [8].

Figure 12.4 shows the main parts of the `AbstractEditPolicy` and the interfaces it implements (for a full description, see the GEF API [8]). It implements the two interfaces `EditPolicy` and `RequestConstants`. The `EditPolicy` defines the method `getCommand(Request)`. A `Request` encapsulates the information needed by an `EditPart` to perform various functions [8]. By comparing the passed `Request` to the constants defined in the `RequestConstants` interface, an `EditPolicy` implementation will find the `Command` that is to be executed as a response to the user action.

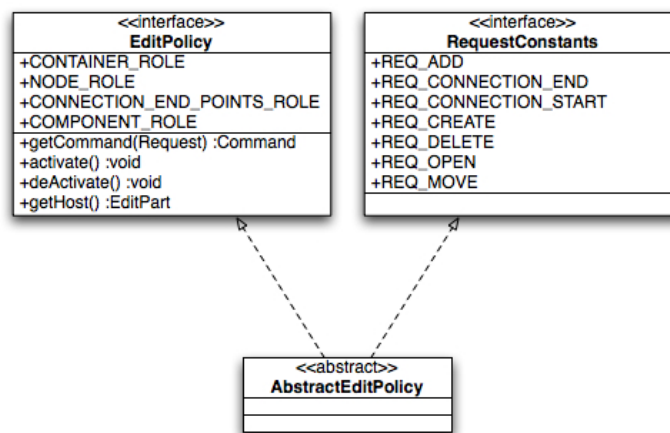


Figure 12.4: The `AbstractEditPolicy` class from GEF (not exhaustive).

12.4 State

The State pattern provides a state-based set of operations and allows an object to change behaviour when its state changes [12].

In GEF, the change is implemented by either switching tools in GEF editors, or by letting the tools (that implements the Tool interface) itself can have several states [15]. For example, a create tool causes the editor to behave differently to a mouse down event than it would when a select tool is active.

According to the GEF API [8], tools process low-level events which they turn into higher-level operations encapsulated as Requests. Figure 12.5 displays some of methods in the Tool interface. The whole description of the interface and that of the standard GEF tools which implements the interface, can be found in the GEF API [8].

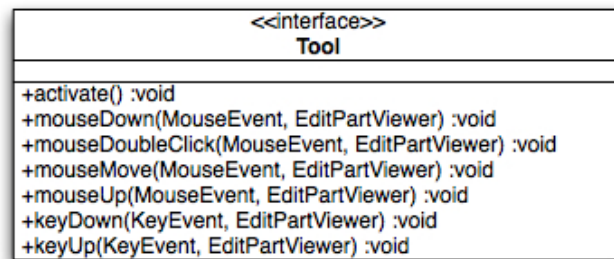


Figure 12.5: The Tool interface with its most important methods.

12.5 Abstract Factory

The Abstract Factory pattern is an creational pattern [7]. In [12], Hunt describes this pattern as used for creating families of related or dependent objects.

This pattern is applied when creating EditPart classes. The abstract factory is the EditPartFactory interface that has one method as shown in Figure 12.6. The createEditPart(EditPart context, Object model) method, will create an EditPart given the specified context and model.

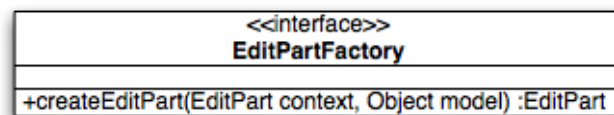


Figure 12.6: The EditPartFactory interface in GEF.

Another interface that acts as an abstract factory is the CreationFactory interface. The class CreationTool uses an CreationFactory to create new EditParts

when the GEF creation tool creates a new object in the model. The `CreationFactory` can be seen in Figure 12.7. Its two methods `getNewObject()` and `getObjectType()` returns a new object and the new object's type. A further description can be found in the GEF API [8].

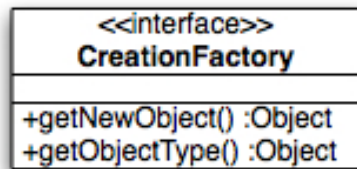


Figure 12.7: The `CreationFactory` interface in GEF.

12.6 Factory Method

Just as the Abstract Factory pattern, the Factory Method pattern is a creational pattern [7]. This pattern uses an abstract class with a method for creating an object, but it is up to the subclass to decide which class to instantiate.

In GEF, this pattern is applied to the installed edit policies that must implement one or more factory methods defined in an abstract superclass.

In [15], Majewski indicates that it is possible to use this method to explicitly create child edit parts without using the factory pattern described in the previous section.

Part III

Contribution

This part focuses on the development of AutAT. It starts out with an chapter describing the requirements. It goes on with design, implementation and testing before provide a basic user documentation for the system. The final chapter describes the user testing session.

Chapter 13

Requirements

Extreme Programming (XP) (described in detail in 7) is different from traditional software engineering processes. One of the great advantages of XP is its emphasis on customer involvement and short development cycles that results in early, concrete and continuous feedback according to [23]. This is an advantage as many customers does not know or sometimes does not understand what the software they are buying will do. Other reasons are the problem of communicating what they want and that the customer change their mind regarding what they want. There are probably numerous other reasons why requirement engineering is so difficult. Ultimately this means that spending lots of time gathering requirements and documenting it in great detail through the use of tools like UML use cases is a waste of time. Astels et. al. recommend against using use cases in [1]. XP uses user stories to capture user requirements which are described in Section 13.2.

This project is under the influence of XP which means that we are working close with our supervisors at BEKK as they can be considered to be our customers. The implementation is a proof of concept where we want to be able to adjust the program with respect to the users feedback. Because of this the requirements are a vision and a set of user stories that are described in the next sections. This has been more than sufficient to communicate the systems requirements.

13.1 Vision

In [1], Astel et. al. says that a system should start out with a vision for the system that should describe where the project should end up. This is related to this projects problem definition that is described in Chapter 2 and to the GQM goal in Chapter 6. Astel et. al. argue that the vision should be a less than 25 words statement about the purpose of creating the system. Our vision is:

The purpose of AutAT is to increase usability, quality and efficiency when writing acceptance tests for web applications, especially in the context of test driven development.

13.2 User Stories

A introduction to user stories is given in Section 7.2 on page 29. The project's users stories are:

US 1 *Register Project*

The user wants to create a new project. The user supplies a name and a description for the project. The project will contain acceptance tests for a single web application.

US 2 *Register User Story*

Within an existing project, the user will create a new user story.

US 3 *Nested User Stories*

User stories can be nested, providing more detailed stories or variations.

US 4 *Start Points*

A start point is a place where a test can start. It consists of a URL within the web application, and is given a logical name that the user can easily refer to. The user must be able to define and use such points.

US 5 *Create Test*

When a user story is defined, a test can be created and is added to that story. The test will illustrate one possible execution of the user story. This execution will involve navigating between web pages within the web application by clicking on links or by submitting a form, and viewing and validating the web pages' content.

US 6 *Edit Test*

The user can edit a test at a later stage.

US 7 *Extension Points*

Some tests may stop in an Extension Point. These tests are extendable, meaning that other tests may continue from one of this test's extension points. The extending test will perform the steps in the first test before continuing with their own steps. This is an alternative to starting a test at a Start Point as described in US 4.

US 8 *Aspects*

An aspect is a set of test steps that can be apply to many tests or parts of tests.

US 9 *Parameterized Tests*

Some tests should be executed several times, with different input values to the forms submitted and different text shown on pages. The user can provide these different sets of inputs and outputs and be able to map these to form elements and checks on pages.

US 10 *Share project*

Several people must be able to work on the same project with its user stories and tests. Every person working on the project need to see additions and changes performed by the others.

US 11 *Export Tests*

The user shall be able to export the tests in a project, in order to integrate these tests with Continuous Integration Tools such as CruiseControl¹ and AntHill².

US 12 *Run tests*

The system shall run all tests or tests for a single user story and give feedback to the user what parts of each test passed and which parts failed.

US 13 *Statistics*

The system can show and save statistics showing which part of tests passed and failed at a certain point in time.

US 14 *Check Tests with SiteMap*

The user can define pages and links between them that indicates if it should be possible to “go” between them. The program tests whether it is possible to go between all te pages in this “SiteMap” by using the tests and test data that is entered.

We believe that these requirements are sufficient for developing AutAT.

¹<http://cruisecontrol.sourceforge.net>

²<http://www.urbanocode.com/projects/anthill>

Chapter 14

Design

This chapter will describe the design of the AutAT application. The first section will describe a domain model for the system. The domain model is the result of further analysis of the user stories listed in 13.2. This model will act as the base for the design of the AutAT application.

The domain model will be mapped to architectural choices in Section 14.2.

The design in this chapter is targeted at US 1–8 and 10–11. US 9, 12, 13 and 14 have been delayed to later iterations in the development of AutAT. In accordance with the XP practices, the design does not target the “problems of tomorrow”.

14.1 Domain model

A *domain model* is a model of the problem domain used when developing a computer system. In our case, the domain is *tests for web applications*. The requirements for the system are shown as user stories in Chapter 13. These requirements have led to the top-level model shown in Figure 14.1. This model shows the concepts of the domain, for readability the properties of each concept is not shown.

In the model, we see the concept of a *Project*. This is a direct mapping from US 1. A set of *User Stories* is associated with a project. These are the stories the web application is supposed to support, as described in Section 7.2. US 2 deals with the relationship between a project and its user stories, while US 3 states that user stories can be nested.

A *Test* is one possible execution of a user story. As there are many possibilities with regards to such executions, there will most often be necessary to have more than one test for a user story. As stated in US 5, a test will consist of a series of steps. As each step in an interaction with a web application involves a web page, steps are shown as *Pages* in the model. A page will consist of a collection of *Elements*. An element is what a user can see on a web page, such as a text,

a link or a form. A test can check if these elements are present and correct or if some element is missing.

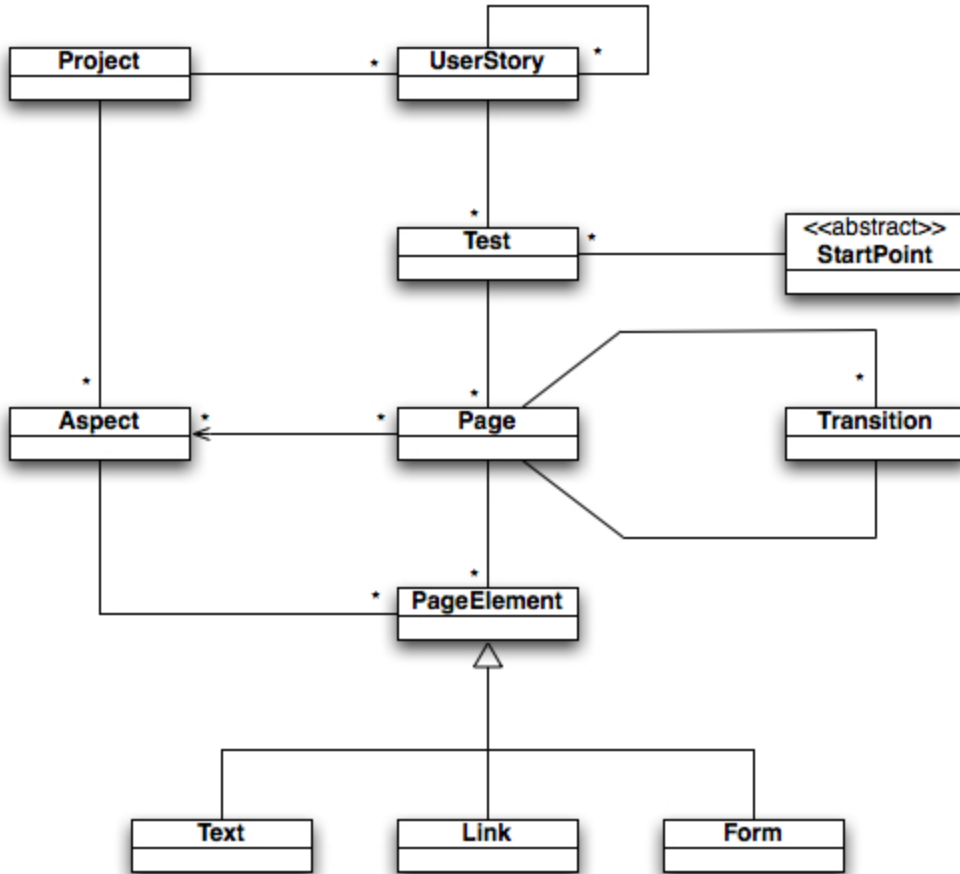


Figure 14.1: Top-level domain model

Of the elements on a page, links and texts are the simple ones. A *Form* is a more complex entity, consisting of several sub-elements as shown in Figure 14.2. A form on a web page will consist of one or more *Form Elements*. A form element can be a single-line text field, a multi-line text area, a password field, a select (drop-down) list, a set of radio buttons and (submit-)buttons.

An *Aspect* as described in US 8 will perform the same checks a page can perform. An aspect will consist of a set of Elements, just as for a page. The purpose of an aspect is to group together repeating checks such as checking a copyright statement, a login form or logout link. An aspect will add its checks to the checks already present in any page it is connected to. However, an aspect will not be able to be part of a transition as described next.

As a user interaction with a web application usually consists of more than looking at one page, the user will also navigate between pages. This navigation is shown in Figure 14.1 as a *Transition* connecting two pages. A transition

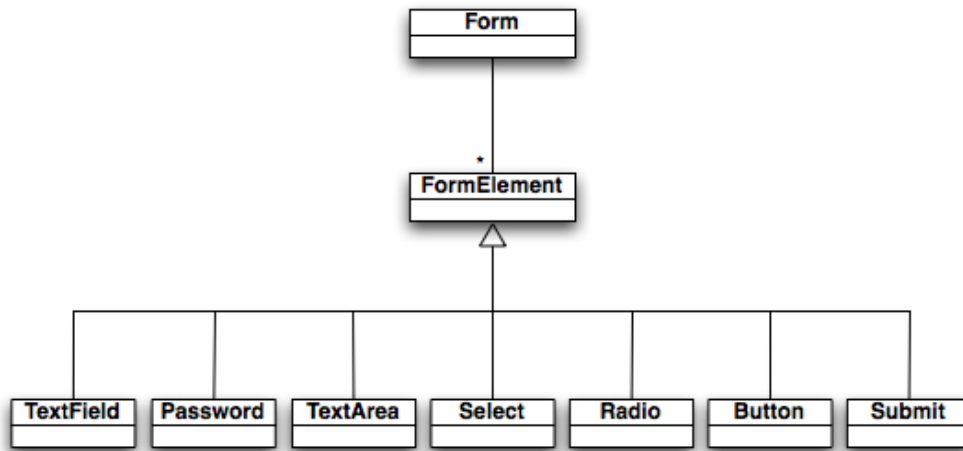


Figure 14.2: The form and form elements.

will start at one page, and stop at the next, and is executed either by clicking on a link or entering values to a form and then pressing a button. These two cases are shown in more detail in Figure 14.3. Clicking a link will result in a *LinkTransition*, a simple entity.

The more complex case of submitting a form with values requires a set of values connected to the fields in the form. A *FormTransition* is connected to the *FormElement* from Figure 14.2. The *FormTransition* also holds *FormInput* values to enter as input before submitting the form.

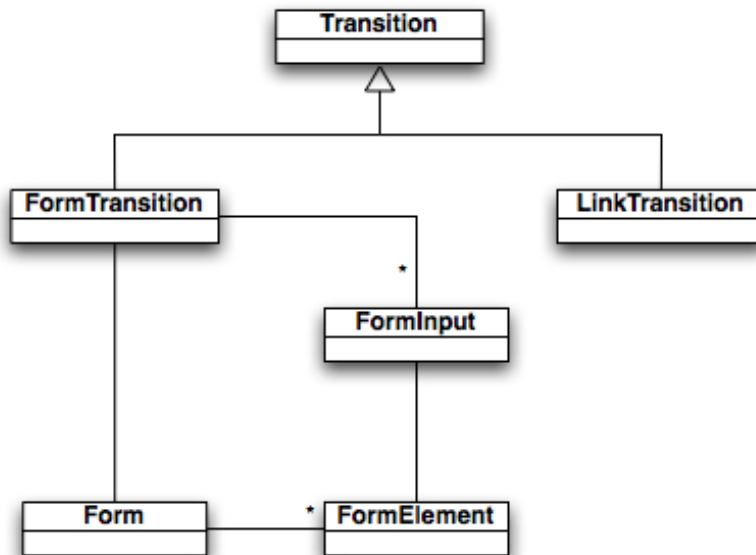


Figure 14.3: Transitions

US 4 and 7 deal with how a test will start. US 4 will let a test start at a specific

URL within the tested web application. US 7 will let a test continue where another test stopped. This can let the first test do some basic set up, such as performing login, before the second test does its business. This results in the concept of a *StartPoint* shown in Figure 14.1. In a conceptual manner, a Start Point can be thought of as shown in Figure 14.4. Here all tests will start at a start point. This start point will be either a simple URL or it will be an extendable test.

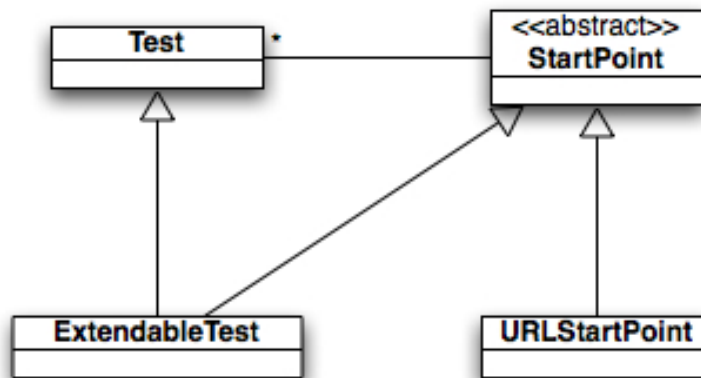


Figure 14.4: StartPoint

14.2 Architecture

The domain model described in the previous section is a more detailed view of the first eight user stories. These are the ones dealing directly with how a user will create and manipulate projects and their respective user stories and tests. This model, as well as the rest of the user stories in Section 13.2, are the base of the architecture for AutAT.

The choice to implement AutAT as a plugin for Eclipse was described in Chapter 10. This choice has certain implications for the overall architecture of AutAT. First of all, it has to integrate into Eclipse and the metaphors used in the Eclipse Platform. The overall architecture of Eclipse and the concepts of a Workspace and the Workbench are described in Chapter 11. Secondly, some of the user stories for AutAT will be supported by other plugins that AutAT will have to relate to.

As we can see from Figure 14.5, AutAT will become a component within an Eclipse installation. It will be depending on the Eclipse runtime to provide access to the local filesystem, AutAT's use of the local filesystem will be described in Section 14.2.1. The use of the Team API to provide access to a remote file system will also be described in the same section.

The internal design of the AutAT plugin will be described in more detail in Section 14.2.2.

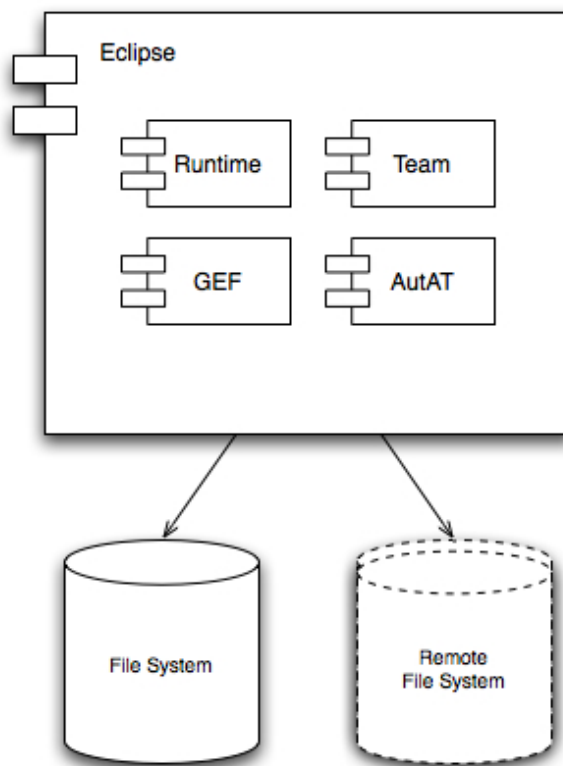


Figure 14.5: The AutAT plugin and dependencies in Eclipse

14.2.1 Using the File System

The Eclipse Workbench consists of a collection of projects, as described in Section 11.1.1. The domain model for AutAT also contains the concept of a project. The AutAT plugin will use the built-in project hierarchy in Eclipse to organize the AutAT projects. As normal Eclipse projects consist of a set of folders and files, so will an AutAT project.

As we see in the domain model, a project consists of several user stories which may contain nested user stories. In the AutAT eclipse project structure, a file system folder will act as a metaphor for a user story. By giving the folders the names corresponding to the user stories for the system, this will provide the link to the document containing the stories.

As for the tests themselves, each test will be stored in a separate file within the folder representing its user story. A folder may contain several tests. The files will be based on XML for saving the tests. XML is relatively easy to use for storing structured data, as is the case from our domain model. We choose the file-extension `.aat` for the files containing an automatic acceptance test.

User story 4 define start points as a combination of a name and a URL. These pairs must be collected in a central place in order for AutAT to easily handle them. For this purpose, we will introduce a new file specific to an AutAT project. As with the tests this file will be based on XML, and it will be placed in the root folder of the project. We call this file `NamesAndURLs.urls`. By giving this file a static name, AutAT will know where to find it, and each test will only need to keep a reference to the start point it is using.

Figure 14.6 shows how we want an example AutAT project to look for a project named “CDDDB”. The project base folder contains two folders in addition to the file containing start points. The folder `tests` is introduced to contain the user stories and tests for the system, while the folder `generated` is to contain exported tests, as specified in US 11. The project contains three user stories and five tests.

```
CDDDB
|-tests
  |-Front Page
    |--testFrontPage.aat
  |-Register New Artist
    |--testRegisterPinkFloyd.aat
    |--testRegisterToriAmos.aat
    |--testInvalidRegistration.aat
  |-Register new Album
    |--test Register new Album for Rolling Stones.aat
|-generated
|--NamesAndURLs.urls
```

Figure 14.6: Example AutAT project hierarchy

US 10 specifies the need to be able to share a project between several people working on the project. This is where the Eclipse Team plugin comes to play. This plugin allows for the use of centralized repositories providing configuration management, file sharing and version control as described in Section 11.3. This allows us to thus satisfy US 10. Different teams may choose to use different repositories, as long as the technology is supplied through a plugin.

14.2.2 AutAT Internals

The actual implementation of the AutAT plugin will be divided into four high-level packages as shown in Figure 14.7. The four packages is the *AutAT core*, *AutAT common* classes, the *AutAT User Interface* and a set of *exporters*. These packages will be described in the following sections.

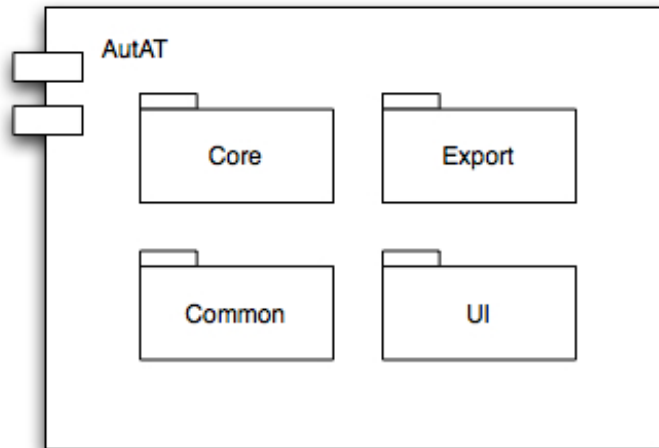


Figure 14.7: The internal structure of the AutAT plugin

AutAT Common

The AutAT Common package will contain the common classes used by the other packages. This includes the class model derived from the domain model, that will be used by all other packages. These classes are the same as in the domain model, but have added properties such as names, identifiers, descriptions, texts and such.

AutAT Core

The AutAT Core is responsible for handling the tests in a project. These responsibilities include reading tests from files, converting them from XML to objects as defined in the AutAT common package, and performing the reverse

operation to save them to files. The conversion also includes linking the start points into the tests.

This core functionality will be used by the other parts, exporters and the UI.

AutAT Exporters

An Exporter will be a component that will transform a test from its object representation into some form that can be executed. The idea is to let several exporters co-exist, providing differing executable tests. The first implementation will be an exporter that will use the jWebUnit WebFixture extension to FIT, providing HTML files that FIT can process.

AutAT UI

For the users of AutAT the User Interface is probably the most important part. As we see from Figure 14.8, the UI package consists of four sub-packages that will provide its part of the functionality for the user.

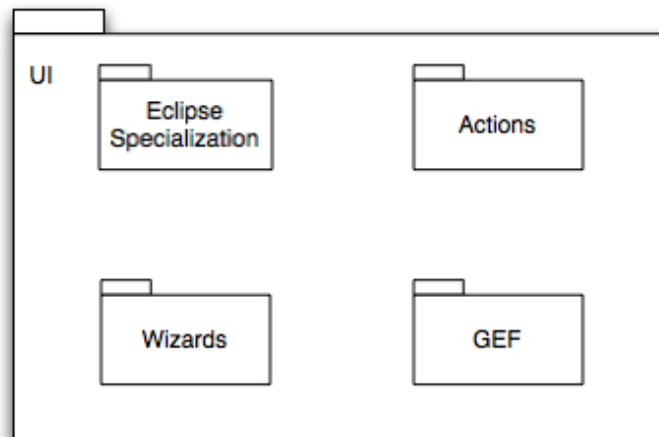


Figure 14.8: The structure of the UI package

First we have the *Eclipse Specialization* package. This package will contain *property pages* for each project and *preference pages* for the AutAT plugin. These will be used to customize the behaviour of a project and for the AutAT plugin respectively.

Secondly a set of *Actions* will be needed. An action is a process that will be triggered from some point in the UI. Actions include exporting tests and running tests. Again, these will plug into existing extension points provided by the Eclipse platform.

A *Wizard* is a dialog window guiding the user through a series of steps when performing a complex task. AutAT will need wizards to create new projects and

to create new tests. The wizards will perform the setup necessary in order to get started. When creating a new project required folders and files will be created, when creating a new test the least possible information that can constitute a test will be created.

By far the largest part of the UI is the *GEF* package. This package will contain the graphical test editor. As this is such a important part, its design will be discussed more closely.

AutAT UI GEF

The structure of the GEF package is closely linked to the patterns described in Chapter 12. However, not all the packages within GEF contribute to the patterns.

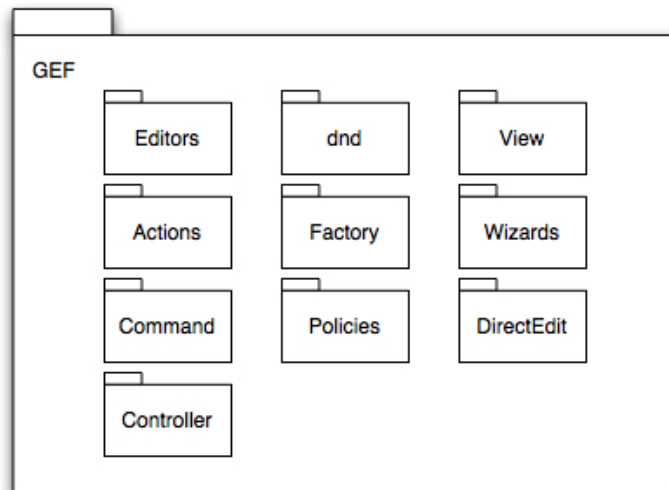


Figure 14.9: The structure of the GEF package

The *Editor* package contains the implementation of the editors that are created for AutAT. There are two editors: the test editor and the an editor for viewing and manipulating the set of start points. The start point editor will not use the GEF patterns, just handle a file in its own way by showing a list of the defined start points.

The test editor on the other hand, will use all the GEF related patterns. The editor will be started when a user opens a test file (**.aat*). This will load the model into memory (see the MVC pattern in Section 12.1) and create controllers from the edit parts in the *controller* package corresponding to the test. It will also create the visual representation of the test using parts from the *view* package.

When creating controllers, the Abstract Factory pattern from Section 12.5 is used. A Factory for edit parts will be provided in the *factory* package. A Factory for creating new elements in a test will also be provided.

The *Policies* package contains the Edit Policies for tests. These are part of the Chain of Responsibility pattern (Section 12.3). As the user interacts with the system, the policies will figure out which command from the *Command* package to execute. These commands will implement the Command pattern from Section 12.2. Some commands will start a *Wizard* that asks the user for information to complete. This information can be the name of a link when navigating, or values to submit in a form.

The *Actions* package is to contain executable actions that are not used by the edit policies. This includes normal undo and redo functionality and contextual (right-click) menus in the editor.

The packages *dnd* and *DirectEdit* provide basic additions to the view. *dnd* provides drag-and-drop functionality, while *DirectEdit* is to give the user the ability to edit text (such as page title, the text on a link) directly in the view by simply clicking on the text to edit.

This division of the different parts of the editor into packages with their own specific responsibilities and mappings to widely accepted patterns will keep the design as simple as possible. It will also make future additions easier due to the modularity of the design.

Chapter 15

Implementation

This chapter will describe the implementation of the AutAT Eclipse plugin. We provide one section on each of the main parts of the application. First we describe the Common package in Section 15.1, before we show the internals of the Core package in Section 15.2. The Exporter package contains functionality to create WebFixture test for FIT and is described in Section 15.3. The GUI implementation is described in Section 15.4, while the final section of the chapter shows key metrics related to the implementation of AutAT, such as a class-count and lines-of-code.

15.1 AutAT Common

The AutAT Common package contains the “value-objects” in the application. It is based on the domain model described in Section 14.1, but has some modifications. First of all, the Project and User Story concepts are not a part of this object model. Secondly, all the classes have added properties such as names and identifiers as well as get- and set-operations to manipulate these properties. For the central concepts, this is shown in Figure 15.1.

Properties have been added to complete the form elements, the transitions and input values. In order to keep this chapter simple, these are not shown here.

One deviation from the domain model is the *Aspects*. In the original domain model, an aspect is owned by a project and can be joined to any number of pages within any number of tests. This has not been a priority when implementing AutAT. However, we wanted to show the concept of an aspect and how it can look and work, therefore aspects was moved into a test. Now an aspect is owned by a test, and can be joined to any pages within this test.

The design in 14.1 included extendable tests. This has been left out of the initial implementation due to time pressure.

The Common package will be used by the other packages. The Core package will populate the objects with values based files in the file system, the UI package

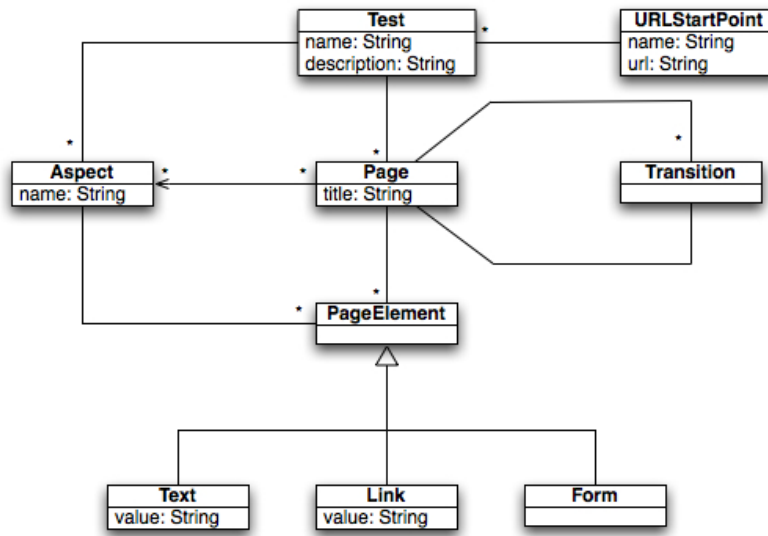


Figure 15.1: Central classes in the Common package (identifiers not shown)

will show graphical editors representing the Common model and the exported tests will be based on the values from a populated object model.

15.2 AutAT Core

The AutAT Core is responsible for handling tests and startpoints, reading these from files and saving updates to files. As described in Section 14.2.1 AutAT defines two file formats, one for saving tests and one for saving start points. Both these formats will use XML as internal representation of the data they hold.

The Java programming language has built in libraries for handling XML. However, these seem harder to use than the libraries provided by the *JDOM* project¹. The JDOM libraries provide an easier abstraction for the elements in the XML document, and navigation and general handling of these elements is easier than with the standard Java libraries. Thus, JDOM was chosen to handle the documents.

The AutAT Core package is implemented using the *Builder* pattern described in [7]. This pattern divides the effort of constructing a complex object (i.e a Test) between several objects, each responsible for a smaller part of the total. In the case of AutAT, one class (TestConverter) will be responsible for reading and saving a test. This will in turn use new classes (i.e. PageConverter) to handle the different parts that constitute a Test. An illustration of how this works is shown in Listings 15.1 and 15.2. The first listing converts a test from XML to an object representation, while the second does the inverse operation.

¹<http://jdom.org>

Listing 15.1: Reading a Test from XML (from TestConverter.java)

```
1 public static Test fromElement(Element testElement, URLList urlList
2 ) {
3     Test test = new Test();
4
5     Attribute idAtt = testElement.getAttribute("id");
6     test.setId(idAtt.getValue());
7
8     Attribute nameAtt = testElement.getAttribute("name");
9     test.setName(nameAtt.getValue());
10
11    Element desc = testElement.getChild("description");
12    test.setDescription(desc.getValue());
13
14    Element startPoint = testElement.getChild("connectionPoint");
15    ConnectionPoint connPoint = ConnectionPointConverter.
16        fromConnectionPoint(startPoint, urlList);
17    test.setStartPoint(connPoint);
18
19    Element pagesElement = testElement.getChild("pages");
20    List pages = PageConverter.fromPagesElement(pagesElement);
21    test.setPages(pages);
22    ...
23    ...
24    return test;
25 }
```

Listing 15.2: Saving a Test to XML (from TestConverter.java)

```
1 public static Element fromTest(Test test) {
2     Element retVal = new Element("test");
3     retVal.setAttribute("id", test.getId());
4     retVal.setAttribute("name", test.getName());
5
6     Element desc = new Element("description");
7     desc.setText(test.getDescription());
8     retVal.addContent(desc);
9
10    Element startPoint = ConnectionPointConverter.
11        fromConnectionPoint(test.getStartPoint());
12    retVal.addContent(startPoint);
13
14    Element pages = PageConverter.fromPages(test.getPages());
15    retVal.addContent(pages);
16    ...
17    ...
18    return retVal;
19 }
```

15.2.1 XML Schemas

An XML schema² is a XML-based description of the structure of a XML document. The schema will define:

²<http://www.w3.org/XML/Schema>

- which elements are permitted to appear in the document
- which attributes defines an element
- the data types for each element and attribute, as well as default values and whether values are required or optional
- relationships between elements, which elements are sub-elements of others and the ordering of such sub-elements

The AutAT plugin has two file formats to handle, one for tests, the other for start points. For each of these, a separate XML schema has been defined. Listing 15.3 shows the schema for the start points, while Listing 15.4 shows a part of the schema for a test. The complete schema for a test is found in Appendix F.

Listing 15.3: XML schema for start points

```
1 <?xml version="1.0"?>
2 <xs:schema
3     xmlns:xs="http://www.w3.org/2001/XMLSchema"
4     targetNamespace="http://autat.sourceforge.net"
5     xmlns="http://autat.sourceforge.net"
6     elementFormDefault="qualified">
7
8     <!-- type for the mapping elements -->
9     <xs:complexType name="mappingType">
10        <xs:attribute name="id" type="xs:string" use="required" />
11        <xs:attribute name="name" type="xs:string" use="required" />
12        <xs:attribute name="url" type="xs:string" use="required" />
13    </xs:complexType>
14
15    <!-- type for the collection of mapping elements -->
16    <xs:element name="urlMappings" >
17        <xs:complexType>
18            <xs:sequence>
19                <xs:element name="mapping" type="mappingType" minOccurs
20                    ="0" maxOccurs="unbounded" />
21            </xs:sequence>
22        </xs:complexType>
23    </xs:element>
24 </xs:schema>
```

Listing 15.4: XML schema for tests

```
1 <?xml version="1.0" ?>
2 <xs:schema
3   xmlns:xs="http://www.w3.org/2001/XMLSchema"
4   targetNamespace="http://autat.sourceforge.net"
5   xmlns="http://autat.sourceforge.net"
6   elementFormDefault="qualified">
7   ...
8   ...
9   <!-- the test type, base element in the test documents -->
10  <xs:element name="test">
11    <xs:complexType>
12      <xs:sequence>
13        <xs:element name="description" type="xs:string" />
14        <xs:element name="connectionPoint" type="
15          connectionPointType" />
16        <xs:element name="pages" type="pagesType" />
17        <xs:element name="aspects" type="aspectsType" />
18        <xs:element name="transitions" type="
19          transitionsType" />
20      </xs:sequence>
21      <xs:attribute name="id" type="xs:string" use="required"
22        />
23      <xs:attribute name="name" type="xs:string" />
24    </xs:complexType>
25  </xs:element>
26 </xs:schema>
```

15.3 AutAT Exporter

The Exporter package contains the functionality used to transform the AutAT Common representation of a test into a representation that is suitable for some testing framework to execute. As one of the user stories (US 11) specifies the need to be able to change testing framework easily, we have created a simple abstract class to extend in order to export tests. This class performs the traversal of the project, telling the implementor to convert one input file and save it to an output file. The signature of the abstract class `DirectoryWalker` is shown in Listing 15.5, and the initial implementation of the converter method for `WebFixture` is shown in Listing 15.6. Parts of the classes are omitted for readability.

Listing 15.5: The abstract class DirectoryWalker

```

1  public abstract class DirectoryWalker implements
    IRunnableWithProgress {
2      protected URLList urlList;
3      protected String baseUrl;
4
5      public DirectoryWalker(IProject project) {
6  ...
7      }
8
9      public boolean walkProject(IProject project) {
10 ...
11     }
12
13     /**
14      * Converts a single file.
15      *
16      * @param inFile The file containing the test.
17      * @param outFile The file to contain the result.
18      * @return true if successful, false
19             otherwise.
20     */
21     public abstract boolean convertFile(File inFile, File outFile);
22
23     public void run(IProgressMonitor monitor) throws
24         InvocationTargetException, InterruptedException {
25         progMonitor = monitor;
26         walkProject(theProject);
27     }
28 }

```

Listing 15.6: Implementation of test conversion for WebFixture

```

1  public boolean convertFile(File inFile, File outFile) {
2      Test test = ToFromXML.fromXML(inFile, urlList);
3
4      ToFromFIT.toFIT(test, outFile, baseUrl);
5
6      return true;
7  }

```

The implementation of the WebFixture converter `ToFromFIT` is done in the same way as the AutAT Core, using the builder pattern and JDOM for creating HTML files. We will not discuss this implementation further here.

15.4 AutAT UI

We will not dive into the inner workings of the AutAT user interface more than described in the design chapter. We feel this does not give much more value to our description of the implementation. However, we will show the results as perceived by the user.

The most central part in the AutAT user interface is the graphical test editor. This is the editor a user will interact with to create, edit and view a test. An example of the Eclipse workbench with an AutAT test is shown in Figure 15.2. This shows how a test appear to the user. We will explain its parts here.

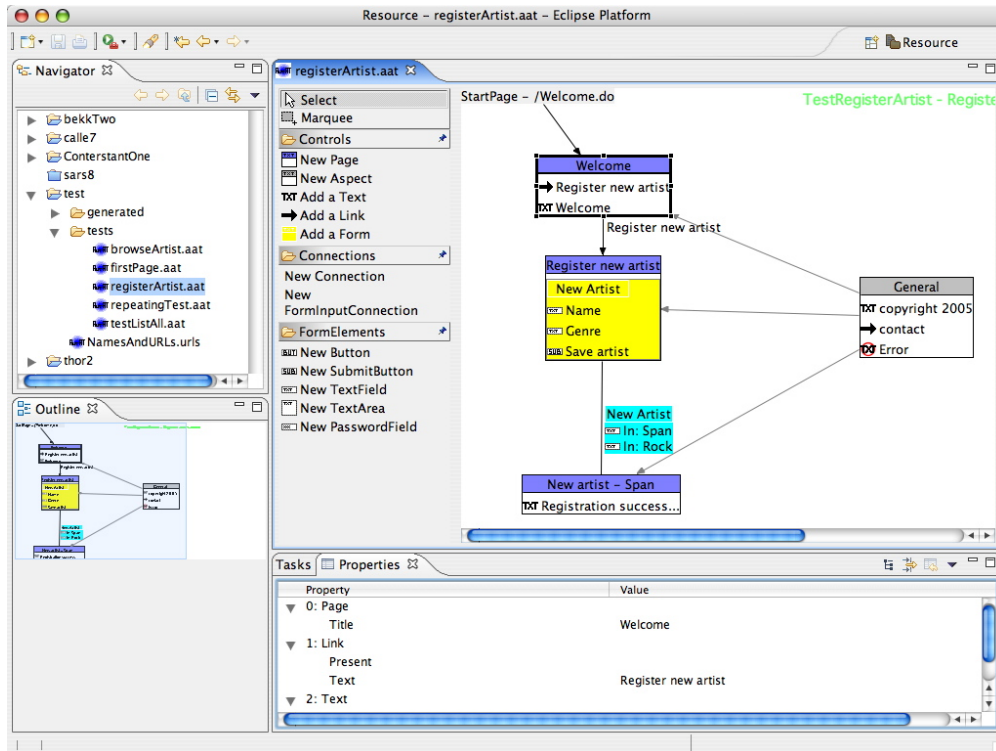


Figure 15.2: A full test shown with the AutAT plugin

The main window is the **editor window** itself. This consists of two parts: the left is a *palette* that contains the set of elements the user can add to the *work area*. The user first select the type of element she wants to add to the test (i.e. a Page, an Aspect or a Transition), then the mouse cursor will change to reflect her choice, and when she clicks in the work area, the selected element is added.

When a new test is opened, it will only contain the start point the user selected when creating the test. As the user adds the first page, a transition (an arrow) will be created, connecting the two. Pages are shown with a light blue header. The user can continue to add elements to her test, checks for text or links on pages or new pages. A form is represented by a yellow area inside a page. When a user has added a form to a page, she can add form elements (such as textfields and buttons) to this form. An aspect is also shown. This is similar to a page, but has a gray header. The aspect is connected to pages by adding a connection, just like when navigating between pages using links.

To navigate between two pages in a test, the user adds a connection between them. This will trigger a dialog box: if the user selected to navigate by pressing

a link, she must provide the text of the link to click; if she selected to submit a form, she must select which form to use and enter values into the fields it has defined.

Figure 15.2 also shows two other AutAT components. In the bottom left corner, an **outline** of the currently active test is shown. This is a down-scaled version of the test, useful when the test becomes visually larger than the available screen area. Below the editor window, the **property editor** is used. This is an alternative to edit text directly in the editor window. This will show the values of the currently selected element in the editor window, and the user can manipulate the values directly.

Figure 15.3 shows the two pages of the wizard that guides the user through the process of creating a new test. The first page (top) makes the user choose a location to place the new test (a user story within the project), provide a filename and a regular name for the test. He is also asked to give a description of the test. The second page makes the user choose a start point where he wants the test to start. These are the start points defined in the file `NamesAndURLs.urls` with the start point editor (described shortly). The second page also lets the user define a new start point. This will invoke a new dialog window for him to enter the details of the start point.

The wizard for creating a new AutAT project is similar to the one for creating a new test. It ask the user for information on where to store the project, its name and the URL of the web application it will test. This wizard is not shown here.

Every test will start at some start point within the web application. These starting points are kept in the file `NamesAndURLs.urls` in the project root folder. The editor shown in Figure 15.4 is used to edit the starting points. The editor consists of a table showing the defined start points (their names and urls) that can be edited at will. The *add* button will add a new start point to this list.

15.5 AutAT Software Metrics

Table 15.1 shows the key metrics for the AutAT application.

Metric	Count
Number of packages	22
Number of classes	180
Lines of code	3877
Average lines of code per class	21.5
Maximum lines in a class (<code>autat.ui.gef.editors.GraphicalTestEditor</code>)	137

Table 15.1: Key software metrics

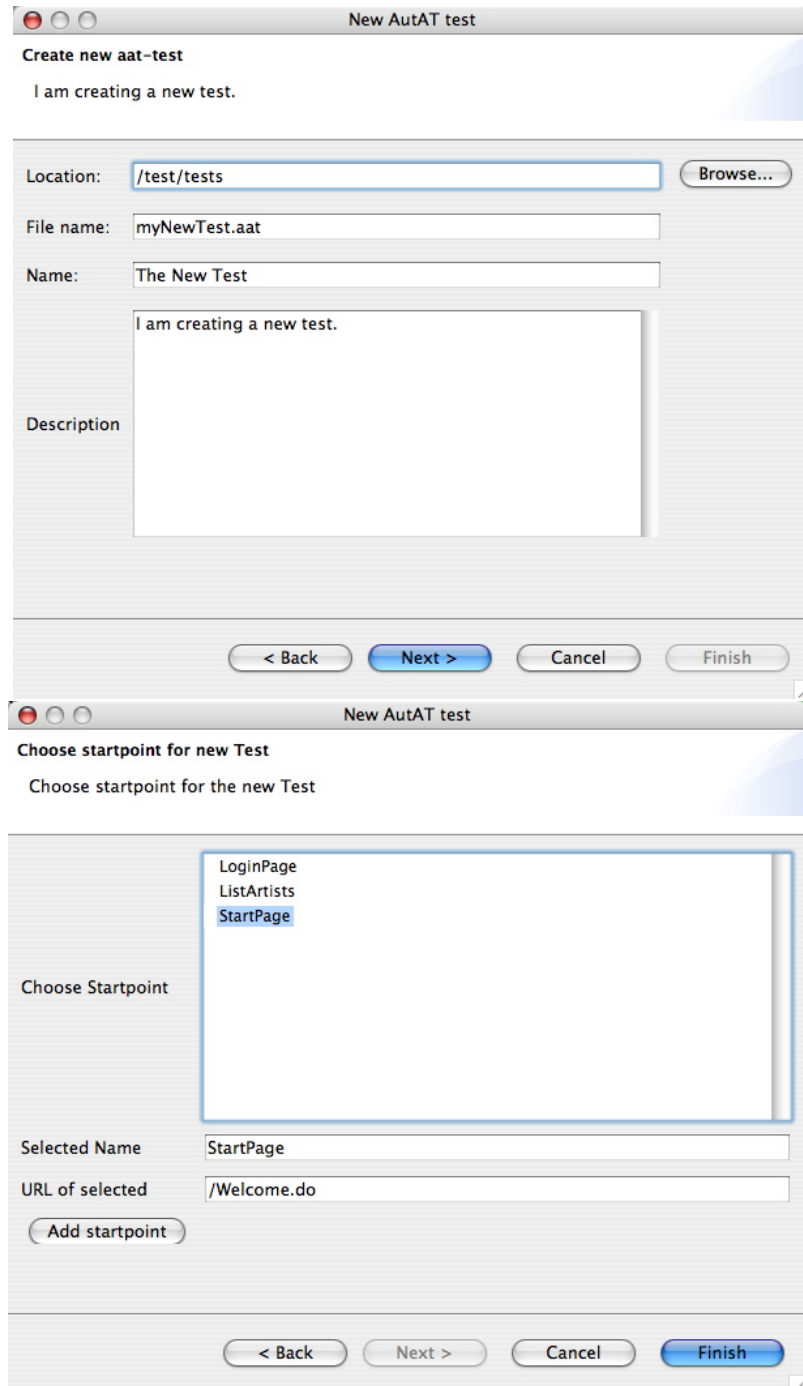


Figure 15.3: The wizard when creating a new test

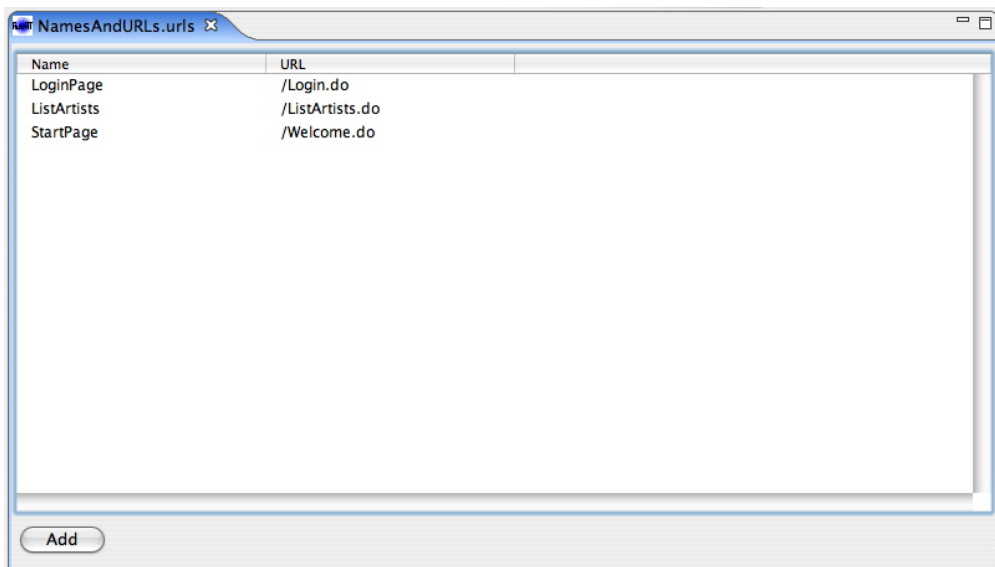


Figure 15.4: The start point editor

Chapter 16

Testing

We have written a lot of nice things on XP and Test Driven Development earlier. Some would say that doing manual tests in XP is like doing witchcraft in a church. What some consider even worse (we would not even think of an analogy) is doing “ad hoc manual testing” in XP. However, for some parts of the AutAT application this is what we have done.

The nature of AutAT as a proof of concept is to explore the possibilities of how we can visualize and maintain acceptance tests for web applications. To write detailed acceptance tests for an exploratory effort only seemed like a way to limit the possible outcomes, and we figured there was no way we could tell in advance what was “right or wrong”, but we could evaluate different solutions as to which is “better or worse” on the way. Also, many within the XP community find GUI testing hard to automate. They recommend writing the GUI of an application as thin as possible, focus automatic testing on the layers beneath the GUI and use manual tests for the actual GUI [5]. For these reasons, there were not specified formal tests for the GUI of AutAT.

To say that no formal tests were specified, is not the same as saying no testing did occur. We constantly evaluated the solutions we were building to find ways to improve them. This process started early with the first paper prototype that was evaluated. The feedback from this session led to a few design changes to the first computerized release. The implementation of the AutAT GUI was a constant evaluation of how the layout should be and how users should interact with the system. This constant evaluation generated small, incremental changes that in sum improved the finished product.

The parts of AutAT below the GUI were developed in a more formal fashion. A modified version of TDD was used. Unit tests and the classes they tested were developed side-by-side. The Core classes, the classes in the Common package and unit tests for them were developed in parallel. When a property was added to a common class, tests were also modified in order to reflect the new reality. The WebFixture exporter was developed in the same fashion. As changes in any of the classes occurred, they would make some unit test fail and either the

test or the class were corrected so that the test passed again.

The user testing session and the empirical study of the results from this act as a form of acceptance test for the project. The results reflect how we have achieved the vision of increasing the efficiency and usability when writing tests and increasing the quality of the tests.

Some might say this is a short description of the testing process in a project that promotes testing. However, as we have argued above the project is mainly a proof of concept, trying to explore future possibilities. We feel that the testing we performed was enough for our purposes. As argued by Hutcheson in [13] testing software is an effort balancing the costs of the testing with the amount of testing needed.

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*.

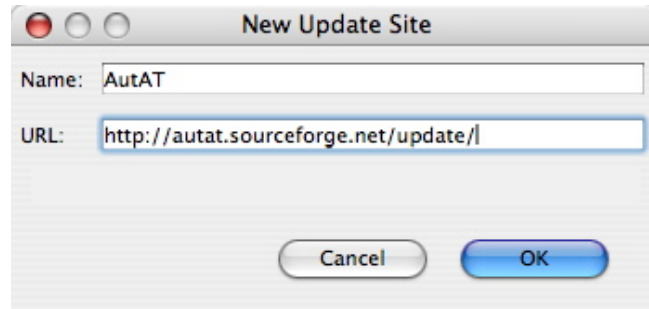


Figure 17.1: Installing a new New Update Site

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

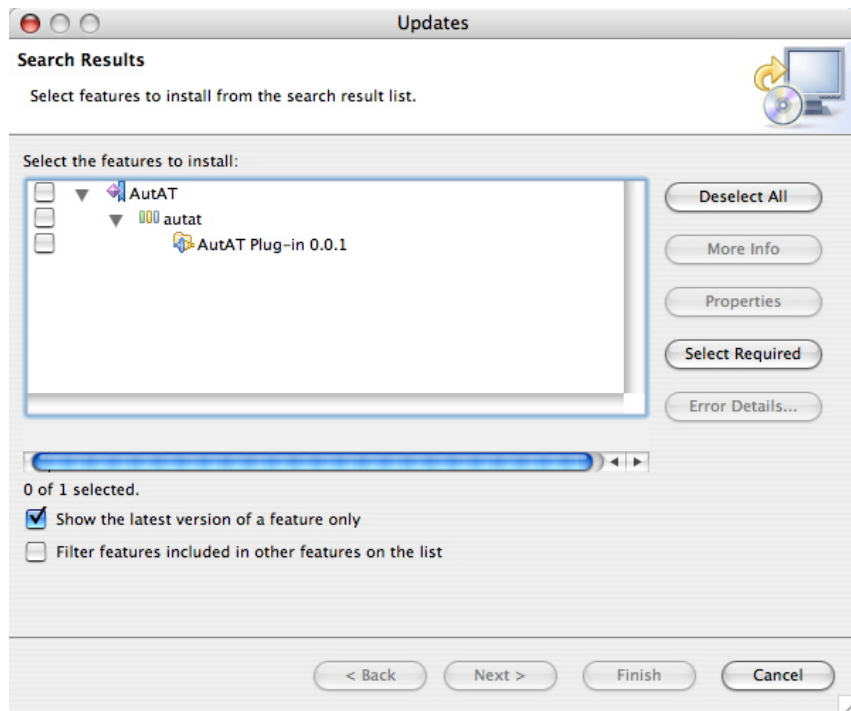


Figure 17.2: Select AutAT

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.

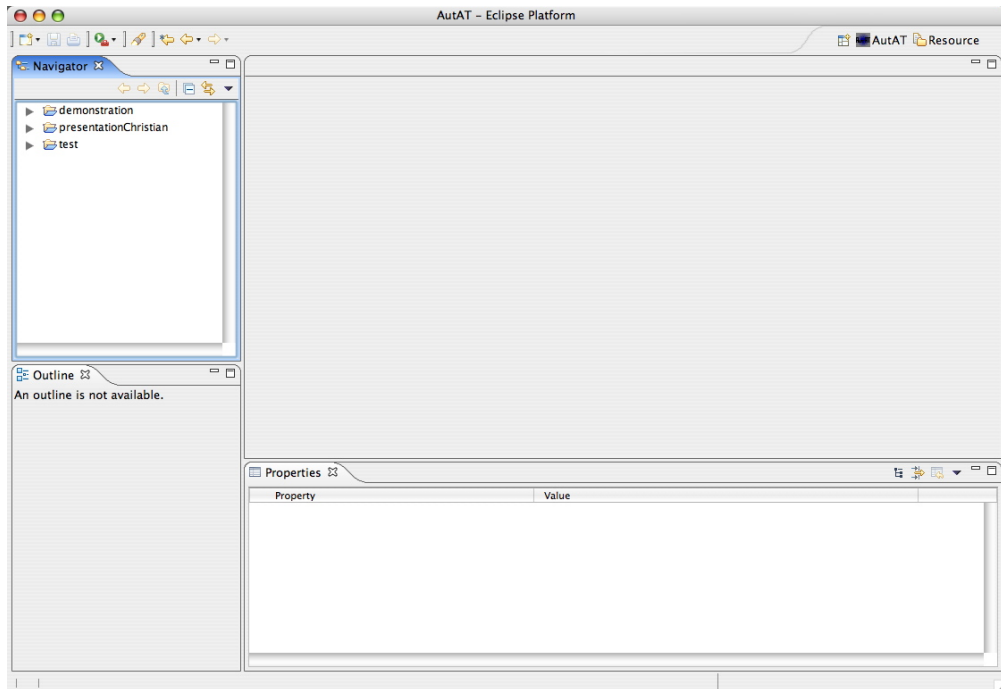


Figure 17.3: The AutAT Perspective

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.

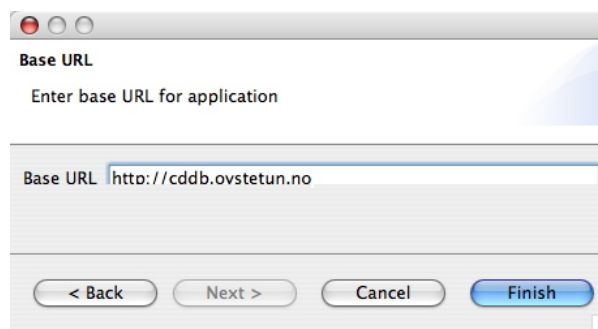


Figure 17.4: Provide a Base URL

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.

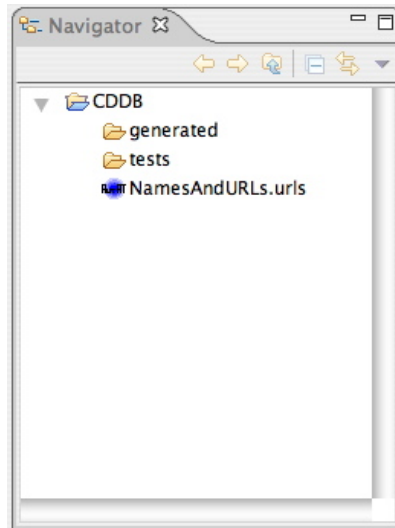


Figure 17.5: The AutAT Navigator

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* >.

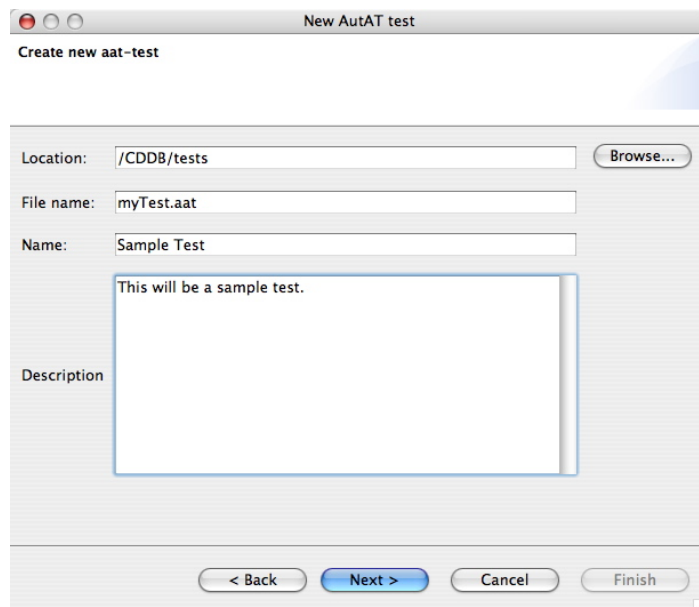


Figure 17.6: Create a new test

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.

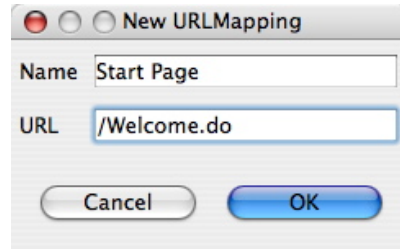


Figure 17.7: Create a new Start Point

An empty test will appear in the Editor window, as shown in Figure 17.8. We will add pages, checks and transitions to complete the test. To add an element to the test, select the appropriate field in the *palette* to the left in the editor window.

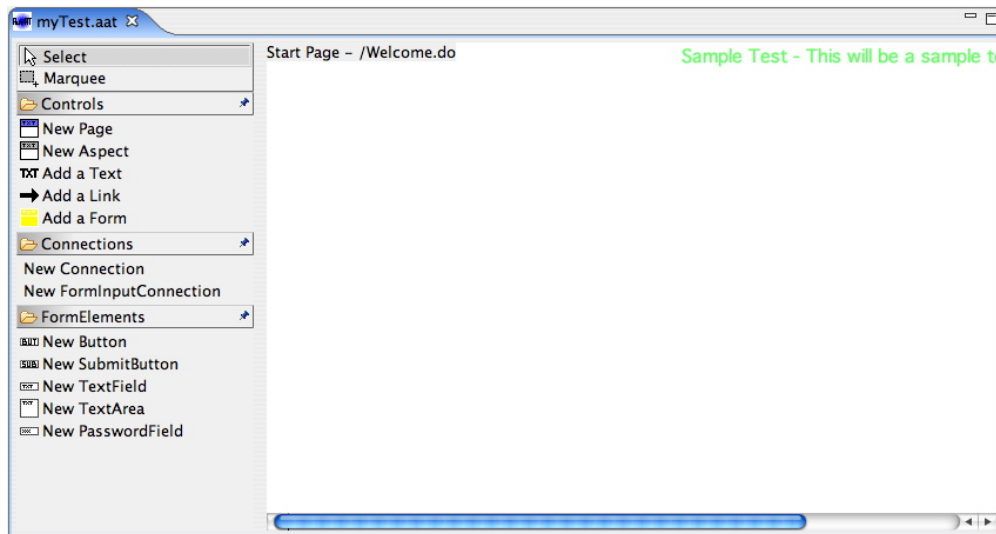


Figure 17.8: An empty test

Start by adding a new page to the test. Type the title “CDDB”. An arrow connecting the start point and this page will appear in the test. Add a text “Welcome to the CDDB” and two links with the texts “List all artists” and “Register new artist” to the page. Create a new page for registering a new artist, with the title “Register new artist”. Connect the first page to the second with a new *Connection*. Add a form to the second page, containing two textfields (“Name” and “Genre”) and a submit button with the text “Save artist”.

The last page will be a page confirming we added a new artist. Create a page with the title “Artist registered: Pink Floyd”. Connect this to the second page with a *FormInputConnection*. You will be prompted to select the form to fill, as shown in Figure 17.9. Type the values as shown in the figure.

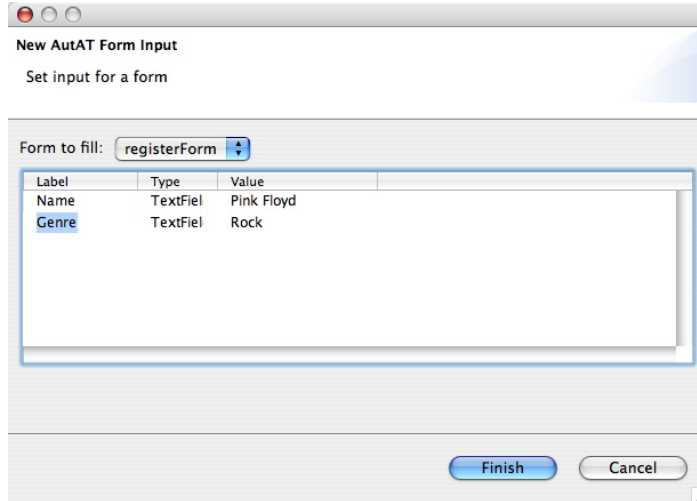


Figure 17.9: Provide input values to a form

The final result is shown in Figure 17.10.

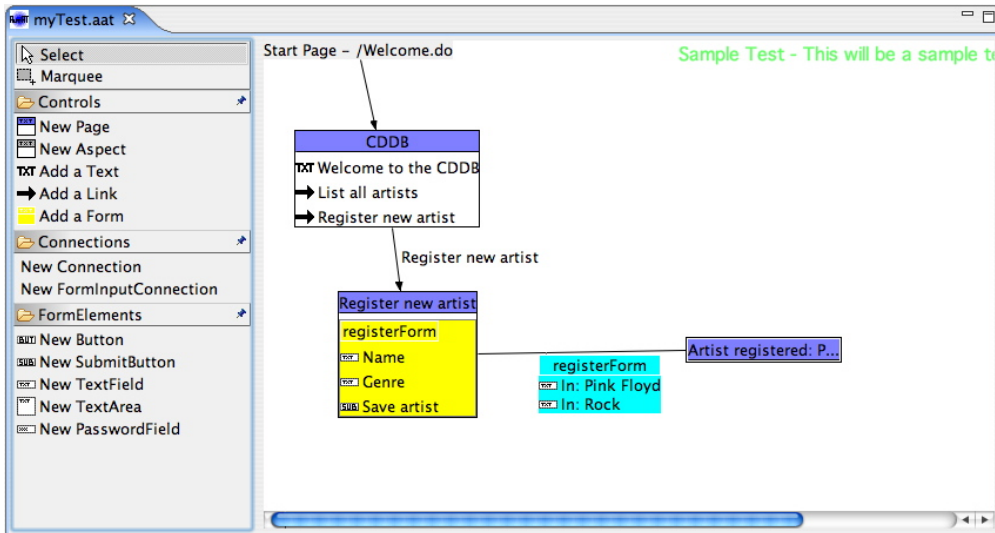


Figure 17.10: The final test

Chapter 18

User Testing Session

The user testing session that is the data collection phase in the GQM-method presented in Chapter 6, had four parts. It started out with some background information, then we worked with FitNesse, before we worked with AutAT. The last part was an evaluation summing up the session. These parts will be explained further in next sections.

Eight users participated in the session.

18.1 Background

The first part of the testing session was a short introduction to the field of testing and especially acceptance testing. The purpose was to motivate the tester for the test session and to let them know the background of AutAT. The philosophy behind Test Driven Development was described as an explanation to why there were no real web site to test against. One aspect that we emphasized was customer involvement which is important in XP and one of the reasons for stating this project.

Before the next session the user filled out the first – Background and Experience – section of a questionnaire. The questionnaire can be found in Appendix A.

18.2 FitNesse

This part started out with a short demonstration of FIT in general and FitNesse in particular. The demonstration lasted approximately fifteen minutes, and showed how to use the test functionality provided by jWebUnit's WebFixture for FitNesse. For aiding the testers in defining their tests we had prepared a little “helper” that can be found in Appendix C – FitNesse Commands.

After this introduction the tester performed the exercises that we had prepared as one can see in Figure 18.1. They can be found in Appendix B – Testing

Exercises. All the testers measured the time it took for completing each exercise.



Figure 18.1: The Test Session

After completing the exercises or exceding the time limit of 40 minutes, the users answered the questions found in Section A.2 of the questionnaire.

18.3 AutAT

This part was similar to the previous part. We started with a minimal introduction to AutAT focusing on the graphical user interface. The functionality was demonstrated and explained. This demonstration lasted approximately ten minutes.

After the introduction, the testers performed the same exercises as in the FitNesse session. They can be found in Appendix B. Again, the time used was measured and recorded.

After the testers had completed the tests they answered the questions shown in Section A.3 which focuses on all important aspects of AutAT.

18.4 Evaluation

Finally there was an evaluation session that began with the users answering the final section (A.4) of the questionnaire which is trying to compare AutAT and FitNesse.

There was also a more informal discussion lasting for about half an hour where the testers talked about what they liked, what they did not like, their general opinions and feedback and so forth.

Part IV

Evaluation

This part sums up the project. It first addresses the research question and goal attainment before it has a discussion of the important aspects of the project. Then we provide conclusions to the project and presents the future work.

Chapter 19

Research Analysis and Goal Attainment

The purpose of this chapter is to analyse and evaluate AutAT with respect to GQM. As seen in the begin of Chapter 6 the goal is linked to a goal attainment, the questions to answers and metrics are linked to measurements. Measurements, answers and the goal attainment is described in this chapter. First in the chapter is a section on the threats to validity as it is important for a good evaluating of AutAT.

19.1 Threats to Validity

In this section we will try to look at the most important threats to research validity. Wohlin et. al. [24] proposes a set of possible threats that we use as a basis for our list of threats to validity.

- **Low statistical power**

Low statistical power means that there is little ability for a test to reveal a true pattern in the data. There are rather few testers in our experiment, but by using paired t-test we believe it is sufficient for doing a good analysis according to Section D.1. An other, closely related, subject is that statistical purists might argue that it is wrong to convert the "agree" to "disagree" scale into a 1 to 4 discrete scale. However, according to Dybå in [6] who uses Stevens and Tukey, it is more important to be able analyze data than to let an overpurified view dictate how data can be used. Their view allows us to do the conversion described above.

- **Reliability of measures**

One should get the same output if one measure a phenomenon twice. As many of the questions and metrics tries to capture subjective human thoughts we can not be sure that the response will be the same doing the same experiment twice. However, guidance to how one should do the

exercises helps.

- **Maturation**

The testers might react differently as time passes. However, as there is little time between testing FitNesse and AutAT, we do not think that this is a serious threat in our case.

- **Selection**

There is a natural variation in human performance. It is a problem that we invited people that we know to the test session instead of doing a random pick out of the population. An other problem is that people within the selected group are different. However, as we can use paired student t-tests, variation due to selection will have little effects on the results.

- **Compensation**

The users can be affected by us giving them pizza for participating in the experiment. This is however not much of a payback for close to three hours of their time, so it should not affect the results much. In addition, the students are also used to such treatment in other experiments. Thus makes this a minor threat in our case.

- **Experimenter expectancies**

There is a chance that testers can bias the results of a study, consciously or unconsciously, based on what they expect from the experiment. It is possible that the users think that we show the state-of-the-art framework FitNesse and our solution AutAT that we propose as a solution for improvement. However, we explained that it was an experiment and proof of concept that might be totally wrong and that an honest review was important. Thus makes this a rather small threat in our case.

19.2 Measurements

The measurements in this section are linked to the metrics defined in Section 6.3. A detailed overview of all the statistical data can be found in Appendix E.

The user testing session described in Chapter 18 had eight testers. In addition, some people at BEKK have tested the software doing the same exercises as the participants in our user testing session. We value their efforts, but as there was only one that actually sent us the filled in questionnaire and the AutAT files we are not going to use that contribution except for measure M19 - User feedback. The reason for excluding this contribution is that we cannot be sure that the process they used was the same as for our experiment and that everything was done according to the instructions.

M1 - Fit knowledge

This metric is shows that there is little or no knowledge of FIT as it average 1.25 as indicated in Section E.1. There where no respondents that had better knowledge than “Heard about it” on the four point scale. This means that the two tools was analysed without bias.

M2 - AutAT time usage

The total of average time used on each exercise by the users is according to Section E.2 24.81 minutes – 24 minutes and 49 seconds.

M3 - FitNesse time usage

As one can see in Section E.3 the total of average time used on each exercise in the user test was 37.56 minutes – 37 minutes and 34 seconds.

M4 - AutAT’s ease of learning

In Section E.5 it is stated that most of the tester thought that AutAT is rather easy to learn.

M5 - FitNesse’s ease of learning

All but one of the testers of FitNesse answered “Somewhat agree” with the statement “Easy to learn” when considering FitNesse as one can see in Section E.6.

M6 - Compared ease of learning

When comparing ease of learning between AutAT and FitNesse which is M6, most testers thought that AutAT was clearly easier to learn than FitNesse. However, the average is not larger as one user totally disagreed as one can see in the stats on page 139.

M7 - AutAT’s ease of use

The testers on average stated “Somewhat agree” when faced with the statement “Easy to us” with respect to AutAT as seen Section E.9.

M8 - FitNesse's ease of use

When faced with the statement “Easy to use” with respect to FitNesse at question 6 in the questionnaire (see Appendix A), the testers answered on average 1.5. That means that they are between “Disagree” and “Somewhat disagree” when faced with the “Easy to use” statement.

M9 - Compared ease of use

On metric 9 on page 18 the testers averaged 3.75, which means that they almost agree with the statement “AutAT is easier to use than FitNesse”.

M10 - AutAT's syntax complexity

With an average of 3.25 according to Section E.13, the testers average between “Agree” and “Somewhat agree”. However, closer to “Somewhat agree” when it comes to the statement “Simple syntax” when considering AutAT.

M11 - FitNesse's syntax complexity

With an average of 2.5 the users are, as seen in Section E.14, neutral to whether FitNesse has a simple syntax or not.

M12 - Compared syntax complexity

When asked to consider the statement “AutAT has an easier syntax than FitNesse”, the testers averaged 3.63 and six out of eight fully agreed with the statement according to Section E.15.

M13 - AutAT's overview

As seen in Section E.17, the testers averaged 3.5 upon the metric 13. It means that they between “somewhat agree” and “agree” to the statement “easy to get an overview” with respect to AutAT.

M14 - FitNesse's overview

The average for metric 14 according to Section E.18 is 1.5. This means that the testers feels that it is rather difficult to get an overview of FitNesse.

M15 - Compared overview

E.19 Seven out of eight agreed with the statement “It is easier to get an overview of AutAT than FitNesse”, and the last tester somewhat agreed.

M16 - Modifying tests

The results from M16 that are presented in Section E.21, indicate that the majority of the testers felt that it was easier to modify AutAT tests than FitNesse tests. Most testers agreed with that and the average was 3.63, which means that the tester was leaning more towards “agree” than “somewhat agree”.

M17 - AutAT errors

As indicated in Section E.23 the total of average of syntactic and semantic (as described in Q1 – Quality, on page 13) errors for each exercise was 3.75. Exercise 3 was the toughest with 1.5 errors on the average, which means that its responsible for 40 % the errors.

M18 - FitNesse errors

The total number of average of syntactic and semantic errors for each exercise was 8.79 as stated in Section E.24.

M19 - User feedback

There were a lot of user feedback that the other metrics did not capture. This feedback can be seen in Section E.26. Most of the feedback is small changes that can make AutAT better. One major feature that some would like is “Copy & Paste”. Other possible features are to look at tests at higher levels and connecting them there, choose from a list of links when one is creating a link connection, create a start page and a start point when creating a new test, the graphical editor should scroll automatically and is should be possible to look at tests at a higher level an connect them there. An other thought is that it should be possible to tree structure tests.

There are some errors that are important to notice (and for us to fix). One is that a few of the testers had problems saving their AutAT files. There where one that thought that there was an different “Look&Feel” from what he was used to, and there were some problems with input in forms when creating the form input connection.

The were some negative sides like one thought that some of the terms were confusing. An other is that the test pages should not be based on its name/title, but that it should be possible to add the tittle to a page or an aspect like a text,

link or form. Some thought that it was difficult to write tests in the property view. Others thought that it was not a good idea to use Eclipse.

On the positive side, there are some things that the testers thought were good; aspects, easy to get an overview, quite fast, easy for nontechnical users to understand, and that it can enable quicker writing of tests. Some stated that they thought it was a very nice concept and prototype. Several of the testers thought that this project can do much for automatic acceptance testing web pages.

One thought that the difference between FitNesse and AutAT in the empirical study would have been greater if we had used other test candidates than well educated computer scientists.

19.3 Answers

The metrics in the previous section is used to answer the questions in the GQM-tree as described in the beginning of Chapter 6. The answers to the questions in Section 6.2 in the introduction are described in this section.

Q1 – Quality

The quality is as stated in Question 1, dependent on the number of errors in the tests developed by AutAT which is Metric 17 and FitNesse which is Metric 18. It is also dependent on the users knowledge of FIT. However, in the measurement of M1 which is found in the previous section, all the testers had so little knowledge of FIT that it has no influence on respectively the number or FitNesse errors.

The total number of average errors on the exercises are measured by M17 and M18. Their values are 3.75 and 8.79. Using these numbers there is a 57 % improvement when using AutAT.

This might not be very accurate, but according to Section E.25, AutAT is better than FitNesse with respect to quality on all the exercises by these p-values:

- Exercise 1: 0.0901
- Exercise 2: 0.0257
- Exercise 3: 0.1084
- Exercise 4: 0.0555
- Exercise 5: 0.0052

By using all the totals for every tester, there is a 1.60 % chance that we are wrong in concluding that AutAT is not better than FitNesse. For the total we have used all the values that were given. It does not give a complete picture as some of the testers did not complete all the FitNesse exercises (as one can see

in E.24) but we still used their total of tests they completed. This means that the 1.60 % chance of being wrong could be lower as we used the total errors all together for these testers on the AutAT exercises. However, we feel that it is sufficient for concluding that AutAT is better than FitNesse when it comes to quality, as 1.60 % is better than the 5 % that we use for alpha in our statistical tests.

Q2 – Efficiency

This answer depends on M1 – FIT knowledge, M2 – AutAT time usage, and M3 – FitNesse time usage. There was hardly any FIT knowledge measured by M1 so this did not influence the time spent on developing the tests with FitNesse.

The total average time used on each exercise was by using FitNesse 37 minutes and 34 seconds, and by using AutAT 24 minutes and 49 seconds as respectively indicated by measuring M3 and M2. This is approximately a 34 % improvement.

However, for concluding that the mean times are actually different we have performed a t-test that can be seen in Section E.4. We conclude that AutAT is better than FitNesse with the following p-values:

- Exercise 1: 0.0044
- Exercise 2: 0.0758
- Exercise 3: 0.0090
- Exercise 4: 0.0036
- Exercise 5: 0.0108

If we pool all the data, the t-test gives us a p-value less than 0.0001 when concluding that AutAT has a lower mean value than FitNesse when it comes to time usage. However, some of the participants did not finish all the FitNesse exercises. It was rather unfortunate, but we used the times they had totally anyway. However, a p-value less than 0.0001 could probably be even smaller giving us more certain results, but doing it this way has a few advantages as we can use pair t-test. This means that learning and the time usage are the only thing that might influence the results.

As we did the FitNesse tests first and followed on with AutAT using the same exercises, it is possible that the tester could learn the detailed content of the tests. This means that the users would not have needed to read the exercises doing them on AutAT. This is unlikely as the testers complained about having to read the exercises twice, but in order to be sure we added 2 minutes to the total time used by every users on AutAT. We did the t-test again and came up with the same result and approximately the same possibility of doing an error as without adding the 2 minutes.

We conclude that we believe that AutAT is more efficient than FitNesse.

Q3 – Usability

As we indicated when defining Q3 on page 13, there are several metrics that contribute to the answer to this question.

The measurements for M7 – AutAT’s ease of use, M8 – FitNesse’s ease of use and M9 – Compared ease of use, indicate clearly that AutAT is easier to use than FitNesse. The reason is that testers indicated that they on average stated “Somewhat agree” on M7 and right between “Disagree” and “Somewhat disagree” on M8. The t-test shown in Section E.12 that compares M7 and M8, indicates that AutAT has better perceived ease of use. With a p-value of 0.0013, which is pretty good. Also upon M9 almost all the tester thought that it was clearly easier to use AutAT than FitNesse. Also considering Section E.12, with a p-value of 0.0039.

Considering syntax complexity AutAT is superior to FitNesse as indicated by the measurement for M12 with a p-value of 0.035. Just considering the individual measurements for AutAT and FitNesse, respectively metrics M10 and M11, it is clear that AutAT is better than FitNesse when it comes to syntax simplicity even though FitNesse’s syntax is seen as neither complex nor simple. According to the t-test (see Section E.16) comparing M10 and M11, AutAT has a simpler syntax than FitNesse. However, the p-value is 0.07. This is not as good as we had hoped for.

The M13 – AutAT’s overview measurement indicates that it is fairly easy to get an overview over AutAT in contrary the M14 – FitNesse’s overview measurement indicates that it is rather hard to get an overview over FitNesse’s tests. However, the M15 – Compared overview, measurement concludes that almost every tester feels that it is quite a lot easier to get an overview of AutAT than with FitNesse. The p-value is 0.0039. The t-test indicates when comparing M13 and M14, that it is easier to get an overview in AutAT than in FitNesse. The p-value is less than 0.001 which is rather low, as one can see in Section E.20.

The measurement of M16 – Modifying tests, state that it is quite a lot easier to modify test with AutAT than with FitNesse. According to Section E.22 the p-value is 0.0039.

Looking at all measurements together it is quite clear that the perceived usability of AutAT is a lot better than FitNesse as AutAT is better on all measurements.

Q4 – Learning

As we stated when we defined Question 4 on page 14, the perceived ease of learning is dependent on M4 – AutAT’s ease of learning, M5 – FitNesse’s ease of learning and M6 – Compared ease of learning. It is also dependent on whether or not the tester has prior knowledge of FIT which is measured by M1.

It is clear that AutAT is somewhat easier to learn as the measurement of M4 states that most of the tester thought that AutAT is rather easy to learn and that measurement of M5 concluded that the FitNesse was somewhat easy to learn. The t-test (see Section E.8) gives us a p-value of 0.003. When on also consider M6 which states that the median tester think that AutAT is easier to learn, it is rather clear that most testers probably would learn AutAT a lot easier than FitNesse. The p-value here is 0.035.

Overall we are pretty confident in concluding that AutAT is easier to learn than FitNesse.

Q5 – GUI

According to Q5 on page 14: this question is answered by M19 – User feedback. The measurement tries to look at some of the strengths, weaknesses and possibilities that the testing session revealed. The strengths are the positive side of the application for instance that it is easy for nontechnical users to understand as well as for programmers.

Some of the opportunities for the AutAT tool are actually some of its bugs and others are small features that some of the testers wanted. When these are implemented, and most of them can be implemented really easily, the system will be quite a lot better.

There are some errors that are harder to cope with, and are therefore the weaknesses of the system. One such error is that some of the participants thought that the “Look&Feel” was a little unusual. Other negative sides are that some of the users thought that some of the terms that we are using is confusing and that having to use the Eclipse platform was rather negative. However, if there are many users that do not care for the Eclipse platform we consider to turn it into a RCP system (see Section 10.3.1).

There are also some strengths to consider. The most important ones are that it can enable faster writing of tests, give a better overview and enables nontechnical users to easier write and understand acceptance testes.

Overall are there a lot of possibilities and strengthes on one side and some weaknesses on the other side. It is clare that if we had used other types of testers with less technical experience, the results would probably have been different. However, as AutAT is a prototype and a proof of concept, it is quite promising.

19.4 Goal Attainment

The Goal Attainment is based in the GQM-Goal found in Section 6.1 as shown in Figure 6.1. We believe that AutAT is a good concept and better than the state-of-the art framework FitNesse when it comes to usability, quality and

efficiency of writing tests as the answers in the previous section have revealed. However, we have only been considering a software developers point of view an test driven development of web applications.

As a result we would say that the Goal is well attained.

Chapter 20

Discussion

The projects's problem definition in Chapter 2 has two major areas of interest. The first is to create a tool for automatic acceptance testing of web applications and the second is to do an empirical evaluation of the tool. These two areas are addressed in this discussion, before we add a few personal experiences that we would like to share with our readers.

20.1 AutAT Tool

There are several interesting aspects related to the AutAT tool. Firstly, the underlying technologies like the testing framework, Eclipse and GEF. And secondly, and probably more interesting is the tool itself.

20.1.1 Test Framework

We used jWebUnit and its WebFixture for FIT (as described in Chapter 9) as our foundation for executing the acceptance tests. It was a natural choice for us and has worked well. There are arguably a lot of features that cannot be tested with this framework. One of the problems is JavaScript which quite often causes jWebUnit to fail. When doing test driven development, this should not be a problem, as the software can be suited to fit the tests, but many applications will need to use JavaScript. However, jWebUnit proved to be more than sufficient to use since this is a proof of concept.

Whether is a good choice for the future development of AutAT is hard to say. However, AutAT is designed to handle a change in the underlying testing technology. As a result it can meet future demands, either by changing to another test framework or by modifying jWebUnit.

20.1.2 Eclipse

To develop plugins for the Eclipse Platform is actually quite easy, due to the good support within Eclipse itself and the plugin development tools. As Eclipse itself is built as a set of plugins, the tools for developing plugins is well supported. When it comes to how well the testers liked the platform there were some that liked it while others disliked it. We, however, think that it is nice and has enabled us to provide a lot of functionality quickly.

20.1.3 GEF

When starting to use the Graphical Editor Framework (GEF) the complexity seems high, as with any complex framework. However, after some initial studies we were pleased with this tool as it provided a good foundation for creating a tool like AutAT. The final iterations of the development process when we knew the framework well, were highly productive.

20.1.4 AutAT Itself

The reason for creating AutAT was to use it as a proof of concept. As it is a proof of concept it contains bugs and not all the features are implemented. However, for checking whether or not it is a step in the right direction it is appropriate. That it is a proof of concept does not mean that it is poorly implemented or just a prototype. AutAT can be developed further to a larger and better tool. AutAT was also developed a little further after the test session to remove some of the minor errors and adding a few minor features that increases the usability and makes it faster to use.

The differences between AutAT and other acceptance testing tools for web applications are many. One is the graphical user interface that is easy to work with and to get an overview of than the earlier text based systems. AutA also introduces new ideas like aspects which let the tester do the same checks on several while only defining it once. By using the existing tools, the same check must be repeated by hand every time it is to be used. The use of aspects reduce redundancy in the tests.

20.2 Experiment

The results from the experiments were presented in the previous chapter. We would have liked to have done even more experiments. However, we are really satisfied as it gave us the answers we needed when it comes to testing AutAT as a proof of concept.

Most of the feedback indicates that we have made the right decisions. We would like to add a quote from our supervisor at BEKK (original in Norwegian and

translated to English). Even though he is deeply involved in the project it makes all the hard work rewarding when someone like what we have done:

Norwegian:

Jeg bøyer meg i støvet. Det programmet dere har laget er vidunderlig!

Vil anslå at man med litt finpussing av GUI blir 5 ganger (!) mer produktiv med AutAT enn med FitNesse (hovedsakelig pga. psykologi og gjenbruk) og har mye større oversikt og får bedre kvalitet. FitNesse ser ut som et sliten travet i forhold.

AutAt vs FitNesse = Java vs Assembly

English:

Hats off. The program you have developed is wonderful!

My estimat is that one will be five times(!) more productive with AutAT than with FitNesse after the GUI has be fine tuned (mainly because of psychology and reuse) and one has a far better overview and gets better quality. FitNesse looks like a old and tired horse in comparison.

AutAt vs FitNesse = Java vs Assembly

– *Christian Schwarz, BEKK Consultant and project supervisor*

20.3 Personal Experiences

We have enjoyed working with this project mainly due to the wide range of aspects that we have had to work with. First of all is getting to know test driven development better. It has also been interesting to get to know the insides of Eclipse and GEF along with its patterns. An other interesting aspect is the empirical testing session that we really think is the most important test when it comes to this project as it checks whether or not this is an useful tool and if it should be developed further. The most creative and difficult aspect was to figure out what we should and could do to make this tool superior to earlier web acceptance testing tools and frameworks.

The cooperation with our supervisors at BEKK and NTNU has been great compared to other projects that we have worked with. We have had the possibility to work on our own, to come up with ideas and been given great feedback which really helped us. Analyzing the empirical data has been rewarding as Professor Stålhane, our supervisor at NTNU, without doubt is amazing when it comes to statistics.

Even though we are formally finished with this project at the delivery of this report, we do not consider our work with AutAT finished. There are many new features that we want to add, and we enjoy working with the people at BEKK. We will continue to work on the open source AutAT project to make it better in the future and help developers all over the world.

Chapter 21

Conclusion

The problem definition has two major topics. First to create a tool for acceptance testing web applications. The AutAT tool is able to test the basic functionality of web pages. It is different to earlier tools as there are new ideas that try to increase the tool's usability, quality and efficiency.

The tool has been empirically tested. The results are quite clear: AutAT is an improvement when it comes to quality and efficiency compared to state of the art tools of test driven development.

The tool aimed at being a proof of concept and a basis for a larger, future tool for developing web acceptance tests. We believe that we have hit what we aimed for, and we are going to add AutAT to the BEKK Open Source Software (BOSS) site in the near future. It is already added to SourceForge.net¹ which BOSS uses as a repository.

This tool can potentially ease test driven development of web applications, and it will be developed further and used by BEKK and anybody else that wants to download it.

¹The world's largest development and download repository for Open Source code and applications

Chapter 22

Future Work

This project could not address all possible features or do a larger empirical study. However, the following issues are possible points of interest for future work and are addressed in this chapter. There is also a section on the future work at NTNU and BEKK.

22.1 Future Work with the AutAT Tool

The test session and the presentation we had at BEKK at the end of this project helped us to identify a set of possible improvements. These and the requirements that we did not implement in addition to some thoughts that we got along the way are listed here as possible future work.

- *Improve form support.* There are some form elements like for instance radio button that are not implemented.
- *Title* on a page should be an element that can be added to an aspect or a page instead of having to be set on a page. The existing “title” should instead be a logical name.
- It should be possible to *run tests* in AutAT instead of having to export the test to FIT.
- A *statistic* module should be added so one can see how well the system is doing. It is also useful for getting feedback on the progress in the development process.
- *Global aspects* in addition to the aspects that are in AutAT today. Global aspects can be reused in many different tests in a project. Today’s aspects are only available for one test.
- *Extending tests* means that it should be possible to start a test from where an previous test ends. This is useful for applications that requires user to do a set of actions like logging in, before it can do anything else.

- *Parameterized tests and data test set* is having a data set that can be used to fill in a form. By parameterize the test it is also possible to check that it is a piece of information on the next page which is in the data set and is related to the input in the form.
- To have *other exporters* can be interesting as jWebUnit does not have to be the preferred choice for all applications.
- Building a *new testing framework* might be interesting as it is not sure that today's testing frameworks supports all that we might like to test in the future.
- *Customized resource navigator* instead of the standard version that AutAT uses now. The reason for having such a navigator is to use the metaphors defined more closely, letting the user see user stories instead of folders organizing the tests.
- *Load and performance testing* can be added, using the already defined tests in the system.
- A *site map* tries to lay out all the pages in an application. It should then use the other test in the system to go between pages to see if it is possible to visit them all.

22.2 Future Empirical Experiments

Testing AutAT on one or more of BEKK's projects is interesting as it is a "real life" study where it is possible to really determine whether or not this is a good tool. These tests will include real customers, providing valuable feedback from non-technical users. BEKK is interested in doing such an experiment and the tool is ready for it.

22.3 Future Work at NTNU and BEKK

BEKK and NTNU have decided to continue this project by proposing it as a student project for student at the final year in master program in computer science this fall semester and also the following spring semester as a master thesis.

Part V

Appendix

The appendix contains chapters concerning the empirical experiment: the questionnaire for the AutAT test session, a set of testing exercises and FitNesse commands used in the testing session. Then we present some statistical background and user feedback with some analysis. The last chapter shows the XML schemas for the files produced and used by AutAT.

Appendix A

Questionnaire for AutAT testing

This first section of this questionnaire should be before the doing any of the exercises in Appendix B

A.1 Background and Experience

1 *FIT knowlegde*

- Using it at a daily basis.
- Tried it.
- Heard about it.
- Unknown.

A.2 FitNesse

Answer this section after doing the five test exercises on FitNesse.

A.2.1 Time Usage

2 *What is the time usage (minutes):*

1. – Exercise:
2. – Exercise:
3. – Exercise:
4. – Exercise:

5. – Exercise:

A.2.2 Statements

What is your opinion about this statements regarding FitNesse?

3 *Easy to learn*

- Agree.
- Somewhat agree.
- Somewhat disagree.
- Disagree.

4 *Simple syntax*

- Agree.
- Somewhat agree.
- Somewhat disagree.
- Disagree.

5 *Easy to get an overview*

- Agree.
- Somewhat agree.
- Somewhat disagree.
- Disagree.

6 *Easy to use*

- Agree.
- Somewhat agree.
- Somewhat disagree.
- Disagree.

7 *Other comments about FitNesse?*

A.3 AutAT

Answer this section after doing the five test exercises on AutAT.

A.3.1 Time Usage

8 *What is the time usage (minutes):*

1. – Exercise:
2. – Exercise:
3. – Exercise:
4. – Exercise:
5. – Exercise:

A.3.2 Statements

What is your opinion of these statements regarding AutAT?

9 *Easy to learn*

- Agree.
- Somewhat agree.
- Somewhat disagree.
- Disagree.

10 *Simple syntax*

- Agree.
- Somewhat agree.
- Somewhat disagree.
- Disagree.

11 *Easy to get an overview*

- Agree.
- Somewhat agree.
- Somewhat disagree.
- Disagree.

12 *Easy to use*

- Agree.
- Somewhat agree.
- Somewhat disagree.
- Disagree.

13 *What do you generally think of AutAT (its strengths, weaknesses and opportunities)?*

14 *What do you think of the AutAT perspective (its strengths, weaknesses and opportunities)?*

15 *What do you think of AutAT's NamesAndURL editor (its strengths, weaknesses and opportunities)?*

16 *What do you think of AutAT's GraphicalTest editor (its strengths, weaknesses and opportunities)?*

17 *What do you think of AutAT's Property view (its strengths, weaknesses and opportunities)?*

18 *Other comments about AutAT?*

A.4 Comparison: FitNesse versus AutAT

The purpose of this section is to compare FitNesse against AutAT.

A.4.1 Statements

What is your opinion of these statements regarding AutAT versus FitNesse?

19 *AutAT is easier and faster to learn than FitNesse*

- Agree.
- Somewhat agree.
- Somewhat disagree.
- Disagree.

20 *AutAT has an easier syntax than FitNesse*

- Agree.
- Somewhat agree.
- Somewhat disagree.
- Disagree.

21 *It is easier to get an overview of AutAT than FitNesse*

- Agree.
- Somewhat agree.
- Somewhat disagree.
- Disagree.

22 *AutAT is easier to use than FitNesse:*

- Agree.
- Somewhat agree.
- Somewhat disagree.
- Disagree.

23 *It is easier to modify tests with AutAT than with FitNesse*

- Agree.
- Somewhat agree.
- Somewhat disagree.
- Disagree.

24 *Other comments about AutAT versus FitNesse*

Appendix B

Testing Exercises

This exercise will create a few tests for a web-based cd database. The database lets users save artists and relate albums to these artists. The users can then list and modify the registered artists and albums.

The system we are testing is located at <http://cddb.ovstetun.no>

B.1 Exercise 1: Test the Front Page

Create a new test, starting from the front page. This page is located at the local URL “/Welcome.do”.

The front page should be titled “Welcome”

Check that the page contains a welcome text.

Check that the page contains a link named “List all artist”

Check that the page contains a link named “Register new artis”

B.2 Exercise 2: List All Artists

Create a new test, starting from the front page.

Follow the link named “List all artists”

The new page should be titled “All artists”

Check that the following artists are listed:

- Tool
- Tori Amos
- Seigmen

Check that the following artists are NOT listed:

- Vikingarna
- NoName

B.3 Exercise 3: Register New Artist

Start at the front page, choose “Register a new artist”.

The registration page should be titled “Register new artist”, and contain a form with the following items:

- A textfield labeled “Name”
- A textfield labeled “Genre”
- A submit button labeled “Save artist”

Enter values into these fields, registering the artist “Span” in a genre named “Rock”.

The next page should be titled “New artist - Span”, containing a confirmation text stating “Registration successful, Span registered”.

B.4 Exercise 4: View Info About Artist and Register a New Album

The list of all artists is located at the local URL “/ListArtists.do”.

Start a new test from this location. The page should still be titled “All artists”.

The artist names should be links. Click the link named “Seigmen”.

The next page should be titled “Artist info - Seigmen”.

Check that the albums “Metropolis” and “Total” are present.

Check for a link named “Register new album”.

Follow this link to a new page titled “Register new album”, containing a form with the textfields “Name”, “Year” and “Rating” and a submit button named “Save album”.

Enter the values “Pluto”, “1992” and “7”.

The next page will contain a confirmation message, with the title “Album registered”.

B.5 Exercise 5: Repeating Checks

Now we want to check that some elements are present on several web pages. We start on the front page, then follow a link named “Genre” which leads to a page titled “Genres”. Here we click a link named “Pop”, and come to a new page named “Genre: Pop”.

For all these pages, we want to check that the following elements are present:

- A link named “About”
- A link named “Forum”
- A link named “Contact”

- A form containing a textfield labeled “Search” and a submitbutton labeled “search”
- A text: “Copyright 2005 tm and sk”.

Appendix C

FitNesse Commands

Here is a description of how FitNesse works and the commands that can be used.

C.1 General commands in FitNesse

Fitnesse uses a general wiki syntax. The pages can contain regular text to describe what the test is doing. What is special is that when you use Camel-Case (alternation of capitalized and non-capitalized) letters, a new page will be created. When a page starts with the word Test (as in TestNewPage), it will be interpreted as a test, and can be executed. Subpages are separated by a punctuation, as in TestSuperPage.TestSubPage

The cells in a table row are separated by a | (pipe). The first row is the name of the fixture used to run the test. When using jwebunit, the first row in a table is always:

```
!|-net.sourceforge.jwebunit.fit.WebFixture-!
```

C.2 Commands for using jWebUnit with FitNesse

A jwebunit test must be initialized with the following lines:

```
|base url |<base url>| location of web application
```

```
|begin |<page>| URL appended to the location. This location will be the first page of the test.
```

C.2.1 Checks for text and links

```
|check|title equals|<value>| checks the title of the active page
```

```
|check|text (not) present|<value>| checks that <value> is present (or not)
```

```
|check|link (not present)|<value>| link with the <value> as text is present (or
```

not)

C.2.2 Navigation commands

|press|link|<text>| presses the link with the given text

C.2.3 Checks for forms

|check|form element present with label|<value>|

|check|submit button (not) present|<value>| checks that a submit button with the given text is present

|check|button present (not) with text|<value>| checks that a button (not submit button) with the given text is present

C.2.4 Enter values to a form

|enter with label|<label>|<value>| enters <value> into the form element with label <label>

|press| submit| submits the form.

C.3 Example

```
!-net.sourceforge.jwebunit.fit.WebFixture-!  
|base url |http://www.idi.ntnu.no/ ovstetun/test| |
|begin |/index.php|  
|check|title equals|Test side|  
|check|form element present with label|Name: |  
|check|submit button present|Save|  
|enter with label|Name: |testName|  
|press|submit|  
|check|text present|Name: |  
|check|text not present|jalla tekst|
```

Appendix D

Statistical Background

This appendix addresses some of the underlying statistics.

D.1 Analysing the Number of Testers

Using:

Type I error: 0.05 Type II error: 0.20

Gives an ES approximately 32.

$$N \geq \frac{ES}{\left(\frac{|\bar{x}-\bar{y}|}{SD}\right)^2}$$

$$SD = \sqrt{\frac{SD_x^2}{n_x} + \frac{SD_y^2}{n_y}}$$

As N is 8, $\frac{|\bar{x}-\bar{y}|}{SD}$ has to be larger than 2 to make sure that we have enough tester.

As one can see in Table D.1 which uses data from t-tests in Appendix E, these values are good except for the simple syntax analysis where it is close to being good enough. One can also argue that N should be 16 as we use paired t-tests, then there are no problems with any of the values.

Section	Analysis	$\frac{ \bar{x}-\bar{y} }{SD}$
E.4	Time, total all exercises	5.70
E.25	Errors, total all exercises	2.65
E.8	Ease of learning	4.25
E.16	Syntax complexity	1.82
E.20	Overview	6.11
E.12	Ease of use	3.97

Table D.1: Number of testers analysis

D.2 Binary Probability Distribution

The binary probability distribution is used as p-values for the analyses where one look at how the users have compared AutAT with FitNesse in Appendix D.

$$\sum_{i=n}^N \binom{N}{i} p^i p^{N-i}$$

$$p = \frac{1}{2} \quad N = 8$$

n	$\binom{8}{n} \frac{1^n}{2} \frac{1^{8-n}}{2}$	$\sum_{i=n}^8 \binom{8}{i} \frac{1^i}{2} \frac{1^{8-i}}{2}$
8	0,00390625	0,00390625
7	0,03125	0,03515625
6	0,109375	0,14453125
5	0,21875	0,36328125
4	0,2734375	0,63671875
3	0,21875	0,85546875
2	0,109375	0,96484375
1	0,03125	0,99609375
0	0,00390625	1

Appendix E

User feedback

This appendix contains all the user feedback from the test session and the data analysis.

Number of testers: 8

For an evaluation of the number of testers see Section D.1.

Testers occupation: 100 % last year Master of Technology in Computer Science.

E.1 M1 - Fit Knowledge

Setting value for each item:

1. Unknown.
2. Heard about it.
3. Tried it.
4. Using it at a daily basis.

Tester:	1	2	3	4	5	6	7	8
Fit knowledge:	1	1	1	1	1	1	2	2

Average: 1.25

Median: 1

Highest value: 2

Lowest value: 1

E.2 M2 - AutAT Time Usage

Tester:	1	2	3	4	5	6	7	8	Average
Exercise: 1	3	3	8	5	7	2.5	4	3	4.44
Exercise: 2	4	3	4	4	5	3.5	7	6	4.56
Exercise: 3	5	5	5	10	5	5	5	4	5.5
Exercise: 4	4	7	6	6	7	5.5	8	5	6.06
Exercise: 5	4	5	4	5	5	3	4	4	4.25
Total:	20	23	27	30	29	19.5	28	22	24.81

E.3 M3 - FitNesse Time Usage

Tester:	1	2	3	4	5	6	7	8	Average
Exercise: 1	10	7	9	6.5	20	5	11	7	9.4375
Exercise: 2	3	3	5	6.5	10	3	9	6	5.69
Exercise: 3	8	6	5	10	10	6	7	7	7.38
Exercise: 4	8	9	6	10	-	8	9	7	8.14
Exercise: 5	7	6	10	7	-	3.5	-	8	6.92
Total:	36	31	35	40	40	25.5	36	35	37.56

Average only counts finished test.

E.4 Time Analysis

For this analysis we have used M2 - AutAT Time Usage, and M3 - FitNesse Time Usage. We are using a t-test to check whether or not AutAT is better than FitNesse when it comes to time usage.

Exercise 1		
t-Test: Paired Two Sample for Means		
	FitNesse	AutAT
Mean	9.4375	4.4375
Variance	22.10267857	4.245535714
Observations	8	8
Pearson Correlation	0.559937503	
Hypothesized Mean Difference	0	
Degrees of Freedom	7	
t Stat	3.592106041	
P(T<=t) one-tail	0.004416144	
t Critical one-tail	1.894577508	

Exercise 2		
t-Test: Paired Two Sample for Means		
	FitNesse	AutAT
Mean	5.6875	4.5625
Variance	7.495535714	1.816964286
Observations	8	8
Pearson Correlation	0.731868107	
Hypothesized Mean Difference	0	
Degrees of Freedom	7	
t Stat	1.609049749	
P(T<=t) one-tail	0.075820406	
t Critical one-tail	1.894577508	

Exercise 3		
t-Test: Paired Two Sample for Means		
	FitNesse	AutAT
Mean	7.375	5.5
Variance	3.410714286	3.428571429
Observations	8	8
Pearson Correlation	0.563970591	
Hypothesized Mean Difference	0	
Degrees of Freedom	7	
t Stat	3.071015748	
P(T<=t) one-tail	0.009020863	
t Critical one-tail	1.894577508	

Exercise 4		
t-Test: Paired Two Sample for Means		
	FitNesse	AutAT
Mean	8.142857143	5.928571429
Variance	1.80952381	1.702380952
Observations	7	7
Pearson Correlation	0.386620647	
Hypothesized Mean Difference	0	
Degrees of Freedom	6	
t Stat	3.991012001	
P(T<=t) one-tail	0.003596141	
t Critical one-tail	1.943180905	

Exercise 5		
t-Test: Paired Two Sample for Means		
	FitNesse	AutAT
Mean	6.916666667	4.166666667
Variance	4.641666667	0.566666667
Observations	6	6
Pearson Correlation	0.318573492	
Hypothesized Mean Difference	0	
Degrees of Freedom	5	
t Stat	3.296704942	
P(T<=t) one-tail	0.010776801	
t Critical one-tail	2.015049176	

Total, all exercises		
t-Test: Paired Two Sample for Means		
	FitNesse	AutAT
Mean	34.8125	24.8125
Variance	22.56696429	17.42410714
Observations	8	8
Pearson Correlation	0.741814919	
Hypothesized Mean Difference	0	
Degrees of Freedom	7	
t Stat	8.699176724	
P(T<=t) one-tail	2.65924E-05	
t Critical one-tail	1.894577508	

Total, all exercises adjusted for reading		
t-Test: Paired Two Sample for Means		
	FitNesse	AutAT
Mean	34.8125	26.8125
Variance	22.56696429	17.42410714
Observations	8	8
Pearson Correlation	0.741814919	
Hypothesized Mean Difference	0	
Degrees of Freedom	7	
t Stat	6.959341379	
P(T<=t) one-tail	0.000109678	
t Critical one-tail	1.894577508	

E.5 M4 - AutAT's Ease of Learning

Setting value for each item:

1. Disagree.
2. Somewhat disagree.

3. Somewhat agree.

4. Agree.

Tester:	1	2	3	4	5	6	7	8
Easy to learn:	4	3	3	4	4	4	4	4

Average: 3.75

Median: 4

Highest value: 4

Lowest value: 3

E.6 M5 - FitNesse's Ease of Learning

Setting value for each item:

1. Disagree.

2. Somewhat disagree.

3. Somewhat agree.

4. Agree.

Tester:	1	2	3	4	5	6	7	8
Easy to learn:	3	3	3	3	3	3	2	3

Average: 2.89

Median: 3

Highest value: 3

Lowest value: 2

E.7 M6 - Compared Ease of Learning

Setting value for each item:

1. Disagree.

2. Somewhat disagree.

3. Somewhat agree.

4. Agree.

Tester:	1	2	3	4	5	6	7	8
AutAT is easier and faster to learn than FitNesse:	4	3	1	4	3	4	4	4

Average: 3.38

Median: 4

Highest value: 4

Lowest value: 1

E.8 Analyzing Ease of Learning

For this analysis we have used M4 - AutAT's Ease of Learning, and M5 - FitNesse's Ease of Learning. We are using a t-test to check whether or not AutAT is better than FitNesse when it comes to ease of learning.

Ease of Learning		
t-Test: Paired Two Sample for Means		
	FitNesse	AutAT
Mean	2.875	3.75
Variance	0.125	0.214285714
Observations	8	8
Pearson Correlation	-0.21821789	
Hypothesized Mean Difference	0	
Degrees of Freedom	7	
t Stat	-3.861740991	
P(T<=t) one-tail	0.00309876	
t Critical one-tail	1.894577508	

Having 7 users stating "somewhat agree" or "agree" and 1 user stating "somewhat disagree" or "disagree" by coincidence is rather unlikely. As seen in Section D.2 the p-value is 0.04.

E.9 M7 - AutAT's Ease of Use

Setting value for each item:

1. Disagree.
2. Somewhat disagree.
3. Somewhat agree.
4. Agree.

Tester:	1	2	3	4	5	6	7	8
Easy to use:	4	3	2	3	4	2	3	3

Average: 3

Median: 3

Highest value: 4

Lowest value: 2

E.10 M8 - FitNesse's Ease of Use

Setting value for each item:

1. Disagree.
2. Somewhat disagree.
3. Somewhat agree.
4. Agree.

Tester:	1	2	3	4	5	6	7	8
Easy to use:	1	2	2	1	3	1	1	1

Average: 1.5

Median: 1

Highest value: 3

Lowest value: 1

E.11 M9 - Compared Ease of Use

Setting value for each item:

1. Disagree.
2. Somewhat disagree.
3. Somewhat agree.
4. Agree.

Tester:	1	2	3	4	5	6	7	8
AutAT is easier to use than FitNesse	4	3	4	3	4	4	4	4

Average: 3.75

Median: 4

Highest value: 4

Lowest value: 3

E.12 Analyzing Ease of Use

For this analysis we have used M7 - AutAT's Ease of Use, and M8 - FitNesse's Ease of Use. We are using a t-test to check whether or not AutAT is better than FitNesse when it comes to ease of use.

Ease of Use		
t-Test: Paired Two Sample for Means		
	FitNesse	AutAT
Mean	1.5	3
Variance	0.571428571	0.571428571
Observations	8	8
Pearson Correlation	0.25	
Hypothesized Mean Difference	0	
df	7	
t Stat	-4.582575695	
P(T<=t) one-tail	0.001267998	
t Critical one-tail	1.894577508	

Having 8 users stating “somewhat agree” or “agree” and 0 user stating “somewhat agree” or “agree” by coincidence is unlikely. As seen in Section D.2 the p-value is 0.004.

E.13 M10 - AutAT’s Syntax Complexity

Setting value for each item:

1. Disagree.
2. Somewhat disagree.
3. Somewhat agree.
4. Agree.

Tester:	1	2	3	4	5	6	7	8
Simple syntax:	4	3	3	3	4	3	3	3

Average: 3.25

Median: 3

Highest value: 4

Lowest value: 3

..

E.14 M11 - FitNesse’s Syntax Complexity

Setting value for each item:

1. Disagree.
2. Somewhat disagree.
3. Somewhat agree.
4. Agree.

Tester:	1	2	3	4	5	6	7	8
Simple syntax:	1	3	2	4	3	3	1	3

Average: 2.5
 Median: 3
 Highest value: 4
 Lowest value: 1

E.15 M12 - Compared Syntax Complexity

Setting value for each item:

1. Disagree.
2. Somewhat disagree.
3. Somewhat agree.
4. Agree.

Tester:	1	2	3	4	5	6	7	8
AutAT has an easier syntax than FitNesse:	4	2	4	4	4	4	3	4

Average: 3.63
 Median: 4
 Highest value: 4
 Lowest value: 2

E.16 Analyzing Syntax Complexity

For this analysis we have used M10 - AutAT's Syntax Complexity, and M11 - FitNesse's Syntax Complexity. We are using a t-test to check whether or not AutAT is better than FitNesse when it comes to syntax complexity.

Syntax Complexity		
t-Test: Paired Two Sample for Means		
	FitNesse	AutAT
Mean	2.5	3.25
Variance	1.142857143	0.214285714
Observations	8	8
Pearson Correlation	-0.288675135	
Hypothesized Mean Difference	0	
Degrees of Freedom	7	
t Stat	-1.655031853	
P(T<=t) one-tail	0.0709458	
t Critical one-tail	1.894577508	

Having 7 users stating “somewhat agree” or “agree” and 1 user stating “somewhat agree” or “agree” by coincidence is rather unlikely. As seen in Section D.2 the p-value is 0.04.

E.17 M13 - AutAT’s Overview

Setting value for each item:

1. Disagree.
2. Somewhat disagree.
3. Somewhat agree.
4. Agree.

Tester:	1	2	3	4	5	6	7	8
Easy to get an overview:	4	3	3	4	3	3	4	4

Average: 3.5

Median: 3.5

Highest value: 4

Lowest value: 3

E.18 M14 - FitNesse’s Overview

Setting value for each item:

1. Disagree.
2. Somewhat disagree.
3. Somewhat agree.
4. Agree.

Tester:	1	2	3	4	5	6	7	8
Easy to get an overview:	1	2	1	1	3	1	1	2

Average: 1.5
 Median: 1
 Highest value: 3
 Lowest value: 1

E.19 M15 - Compared Overview

Setting value for each item:

1. Disagree.
2. Somewhat disagree.
3. Somewhat agree.
4. Agree.

Tester:	1	2	3	4	5	6	7	8
It is easier to get an overview of AutAT than FitNesse:	4	4	4	3	4	4	4	4

Average: 3.88
 Median: 4
 Highest value: 4
 Lowest value: 3

E.20 Analyzing Overview

For this analysis we have used M13 - AutAT's Overview, and M14 - FitNesse's Overview. We are using a t-test to check whether or not AutAT is better than FitNesse when it comes to overview.

Ease of getting an overview		
t-Test: Paired Two Sample for Means		
Mean	1.5	3.5
Variance	0.571428571	0.285714286
Observations	8	8
Pearson Correlation	-0.353553391	
Hypothesized Mean Difference	0	
Degrees of Freedom	7	
t Stat	-5.291502622	
P(T<=t) one-tail	0.000566892	
t Critical one-tail	1.894577508	

Having 8 users stating “somewhat agree” or “agree” and 0 user stating “somewhat agree” or “agree” by coincidence is unlikely. As seen in Section D.2 the p-value is 0.004.

E.21 M16 - Modifying Tests

Setting value for each item:

1. Disagree.
2. Somewhat disagree.
3. Somewhat agree.
4. Agree.

Tester:	1	2	3	4	5	6	7	8
It is easier to modify tests with AutAT than with Fit-Nesse:	4	3	4	3	4	4	3	4

Average: 3.63

Median: 4

Highest value: 4

Lowest value: 3

E.22 Analyzing Modifying Tests

Having 8 users stating “somewhat agree” or “agree” and 0 user stating “somewhat agree” or “agree” by coincidence is unlikely. As seen in Section D.2 the p-value is 0.004.

E.23 M17 - AutAT Errors

Tester:	1	2	3	4	5	6	7	8	Average
Exercise: 1	1	0	3	1	0	0	1	0	0.75
Exercise: 2	0	0	1	0	2	0	0	0	0.38
Exercise: 3	1	3	2	1	1	3	0	1	1.5
Exercise: 4	0	1	0	0	1	2	1	1	0.75
Exercise: 5	0	0	1	0	1	0	1	0	0.38
Total:	2	4	7	2	5	5	3	2	3.75

Average only counts finished tests.

E.24 M18 - FitNesse Errors

Tester:	1	2	3	4	5	6	7	8	Average
Exercise: 1	2	3	3	0	0	1	2	0	1.38
Exercise: 2	4	1	2	0	2	2	0	1	1.5
Exercise: 3	2	4	3	0	4	2	1	1	2.13
Exercise: 4	5	3	4	1	-	1	1	1	2.29
Exercise: 5	2	1	2	0	-	2	-	2	1.5
Total:	15	12	14	1	6	8	4	5	8.79

Average only counts finished test.

E.25 Error Analysis

For this analysis we have used M17 - AutAT Errors, and M18 - FitNesse Errors. We are using a t-test to check whether or not AutAT is better than FitNesse when it comes to number of errors.

Exercise 1		
t-Test: Paired Two Sample for Means		
	FitNesse	AutAT
Mean	1.375	0.75
Variance	1.696428571	1.071428571
Observations	8	8
Pearson Correlation	0.503322296	
Hypothesized Mean Difference	0	
Degrees of Freedom	7	
t Stat	1.488351394	
P(T<=t) one-tail	0.090131186	
t Critical one-tail	1.894577508	

Exercise 2		
t-Test: Paired Two Sample for Means		
	FitNesse	AutAT
Mean	1.5	0.375
Variance	1.714285714	0.553571429
Observations	8	8
Pearson Correlation	0.219970673	
Hypothesized Mean Difference	0	
Degrees of Freedom	7	
t Stat	2.346242607	
P(T<=t) one-tail	0.025685384	
t Critical one-tail	1.894577508	

Exercise 3		
t-Test: Paired Two Sample for Means		
	FitNesse	AutAT
Mean	2.125	1.5
Variance	2.125	1.142857143
Observations	8	8
Pearson Correlation	0.504184173	
Hypothesized Mean Difference	0	
Degrees of Freedom	7	
t Stat	1.357241785	
P(T<=t) one-tail	0.108418773	
t Critical one-tail	1.894577508	

Exercise 4		
t-Test: Paired Two Sample for Means		
	FitNesse	AutAT
Mean	2.285714286	0.714285714
Variance	2.904761905	0.571428571
Observations	7	7
Pearson Correlation	-0.572896485	
Hypothesized Mean Difference	0	
Degrees of Freedom	6	
t Stat	1.868257106	
P(T<=t) one-tail	0.055473277	
t Critical one-tail	1.943180905	

Exercise 5		
t-Test: Paired Two Sample for Means		
	FitNesse	AutAT
Mean	1.5	0.166666667
Variance	0.7	0.166666667
Observations	6	6
Pearson Correlation	0.292770022	
Hypothesized Mean Difference	0	
Degrees of Freedom	5	
t Stat	4	
P(T<=t) one-tail	0.005161708	
t Critical one-tail	2.015049176	

Total, all exercises		
t-Test: Paired Two Sample for Means		
	FitNesse	AutAT
Mean	8.125	3.75
Variance	25.55357143	3.357142857
Observations	8	8
Pearson Correlation	0.40487462	
Hypothesized Mean Difference	0	
Degrees of Freedom	7	
t Stat	2.674283672	
P(T<=t) one-tail	0.015900802	
t Critical one-tail	1.894577508	

E.26 M19 - User Feedback

E.26.1 FitNesse

Comments about FitNesse.

- Quite fast, but rather cumbersome.
- Should have been modular so one could do checks on every page by defining them in one place.
- Too much writing.
- Difficult syntax. Easily writing errors.
- Inconsistent syntax.
- Really boring.

E.26.2 AutAT

What do you generally think of AutAT (its strengths, weaknesses and opportunities)?

- Should not be based on the title/name of a page.
- Some start difficulties.
- Quite fast.
- Error when saving the tests.
- Too much manual work when creating projects and tests. The wizards should do more.
- Easy for non-technical personnel to understand.
- Easy to get an overview.
- Aspects are good.
- Some terms are confusing.
- It looks promising, thinks that it can enable faster writing of tests and is suitable for others than programmers.
- "It's totally slick!" :-)

What do you think of the AutAT perspective (its strengths, weaknesses and opportunities)?

- A little bit too small editor area. Should scroll automatically.
- A little bit unusual "Look&Feel". Had some trouble with the "Drag'n Drop".
- Connections could have started directly from the links on the pages.
- "Nice and tidy".
- Should be able to look at tests at a higher level and connecting them there.
- Easy to get an overview.
- Some difficult menus.
- Should be able to choose from a list of links when one is creating a link connection.
- Should be able to create a start page when one defines a test, not before.
- A little bit too much "mouse work". Should be able to use more keyboard shortcuts.
- Should be able to copy and paste elements.

- It gives a very nice graphical overview of all the tests.

What do you think of AutAT's NamesAndURL editor (its strengths, weaknesses and opportunities)?

- Should auto save.
- Not easy to understand right away how it should be linked with the rest of the system.
- Should not have to click "Add" for adding a new element, but rather just start to edit in the table.
- Should be able to create a new start point when a new test is create.
- "Works fine".
- The "Add" button should be on top and more visible.

What do you think of AutAT's GraphicalTest editor (its strengths, weaknesses and opportunities)?

- Should start inn edit mode so one can start to fill the text in the elements right away.
- Very nice concept, but there are a few minor bugs.
- Should be possible to edit with double click in the figure.
- "Works fine".
- Should automatically return to being a pointer when something is added.
- The tool should still be selected when something is added.
- I didn't care much for the colors.
- Should be able to right click on a page that one has created and add a link, a form, etc.

What do you think of AutAT's Property view (its strengths, weaknesses and opportunities)?

- Takes up a lot of space.
- Should not have a name on a form.
- Too wide.
- A little bit difficult to write test.
- Should work more like an Excel spreadsheet.
- Didn't use it much.

- Thinks it was easy for getting an overview and pretty self explanatory.
- A little bit too much “mousing”.

Other comments about AutAT

- Should have ”tree structured” tests.
- Should be possible to “Copy & Paste”.
- Very good prototype.
- “Nice and Nice and Nice”.
- Can probably do much for automatic acceptance testing web pages.
- Some problems with form input connection (had to add values after the connection was established).

E.26.3 Comparison: FitNesse versus AutAT

Other comments about AutAT versus FitNesse

- AutAT is a better solution, but I do not like that it requires an IDE.
- The reason it was somewhat hard to learn was that Eclipse was somewhat difficult.
- The advantage with FitNesse is that is easy to edit with “Cut & Paste”.
- A lot easier to get an overview with a graphical user interface.
- AutAT is a lot more customer friendly, but I am still not sure that it is something that I would have given to a customer.
- The difference between AutAT and FitNesse would probably have been greater with other user groups than programmers.
- “AutAT Rules”.

Appendix F

XML schemas

The XML schemas used in AutAT. Listing F.1 shows the start point format, while Listing F.2 shows the format for a test.

Listing F.1: XML schema for start points

```
1 <?xml version="1.0" ?>
2 <xs:schema
3   xmlns:xs="http://www.w3.org/2001/XMLSchema"
4   targetNamespace="http://autat.sourceforge.net"
5   xmlns="http://autat.sourceforge.net"
6   elementFormDefault="qualified">
7
8   <!-- type for the mapping elements -->
9   <xs:complexType name="mappingType">
10    <xs:attribute name="id" type="xs:string" use="required" />
11    <xs:attribute name="name" type="xs:string" use="required" />
12    <xs:attribute name="url" type="xs:string" use="required" />
13  </xs:complexType>
14
15  <!-- type for the collection of mapping elements -->
16  <xs:element name="urlMappings" >
17    <xs:complexType>
18      <xs:sequence>
19        <xs:element name="mapping" type="mappingType" minOccurs
20          ="0" maxOccurs="unbounded" />
21      </xs:sequence>
22    </xs:complexType>
23  </xs:element>
24 </xs:schema>
```

Listing F.2: XML schema for tests

```
1 <?xml version="1.0"?>
2 <xs:schema
3   xmlns:xs="http://www.w3.org/2001/XMLSchema"
4   targetNamespace="http://autat.sourceforge.net"
5   xmlns="http://autat.sourceforge.net"
6   elementFormDefault="qualified">
7
8 <!-- type for the startPoint -->
9 <xs:complexType name="startPointType">
10   <xs:attribute name="id" type="xs:string" use="required" />
11 </xs:complexType>
12
13 <!-- type for connectionPoint -->
14 <xs:complexType name="connectionPointType">
15   <xs:sequence>
16     <xs:element name="startPoint" type="startPointType"
17       minOccurs="0" maxOccurs="1"/>
18   </xs:sequence>
19 </xs:complexType>
20
21 <!-- type for link element: linkType -->
22 <xs:complexType name="linkType">
23   <xs:simpleContent>
24     <xs:extension base="xs:string">
25       <xs:attribute name="not" type="xs:boolean" />
26     </xs:extension>
27   </xs:simpleContent>
28 </xs:complexType>
29
30 <!-- type for text element: textType -->
31 <xs:complexType name="textType">
32   <xs:simpleContent>
33     <xs:extension base="xs:string">
34       <xs:attribute name="not" type="xs:boolean" />
35     </xs:extension>
36   </xs:simpleContent>
37 </xs:complexType>
38
39 <!-- type for form types: formFieldType -->
40 <xs:complexType name="formFieldType">
41   <xs:attribute name="id" type="xs:string" />
42   <xs:attribute name="name" type="xs:string" />
43 </xs:complexType>
44
45 <!-- type for form element: formType -->
46 <xs:complexType name="formType">
47   <xs:choice minOccurs="0" maxOccurs="unbounded">
48     <xs:element name="textField" type="formFieldType" />
49     <xs:element name="textArea" type="formFieldType" />
50     <xs:element name="password" type="formFieldType" />
51     <xs:element name="button" type="formFieldType" />
52     <xs:element name="submit" type="formFieldType" />
53   </xs:choice>
54   <xs:attribute name="id" type="xs:string" />
55   <xs:attribute name="name" type="xs:string" />
```



```

56     <xs:attribute name="not" type="xs:boolean" />
57 </xs:complexType>
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="form" type="formType" />
65     </xs:choice>
66 </xs:complexType>
67
68
69 <!-- type for a single page: pageType -->
70 <xs:complexType name="pageType">
71     <xs:sequence>
72         <xs:element name="title" type="xs:string" />
73         <xs:element name="elements" type="elementsType" />
74     </xs:sequence>
75     <xs:attribute name="id" type="xs:string" use="required" />
76     <xs:attribute name="xPos" type="xs:integer" use="required" />
77     <xs:attribute name="yPos" type="xs:integer" use="required" />
78 </xs:complexType>
79
80 <!-- type for a list of pages: pagesType -->
81 <xs:complexType name="pagesType">
82     <xs:sequence>
83         <xs:element name="page" type="pageType" maxOccurs="
            unbounded" />
84     </xs:sequence>
85 </xs:complexType>
86
87 <!-- type for a single aspect: aspectType -->
88 <xs:complexType name="aspectType">
89     <xs:sequence>
90         <xs:element name="title" type="xs:string" />
91         <xs:element name="elements" type="elementsType" />
92     </xs:sequence>
93     <xs:attribute name="id" type="xs:string" use="required" />
94     <xs:attribute name="xPos" type="xs:integer" use="required" />
95     <xs:attribute name="yPos" type="xs:integer" use="required" />
96 </xs:complexType>
97
98 <!-- type for a list of aspects: aspectsType -->
99 <xs:complexType name="aspectsType">
100     <xs:sequence>
101         <xs:element name="aspect" type="aspectType" minOccurs="0"
            maxOccurs="unbounded" />
102     </xs:sequence>
103 </xs:complexType>
104
105 <!-- type for simple transitions: simpleTransitionType -->
106 <xs:complexType name="simpleTransitionType">
107     <xs:attribute name="from" type="xs:string" use="required" />
108     <xs:attribute name="to" type="xs:string" use="required" />
109 </xs:complexType>
110

```

```
111 <!-- type for link transition: linkTransitionType -->
112 <xs:complexType name="linkTransitionType">
113   <xs:attribute name="from" type="xs:string" use="required" />
114   <xs:attribute name="to" type="xs:string" use="required" />
115   <xs:attribute name="text" type="xs:string" use="required" />
116 </xs:complexType>
117
118
119 <!-- type for form input values -->
120 <xs:complexType name="formInputValue">
121   <xs:attribute name="id" type="xs:string" use="required" />
122   <xs:attribute name="value" type="xs:string" use="required" />
123 </xs:complexType>
124
125 <!-- type for form transition: formTransitionType -->
126 <xs:complexType name="formTransitionType">
127   <xs:choice minOccurs="0" maxOccurs="unbounded">
128     <xs:element name="textField" type="formInputValue" />
129   </xs:choice>
130   <xs:attribute name="from" type="xs:string" use="required" />
131   <xs:attribute name="to" type="xs:string" use="required" />
132   <xs:attribute name="formId" type="xs:string" use="required" />
133 </xs:complexType>
134
135 <!-- type for aspect transition: aspectTransitionType -->
136 <xs:complexType name="aspectTransitionType">
137   <xs:attribute name="from" type="xs:string" use="required" />
138   <xs:attribute name="to" type="xs:string" use="required" />
139 </xs:complexType>
140
141 <!-- type for a list of transitions -->
142 <xs:complexType name="transitionsType">
143   <xs:choice minOccurs="0" maxOccurs="unbounded">
144     <xs:element name="simpleTransition" type="
145       simpleTransitionType" />
146     <xs:element name="linkTransition" type="linkTransitionType"
147       />
148     <xs:element name="formTransition" type="formTransitionType"
149       />
150     <xs:element name="aspectTransition" type="
151       aspectTransitionType" />
152   </xs:choice>
153 </xs:complexType>
154
155 <!-- the test type, base element in the test documents -->
156 <xs:element name="test">
157   <xs:complexType>
158     <xs:sequence>
159       <xs:element name="description" type="xs:string" />
160       <xs:element name="connectionPoint" type="
161         connectionPointType" />
162       <xs:element name="pages" type="pagesType" />
163       <xs:element name="aspects" type="aspectsType" />
164       <xs:element name="transitions" type="transitionsType" /
165       >
166     </xs:sequence>
167   </xs:complexType>
168 </xs:element>
```

```
162     <xs:attribute name="id" type="xs:string" use="required" />
163     <xs:attribute name="name" type="xs:string" />
164   </xs:complexType>
165 </xs:element>
166
167 </xs:schema>
```

Bibliography

- [1] David Astels, Granville Miller, and Miroslav Novak. *A Practical Guide to eXtreme Programming*. Prentice Hall PTR, first edition, 2002.
- [2] Robert V. Binder. *Testing Object Oriented Systems*. Addison-Wesley, first edition, October 1999.
- [3] Eric Braude. *Software design: from programming to architecture*. John Wiley & Sons, Inc, first edition, 2004.
- [4] Eric M. Burke and Brian M. Coyner. *Java Extreme Programming Cookbook*. O'Reilly, first edition, March 2003.
- [5] Lisa Crispin and Tip House. *Testing Extreme Programming*. Pearson Education, 2003.
- [6] Tore Dybå. *Enabling Software Process Improvement: An Investigation of the Importance of Organizational Issues*. PhD thesis, Norwegian University of Science and Technology, 2000.
- [7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: elements of reusable object oriented software*. Addison-Wesley Publishing Company, first edition, March 1994.
- [8] GEF. GEF API – can be found within the GEF SDK. <http://download.eclipse.org/tools/gef/downloads/drops/R-3.0.1-200408311615/GEF-SDK-3.0.1.zip>, 2005.
- [9] Graphical Editing Framework. Graphical Editing Framework website. <http://www.eclipse.org/gef>, 2005.
- [10] Paul Hamill. *Unit Test Frameworks*. O'Reilly, first edition, November 2004.
- [11] Randy Hudson and Pratik Shah. GEF In Depth tutorial. <http://www.eclipse.org/gef/reference/GEF>
- [12] John Hunt. *Guide to the Unified Process featuring UML, Java and design patterns*. Springer-Verlag, first edition, 2003.
- [13] Marnie L. Hutcheson. *Software Testing Fundamentals: Methods and Metrics*. Wiley, 2003.

- [14] Craig Larman. *Applying UML and patterns: an introduction to object-oriented analysis and design and Unified Process*. Prentice Hall PTR, second edition, 2002.
- [15] Bo Majewski. A Shape Diagram Editor. December 2004.
- [16] Glenford J Myers, Tom Badgett, Todd M Thomas, and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, second edition, November 2004.
- [17] Eclipse RCP. Eclipse Rich Client Platform. <http://www.eclipse.org/rcp>, May 2005.
- [18] Trygve Reenskaug. The Model-View-Controller(MVC): Its Past and Present. http://heim.ifi.uio.no/~trygver/2003/javazone-jaoo/-MVC_pattern.pdf, 2003.
- [19] Tor Stålhane. SPIQ: Etablering av Måleplaner. *SPIQ notat*, June 1998.
- [20] Tore Dybå and Kari Juul Wedde and Tor Stålhane and Nils Brede Moe and Reidar Conradi and Torgeir Dingsøy and Dag Sjøberg and Magne Jørgensen. *SPIQ: Metodehåndbok*. 3 edition, 2000.
- [21] Rini van Solingen and Egon Berghout. *The Goal/Question/Metric Method: a practical guide for quality improvement of software development*. McGraw-Hill Publishing Company, 1999.
- [22] Hans van Vliet. *Software Engineering: Principles and Practice*. John Wiley & Sons, second edition, August 2000.
- [23] Kent Beck with Cynthia Andres. *Extreme Programming Explained: Embrace Change*. Addison-Wesley: Pearson Education, second edition, November 2004.
- [24] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, 2000.