# Semi-automatic Test Case Generation

**Olav Undheim**

Master of Science in Computer Science
Submission date: June 2011
Supervisor: Tor Stålhane, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

# PROBLEM DESCRIPTION

Requirements in CESAR can be described using a set of templates called boilerplates. The information that can be obtained using a combination of requirements written in boilerplates and a domain ontology opens several opportunities for automatic and semi-automatic analysis.

This project focuses on using this information to generate tests based on a selected test strategy. The following activities are needed:

- Study the tools and methods currently applied in the CESAR project
- Study other published tools and methods relevant to the problem – (semi) automatic selection of test strategy and test cases
- Build a tool prototype and test it out on students and – time allowing – in an industrial environment.

Assignment given: 17. January 2011
Supervisor: Tor Stålhane, IDI

# ABSTRACT

In the European research project CESAR, requirements can be specified using templates called boilerplates. Each statement of a requirement consists of a boilerplate with inserted values for the attributes of the boilerplate. By choosing attribute values from a domain ontology, a consistent language can be achieved. This thesis seeks to use the combination of boilerplates and a domain ontology in a semi-automatic test generation process.

There are multiple ways to automate the test generation process, with various degrees of automation involved. One option is to use the boilerplates and the domain ontology to create a test model that can be used to generate tests. Another option is to use the information from the domain ontology to assist the user when he creates tests. In this thesis, the latter option is investigated and a tool named WikiTest is developed.

WikiTest uses Semantic MediaWiki and Semantic Forms to utilize the ontology and assist the user in the test creation process. Using a Cucumber syntax, the tests can be specified in a relatively free format that does not sacrifice the ability to automate test execution. An experiment is conducted, where the results show that WikiTest is easier to use and leads to a higher test case quality than the alternatives can do. Being able to inspect the domain ontology while creating tests did not give the same results as when the ontology was integrated directly in the tool.

**Keywords:** Acceptance test driven development, Boilerplates, Cucumber, Domain ontology, Test generation, Semantic MediaWiki

# PREFACE

This thesis is the result of the Master Thesis course TDT4900, conducted during the Spring of 2011. The course belongs to the section for Program and Information Systems at the Norwegian University of Science and Technology (NTNU). The work of this thesis is a continuation of the specialization course TDT4520 conducted during the Fall of 2010.

I would like to thank my supervisor, Tor Stålhane, for his valuable help and input throughout the project. Thanks to the students who participated in the experiment, and the professors and PhD students who took part in the focus group. Special thanks are given to my fellow students at Fiol for making the semester extra enjoyable.

Trondheim, June 11, 2011

---

Olav Undheim

# CONTENTS

**Contents** xi

**List of Figures** xv

**List of Tables** xvii

**Abbreviations** xix

**1 Introduction** 1
- 1.1 Motivation 1
- 1.2 Project Context 2
- 1.3 Problem Description 2
- 1.4 Project Scope 3
- 1.5 Report Outline 3

**2 Research Agenda** 5
- 2.1 Research Questions 5
- 2.2 Research Methodology 6

**I Background** 9

**3 Software Requirements** 11
- 3.1 Requirements in Software Engineering 11
- 3.2 Boilerplates 12
- 3.3 Domain Ontologies 14

**4 Software Testing** 19
- 4.1 Test Methods 20
- 4.2 Test Coverage 20

4.3 Test Levels . . . 21
4.4 Test Approach for Further Work . . . 24
4.5 Cucumber . . . 27

**5 Technology Platform** **31**
5.1 Technology Platform Options . . . 31
5.2 Wiki Comparison . . . 34
5.3 Semantic MediaWiki . . . 36

**II Development of a Tool** **39**

**6 Tool Requirements** **41**
6.1 Early Outlook . . . 41
6.2 Focus Group . . . 43
6.3 User Stories . . . 47

**7 Tool Implementation** **51**
7.1 Setup and Installation . . . 51
7.2 Importing a Domain Ontology . . . 54
7.3 Creating a Feature . . . 55
7.4 Autocomplete . . . 59
7.5 Creating a Domain Concept . . . 60
7.6 Navigation . . . 62
7.7 Missing Tests . . . 63
7.8 Ontology Browser . . . 64

**III Experiment** **65**

**8 Definition** **67**
8.1 The GQM Process . . . 67
8.2 Goal . . . 68
8.3 Questions . . . 69
8.4 Metrics . . . 72
8.5 GQM Summary . . . 77

**9 Planning** **79**
9.1 Context Selection . . . 79
9.2 Hypotheses Formulation . . . 80
9.3 Variables Selection . . . 82
9.4 Selection of Subjects . . . 82

9.5 Experiment Design . . . . . . . . . . . . . . . . . . . . . . . . 82
9.6 Instrumentation . . . . . . . . . . . . . . . . . . . . . . . . . . 83
9.7 Validity Evaluation . . . . . . . . . . . . . . . . . . . . . . . . 84

**10 Operation** **93**
10.1 Preparation . . . . . . . . . . . . . . . . . . . . . . . . . . . 93
10.2 Execution . . . . . . . . . . . . . . . . . . . . . . . . . . . . 93
10.3 Data Validation . . . . . . . . . . . . . . . . . . . . . . . . . 94

**11 Data Analysis** **95**
11.1 Measurements . . . . . . . . . . . . . . . . . . . . . . . . . . 95
11.2 Hypotheses Testing . . . . . . . . . . . . . . . . . . . . . . . 111
11.3 Summary of Hypotheses . . . . . . . . . . . . . . . . . . . . . 117

**12 Interpretation** **119**
12.1 Participant Feedback . . . . . . . . . . . . . . . . . . . . . . 119
12.2 Validity Discussion . . . . . . . . . . . . . . . . . . . . . . . 122
12.3 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . 123

**IV Evaluation** **125**

**13 Results** **127**
13.1 Approach Selected . . . . . . . . . . . . . . . . . . . . . . . . 127
13.2 Tools and Methods . . . . . . . . . . . . . . . . . . . . . . . . 128
13.3 Tool Requirements . . . . . . . . . . . . . . . . . . . . . . . . 128
13.4 Empirical Evaluation . . . . . . . . . . . . . . . . . . . . . . . 129

**14 Discussion** **131**
14.1 Further Work . . . . . . . . . . . . . . . . . . . . . . . . . . . 134

**15 Conclusion** **135**

**References** **142**

**Appendices**

**A Wiki Code** **A-1**

**B Experiment Data** **B-1**

# LIST OF FIGURES

3.1 Different levels of requirements . . . . . . . . . . . . . . . . . . 12
3.2 Boilerplates in DOORS . . . . . . . . . . . . . . . . . . . . . . . 13
3.3 Data property (left) and object property (right) . . . . . . . . . 15
3.4 Annotations in Protègè . . . . . . . . . . . . . . . . . . . . . . . 16
3.5 A boilerplate for a steam pressure requirement . . . . . . . . . . 17

4.1 The V-model in software development . . . . . . . . . . . . . . . 22
4.2 Generate abstract tests directly . . . . . . . . . . . . . . . . . . 24
4.3 Generate test model . . . . . . . . . . . . . . . . . . . . . . . . . 24
4.4 Dimensions of model-based testing . . . . . . . . . . . . . . . . 25
4.5 TMap test lifecycle . . . . . . . . . . . . . . . . . . . . . . . . . 27
4.6 Process overview with Cucumber . . . . . . . . . . . . . . . . . 27
4.7 Cucumber start game . . . . . . . . . . . . . . . . . . . . . . . . 28

7.1 Semantic Gardening import . . . . . . . . . . . . . . . . . . . . 54
7.2 Composition of the feature form . . . . . . . . . . . . . . . . . . 56
7.3 Visualization of an example feature . . . . . . . . . . . . . . . . 58
7.4 Edit a feature using a form . . . . . . . . . . . . . . . . . . . . . 58
7.5 Ontology browser . . . . . . . . . . . . . . . . . . . . . . . . . . 64

8.1 The GQM hierarchy . . . . . . . . . . . . . . . . . . . . . . . . 67
8.2 GQM tree . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77

11.1 Programming experience . . . . . . . . . . . . . . . . . . . . . . 96
11.2 Wiki experience . . . . . . . . . . . . . . . . . . . . . . . . . . . 97
11.3 Written automatic test before . . . . . . . . . . . . . . . . . . . 98
11.4 Experience with Cucumber . . . . . . . . . . . . . . . . . . . . . 99
11.5 Feature coverage . . . . . . . . . . . . . . . . . . . . . . . . . . 100
11.6 Scenario completeness . . . . . . . . . . . . . . . . . . . . . . . 100
11.7 Syntactic correctness . . . . . . . . . . . . . . . . . . . . . . . . 101
11.8 Ambiguity score . . . . . . . . . . . . . . . . . . . . . . . . . . . 101

11.9 Verifiability . . . 102
11.10 Ontology makes the domain easier to understand . . . 102
11.11 Ontology makes it easier to write tests . . . 103
11.12 Steam boiler help . . . 103
11.13 Difficulty of using the Cucumber syntax . . . 104
11.14 Cucumber help . . . 105
11.15 Rather write tests in free format . . . 105
11.16 Ease of use . . . 106
11.17 More help for WikiTest . . . 106
11.18 Overview of tests . . . 107
11.19 User interface satisfaction . . . 108
11.20 Time used at the assignment . . . 109
11.21 Enough time to do the assignment . . . 109
11.22 Time used per feature . . . 110
11.23 Time used per scenario . . . 110

12.1 Ishikawa diagram for group A . . . 120
12.2 Ishikawa diagram for group B . . . 121
12.3 Ishikawa diagram for group C . . . 122

# LIST OF TABLES

5.1 Comparison of wikis . . . . . . . . . . . . . . . . . . . . . . . . 35

7.1 Software in the minor installation . . . . . . . . . . . . . . . . . 52
7.2 Software in the full installation . . . . . . . . . . . . . . . . . . 52

8.1 Summary of goal, questions and metrics . . . . . . . . . . . . . 77

9.1 High priority validity threats . . . . . . . . . . . . . . . . . . . . 90
9.2 Medium priority validity threats . . . . . . . . . . . . . . . . . . 91

11.1 Summary of extra measurements . . . . . . . . . . . . . . . . . 116
11.2 $H_0$-hypotheses that could not be rejected . . . . . . . . . . . . 117
11.3 Alternative hypotheses accepted . . . . . . . . . . . . . . . . . . 118

13.1 Groups in the experiment . . . . . . . . . . . . . . . . . . . . . 129

B.1 Average points per feature between the three groups . . . . . . B-2
B.2 Average points per feature between text editor and wiki . . . . B-2
B.3 Average points per feature between no ontology and ontology . B-3
B.4 Points grouped by experience . . . . . . . . . . . . . . . . . . . B-3
B.5 Points grouped by experience . . . . . . . . . . . . . . . . . . . B-4
B.6 Points grouped by experience . . . . . . . . . . . . . . . . . . . B-4
B.7 Syntactic correctness between the three groups . . . . . . . . . B-5
B.8 Syntactic correctness between text editor and wiki . . . . . . . B-5
B.9 Number of subjects who used at least one Cucumber backgroundB-6
B.10 Number of subjects who used at least one Cucumber backgroundB-6
B.11 Ontology makes the domain easier to understand . . . . . . . B-7
B.12 Overview of tests between the three groups . . . . . . . . . . . B-7
B.13 Overview of tests between text editor and wiki . . . . . . . . . B-8
B.14 User interface satisfaction between the three groups . . . . . . B-8
B.15 User interface satisfaction between text editor and wiki . . . . B-9

B.16 Ease of use . . . . . . . . . . . . . . . . . . . . . . . . . . . . . B-9
B.17 Efficiency per scenario between the three groups . . . . . . . . B-10
B.18 Efficiency per scenario between text editor and wiki . . . . . . B-10
B.19 Efficiency per scenario between no ontology and ontology . . . B-11
B.20 Feature completion score between the three groups . . . . . . B-11
B.21 Feature completion score between text editor and wiki . . . . B-12
B.22 Scenario completeness score between the three groups . . . . . B-12
B.23 Scenario completeness score between text editor and wiki . . . B-13
B.24 Ambiguity score between the three groups . . . . . . . . . . . B-13
B.25 Ambiguity score between text editor and wiki . . . . . . . . . B-14
B.26 Verifiability score between the three groups . . . . . . . . . . . B-14
B.27 Verifiability score between text editor and wiki . . . . . . . . . B-15
B.28 Ontology makes it easier to write test . . . . . . . . . . . . . . B-15
B.29 Steam boiler help . . . . . . . . . . . . . . . . . . . . . . . . . . B-16
B.30 Difficulty of using the Cucumber syntax . . . . . . . . . . . . B-16
B.31 Difficulty of using the Cucumber syntax . . . . . . . . . . . . B-17
B.32 Rather write tests in free format . . . . . . . . . . . . . . . . . B-17
B.33 Cucumber help . . . . . . . . . . . . . . . . . . . . . . . . . . . B-18
B.34 Cucumber help . . . . . . . . . . . . . . . . . . . . . . . . . . . B-18

# ABBREVIATIONS

| | |
|---|---|
| AGEDIS | Automated Generation and Execution of Test Suites for Distributed Component-based Software |
| ANOVA | Analysis of variance |
| ATDD | Acceptance Test Driven Development |
| BAT | Business Acceptance Testing |
| BDD | Behavior Driven Development |
| CESAR | Cost-efficient methods and processes for safety relevant embedded systems |
| D-MINT | Deployment of Model-Based Technologies to Industrial Testing |
| FIT | Framework for Integrated Test |
| GNLQ | A tool created in the CESAR project |
| GQM | Goal, question, metric |
| IDE | Integrated Development Environment |
| MBT | Model-Based Testing |
| MOGENTES | Model-based Generation of Tests for Dependable Embedded Systems |
| NTNU | Norwegian University of Science and Technology |
| OWL | Web Ontology Language |
| RCP | Rich Client Platform |

| | |
|---|---|
| RMM | Requirement Meta Model |
| RSL | Requirement Specification Language |
| SMW | Semantic MediaWiki |
| SRS | Software Requirements Specification |
| SUT | System Under Test |
| TDD | Test Driven Development |
| TMap | Test Management Approach |
| UAT | User Acceptance Testing |
| UML | Unified Modeling Language |

# CHAPTER 1

# INTRODUCTION

This chapter explains the context and foundation for the rest of the thesis. First, the motivation and context sections define the background for the project. Then, the problem definition states the main goals of the thesis, and the scope definition explains what is chosen to be within the boundaries of the problem definition and what will be left out. This chapter ends with a report outline, which describes the structure of the rest of the thesis.

## 1.1 Motivation

In agile software development, acceptance tests are used as a passing criteria for a requirement. The acceptance tests are supposedly written by the customer, although in many cases the customer lacks the time or knowledge to do so. A situation where a customer without programming knowledge is able to write tests is highly desirable. In this regard, tool support will be important to reduce the effort it takes for a customer to create automatic acceptance tests. Automatic test case generation has been a wishful dream for many software developers over the past decades. Even as early as in the 1970s, software developers and researchers wrote articles about this possibility [1, 2]. Finding a way to automate parts of the test creation process would be beneficial to the software development process.

## 1.2 Project Context

CESAR (**C**ost-**e**fficient methods and processes for **sa**fety **r**elevant embedded systems) is a European project that focuses on software development in industrial automation and transportation domains (automotive, aerospace and rail). One of the main goals is to reduce software development time and effort by 30-50%. An important focus area is requirement formalization in order to facilitate automatic test and code generation. One way to achieve this is to use methods for requirement engineering that express requirements in a formalized way. Important topics are domain ontologies, requirement specification languages (RSL) with an underlying requirements meta model (RMM), automatic test code generation and tool development. This thesis is based on the CESAR's goals of innovations in the topics of test definitions and test execution. Two of the goals in CESAR reads as follows [3, p. 34]:

- *Automatic test specification from the SRS:* For each system requirement a test should be automatically specified, in order to determine whether the system satisfies the requirement
- *Customizable test generation:* It should be possible to define rules, which allow the user to generate a customized set of tasks

The execution of tests, comparison of the actual output to a test oracle, and analysis of the outcome are also parts of the goals.

## 1.3 Problem Description

This thesis seeks to explore the creation of executable acceptance tests based on domain knowledge and requirements. A main goal is to create a tool that can use requirements, in the form of boilerplates [4][5], and a domain ontology [6], to automate parts of the test creation process. As a part of this goal, it will be important to empirically validate the usefulness of the automation process selected.

## 1.4 Project Scope

The process chain from requirements engineering to acceptance testing is a long process which involves stakeholders with different priorities. This thesis seeks to explain this process and identify where improvements can be made. The focus will be at the stage where requirements are already specified, and how acceptance tests can be specified for the requirements and afterwards executed. There are several ways automation could be used to aid the test specification process, but the scope will be on a collaboration tool which uses a domain ontology to aid the users. Total automation of test case generation is not part of the scope, and neither is creating a tool for the actual execution of the tests, i.e. a test runner. Time will rather be spent studying how tests can be created and later executed in an already existing test execution tool. In other words, a tool should be made for the specification of tests, but not for running the tests.

## 1.5 Report Outline

This thesis is organized as follows. Chapter 2 states the research questions to be considered. Then follows the four main parts of the thesis. The first part is a background of topics related to the research questions. The second part describes the implementation of a tool for acceptance tests. The third part explains how an experiment was designed and the operation of the experiment. The fourth and last part describes the results and discussion of the thesis, followed by the conclusion.

# CHAPTER 2

# RESEARCH AGENDA

This chapter describes the research which is to be carried out in the thesis. The first section defines the research questions, while the second section discuss the research methods applicable to software engineering, and which methods will be used to answer the research questions.

## 2.1 Research Questions

This thesis is a continuation of the specialization project conducted during the Fall of 2010 [7], which primarily was a literature study. The research agenda of this thesis is to continue the work, but now with focus on tool implementation and empirical validation. The following research questions are posed:

**RQ1: Which of the earlier identified approaches using boilerplates and domain ontologies should be used for further work?**
In the work leading up to this thesis, a thorough examination of domain ontologies and boilerplates in the context of test automation was performed [7]. In the end, three approaches for the use of boilerplates and ontologies were proposed. The first approach was to create a tool to create abstract test cases directly from the boilerplates and the ontology. The second was to create a behavioral test model, and then use this test model as input to an existing model-based testing (MBT) tool. The third was to create a tool which use the boilerplates and ontology to provide assistance in the test creation process. The first research question is to further explore these three approaches and select one of them as the basis for the rest of the thesis.

**RQ2: For the approach selected in RQ1, which relevant tools and methods already exist?**
This research question seeks to identify which tools exist and which can be used in the approach. Identifying existing tools and methods is important both in order to avoid duplication of work, and to be able to identify where further improvements can be made. In general, open-source tools will be preferred over commercial tools because of cost issues and the possibility for further development.

**RQ3: For the approach selected in RQ1, what are the requirements for a new tool used in the approach?**
After identifying which tools and methods already exist in RQ2, it will be important to identify what features a new tool should have. Effort will be on analyzing the core needs of the users, and prioritize these for later development.

**RQ4: What is the effect of applying the tool-supported approach in an empirical setting?**
The fourth and last research question focus on evaluating the usefulness of the approach as a whole, and the tool in particular. The empirical evaluation should not take place too late in the development process, as there should be time to improve the tool based on the feedback from the experiment.

## 2.2 Research Methodology

The goal of research is to produce new knowledge. The research methods selected here will be applied to answer the research questions. This includes a study of existing literature, processes and products used in the area of automatic acceptance testing.

In [8], four research methods relevant for software engineering are described:

- **The scientific method:** After observing the world, a model is built to simulate the observation.
- **The engineering method:** First study the existing solutions, then propose changes and finally analyze the results of the changes.
- **The empirical method:** First, a model is proposed through hypothesis. Then, the model is evaluated empirically.

- **The analytical model:** First, a formal theory is proposed. Then, this theory is evaluated up against empirical observations.

We will use a mixture of the engineering method and the empirical method. The engineering method is used for RQ1, the selection of an approach to improve, and RQ2, the study of existing solutions. The changes proposed in RQ3 are also part of the engineering method. These changes are evaluated in RQ4, by the use of the empirical method.

A major part of the research will be to create a prototype tool for the process that is selected. In the act of doing this, two empirical methods will be used.

**Focus group**

The focus group is a qualitative research method which will be used in the requirement elicitation and in the early stage of the development. The focus group is a group of participants who together discuss some object, in this case the functionality of the tool to be created. The use of a focus group is used to investigate RQ3.

**Experiment**

The experiment will be used to answer RQ4. It will be performed on a group of students to evaluate the use of a domain ontology process and the tool itself. The experiment is defined around a set of hypotheses, which can be found in Chapter 9.2.

# Part I

# Background

# CHAPTER 3

# SOFTWARE REQUIREMENTS

This chapter is the first of three background chapters that provide the foundation for the rest of the thesis. Some content from the specialization project [7] is repeated in order to keep this thesis relatively autonomous. The background chapters are used to answer RQ1, the identification of different ways to use boilerplates and domain ontologies for semiautomatic test creation, and the selection of one of these approaches. The chapter starts with a general discussion about requirements in software engineering, followed by a description of boilerplates and domain ontologies.

## 3.1 Requirements in Software Engineering

In engineering, a requirement is a singular need of the system. In software engineering, requirements are split into two groups. The first group is functional requirements, which says something about what the system should do. The second group is non-functional requirements, which says something about the quality of the system, for example its reliability or performance. A more detailed classification of requirements is given in Figure 3.1 [9]. The figure shows the connections between different types of requirements. The business requirement is a high level goal, which can be divided into subgoals as seen from the user. These user level goals can be divided into more specific product level goals.

Requirement engineering, the creation of requirements, starts with an elicitation phase where the objective is to discover potential needs. The needs are converted into requirements that are analyzed and formally specified. In or-

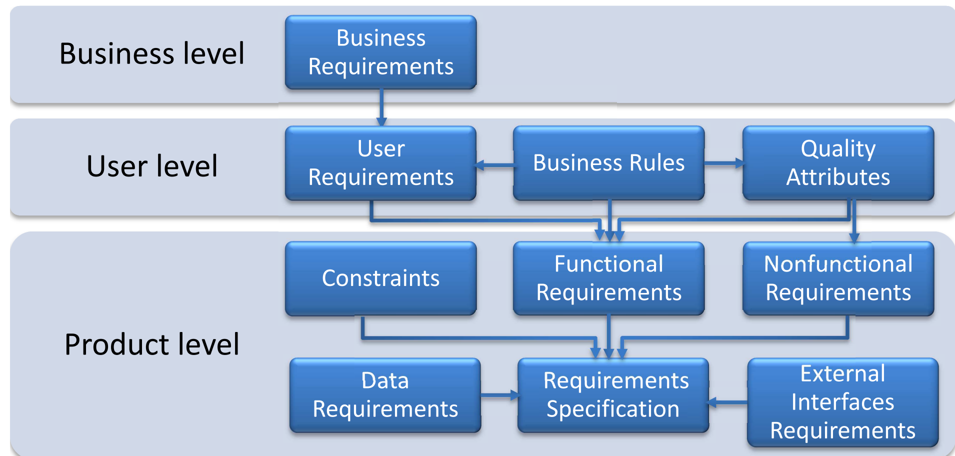


Figure 3.1: Different levels of requirements

der to be testable, a requirement needs to be specified in a precise way. Each requirement must be reviewed to see that these requirements were actually what the customer wanted. The cost of having incorrect requirements grows larger the later they are found. Finding and fixing a software problem after delivery may be a hundred times more expensive than finding and fixing it during the requirements and design phase [10]. Creating acceptance tests at the same time as the requirements are specified is a way to make sure that the requirements will in fact be possible to test.

Well formulated requirements have certain characteristics. Many checklists and questionnaires exist in the literature stating the characteristic of a good requirement. Typical characteristics are consistency, cohesion, completeness, traceability and verifiability [11].

## 3.2 Boilerplates

In the industry, requirements are often created using unstructured, plain text. Free text offers the freedom to formulate the requirements as you want, but with that, the risk of ambiguity and inconsistency increases. Boilerplates are templates to structure requirements. The requirements will still be written with a natural language, but with a semi-formal structure [4].

A simple example of a boilerplate is: “The <user> shall be able to <capability>”. Several boilerplates exist, so you can select the most appropriate for the requirement you need. Then, you fill in the attributes using your own terms. In the former example, <capability> could be changed to “rent

a book". Each requirement consists of a boilerplate with specific attributes [4]. Using boilerplates is a way to be consistent; to be precise and use the same vocabulary in all the requirements. Figure 3.2 shows how boilerplates look in DOORS, a requirement tool extended with boilerplates.

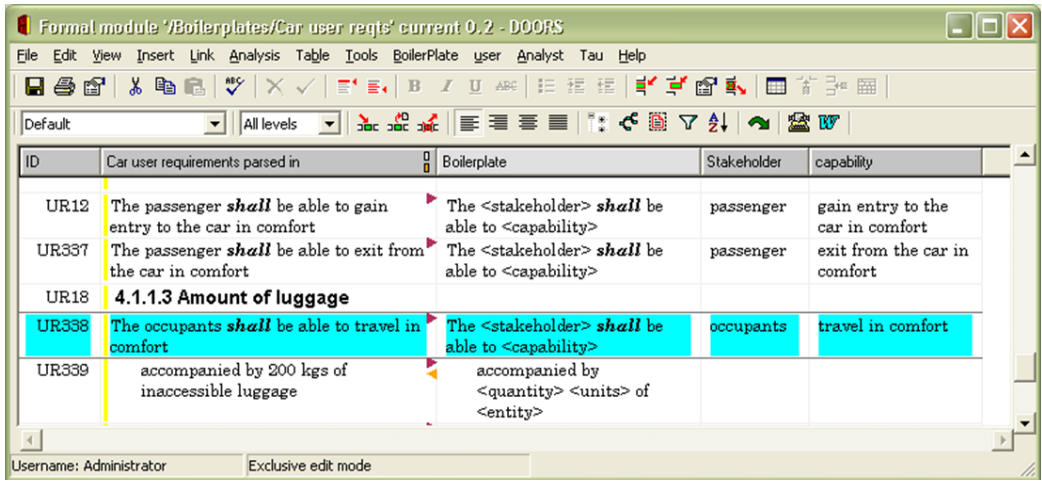


Figure 3.2: Boilerplates in DOORS

Boilerplates are similar to user stories. One difference, however, is that user stories are often written using the same, following format: "As a <user> I want to <capability> so that <business value>". With boilerplates, many more templates exist and a part of the process is to identify the template that fits the most. Attributes of boilerplates are: User, Capability, Quantity, Time Unit, Event, Operational condition, System function, Action, Entity, State, Effect. The same boilerplates can be used both for functional and non-functional requirements.

Related work has been done with more formal templates. For real-time embedded systems, as is a focus in the CESAR project, Wei-Tek Tsay et al. has investigated the possibilities of verification patterns [12][13]. An approach using a restricted English grammar provided by Konrad and Cheng [14] was evaluated in a case study with 289 informal behavioral requirements [15]. The grammar looks like natural language, but allows an automatic translation into linear time logic. Although both the approach of Wei-Tek Tsay and Konrad and Cheng are more formal than boilerplates, they give an indication of the interest in formalized requirements that can be analyzed automatically. While there were about 30 original boilerplates made by Jeremy Dick [5], in the CESAR project it was discovered that it was beneficial to add more, including the possibility to recursively merge one boilerplate with another.

This makes it possible to create boilerplates of any length, but it makes it correspondingly more difficult to use the boilerplates in automatic test generation.

## 3.3 Domain Ontologies

A domain ontology is a formal specification of conceptual knowledge in a domain [16]. In this context, the domain is the *problem domain* in which the system is going to be used. The ontology contains information about the concepts and relationships of the domain. Concepts can have different meanings and uses depending on which domain they are used in, which is why we create specific ontologies for each domain. The ontologies provide insight into the rules, constraints and axioms amongst concepts in a domain.

In [6], a domain ontology is used with the tool GNLQ, which will be further described in Section 3.3.4. With the help of this tool, the requirements are first analyzed using natural language processing, and new concepts are discovered. An analyst or a domain expert identifies the newly found concepts that are relevant and should be added to the existing ontology. These valid concepts are further defined and then added to the knowledge base. The domain ontology provides a common vocabulary which can be used to improve communication among stakeholders. It can also be used to reason over a system, checking information for completeness, consistency and correctness. From the viewpoint of requirements, a domain ontology can be used both to elicit new requirements and to check existing ones.

Protege (Protégé) is a free, open source ontology editor and knowledge-base framework[1]. Protege gives support for consistency checking ("can a class have any instances?"), classification ("is A subclass of B?") and instance classification ("which classes do an individual belong to?"). Web Ontology Language (OWL) ontologies consists of:

- Entities
  - Individuals: instances in the domain
  - Classes: sets that contain individuals
  - Properties: binary relations
- Expressions: represents complex notions, e.g. restrictions
- Axioms: statements that are asserted to be true in the domain

[1] http://protege.stanford.edu/

### 3.3.1 Data Properties

Data properties can be represented in many ways: string, int, float, boolean and so on. The restriction types that may be used are *some* (existential), *only* (universal), *min*, *max* and *exactly*. For a given property $P$, a minimum cardinality restriction specifies the number of $P$ relationships that an individual must participate in. Data properties are similar to the entity qualifiers in boilerplates. Figure 3.3 shows how a data property and an object property is represented in an ontology.

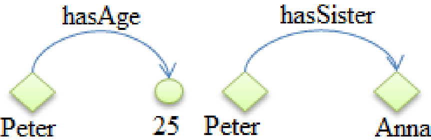


Figure 3.3: Data property (left) and object property (right)

### 3.3.2 Object Properties

Object properties specify relationships between individuals, such as "has_a" or "gets_speed_signal_from". The name of the object property may make sense to a human, but it has no syntactic function. It is possible to specify that the properties are: functional, inverse functional, transitive, symmetric, antisymmetric, reflexive and irreflexible. One possibility is to create a set of predefined names for object properties that a test tool could search for and use in a generation process. For instance, the <operational condition> in boilerplates uses the connectors "in|within|outside|between|
after|before|while". A tool could be configured to search for these terms and interpret them depending on their meaning.

### 3.3.3 Annotations

Annotations are used as extra information for a class or a relationship in the ontology. Typical annotations are "label", "comment" and "versionInfo", but it is also possible to create your own annotations. From a testing point of view, annotations such as comments can give extra information, but they lack utility for automatic test generation. Since it is possible to create your own annotations, it is possible to create specific annotations to aid the test generation process. These annotations need to be standardized, meaning that they all had the same format, so that a test tool could make use of them. Since the ontologies can be reused, this test information would have

to be general so that it could be used for different applications using the same domain ontology.

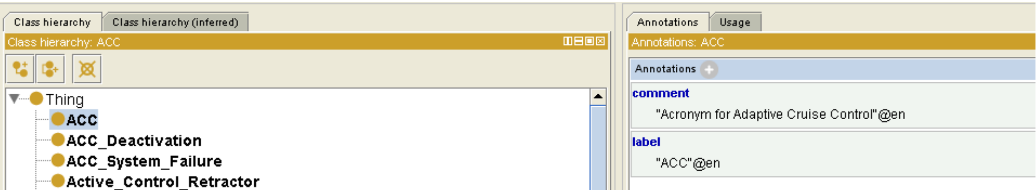


Figure 3.4: Annotations in Protègè

### 3.3.4 GNLQ

GNLQ is a "knowledge based guided requirements elicitation tool" [17]. It is developed as a plugin to Eclipse, and is a part of the CESAR project, where it has a role in requirements engineering[2]. It is described here as the output of GNLQ is likely to be used in the tool that will be created later in the thesis.

GNLQ takes an OWL domain ontology as input. Classes in the ontology, for example a sensor, can have relationships to other classes. A class may also have failure modes, e.g. commission, omission and stuck. In GNLQ, it is possible to adjust an existing domain ontology, or to create a new one from scratch. In order to add requirements, a user picks the boilerplates that fit the requirement and then select classes from the ontology as attributes to the boilerplate requirement. The output of the tool is a domain ontology in OWL and boilerplate requirements specified in XML.

Figure 3.5 shows a screen shot of a requirement in GNLQ. The boilerplate to be filled out is "if <state> greater than <quantity> then the <object> shall <action> <entity>". With attributes filled in, the boilerplate says "if Steam Pressure greater then critical pressure level then the steam boiler shall open Safety Valve"'.

[2] http://sourceforge.net/projects/gnlq/

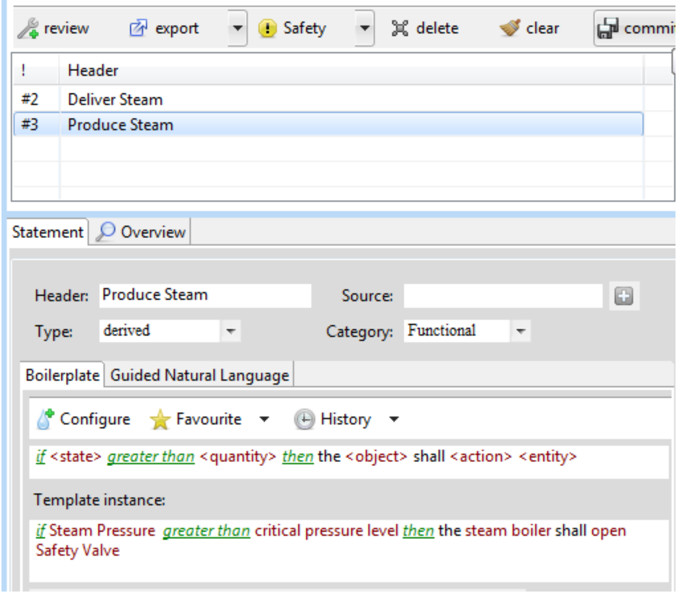


Figure 3.5: A boilerplate for a steam pressure requirement

If a tool is to be made that uses the ontology and the boilerplates made from GNLQ, then the tool could either be a standalone application or be integrated into GNLQ. In both approaches the tool would use the same input and it should thus not be difficult to go from one approach to the other. As GNLQ is still in the alpha release phase, it may be better to create a separate application at first to reduce the coupling.

# CHAPTER 4

# SOFTWARE TESTING

Software testing is the process of uncovering evidence of defects in a program. Often, a distinction is made between discovering a defect (testing), and finding the cause of the defect and fixing it (debugging) [18, p. 2]. While quality is sometimes defined as meeting requirements, testing gives an indication of the extent to which the requirements are met. A distinction is made between testing to see if the system meets the requirements, called *verification*, and testing to see if the requirements are actually what the customer wanted, called *validation*.

Quality has to be built in, not tested in. Testing is the instrument that can provide insight into the quality of an information system. Testing can seldom find all the defects in a piece of software, but will give an indication as to the quality of the software. Testing can increase trust in the product's behavior.

There are multiple terms used to express that something is a *bug*. Although the meanings are similar, there are some important distinctions [18, p. 9]. A *defect* is a flaw in the program that can lead to a *fault*, which may be undetected for a long time. A fault can lead to an *error*, which means that the system is in an erroneous state. If no corrective action is taking, the error may lead to a *failure*, which the end user can observe. There is hardly any difference between a defect and a fault, and the two terms are often used interchangeably.

There is an important difference between static testing and dynamic testing. Examples of static testing are reviews, walkthroughs and inspections, while dynamic testing is testing the software by executing it.

## 4.1 Test Methods

Tests are classified as either black box, white box or gray box, depending on what information the tests are derived from.

### 4.1.1 Black Box Testing

Black box testing (functional testing) uses only the external interface of the item under test. For a given set of input, a specific output is expected. The tester does not need to know anything about the implementation or the code. Instead, the tests are based on requirements and functionality; the only thing that matters is the external visible properties of the software under test. A negative effect of this is that a lot of different input may lead to the same execution path of the program, and are thus essentially testing the same thing, while some parts of the code may not have been tested at all.

### 4.1.2 White Box Testing

White box testing (structural testing) uses knowledge of the internal logic of the code when constructing a test. Instead of testing the functionality, the internal structure of the code is used to define test cases. This information can be used to achieve a certain test coverage. White box testing is usually done at the unit level, and debugging is always a white box activity. Both static and dynamic testing are part of white box testing.

### 4.1.3 Gray Box Testing

Gray box testing is a combination of black box and white box testing. The tester designs the tests based on some knowledge of the internal properties of the software, but the tests are executed as black box tests. An example is state diagrams, which can be used to design the tests. The advantage of gray box testing is that you can look for assumptions made by the system, and then test these, instead of trying to guess which assumptions the developer has made in the program.

## 4.2 Test Coverage

Test coverage is defined as the number of units tested divided by the total number of units. A unit can be a requirement, a code line or something else. Coverage methods differ depending of the units they measure. In general,

there are two ways to measure coverage. The first is functional based testing, which measures the program's conformance to the specification or requirements. The second is structural coverage, which measures units in the actual software code. The idea of coverage is that the quality of a test is measured by the degree that the test covers the program.

According to [19], structural coverage based on control flow is the metrics most suitable for automated collection and analysis. The metrics are similar, but have different qualities. The simple metrics, such as statement coverage, is easy to understand, collect and analyze. It does, however, not provide a good measurement of the thoroughness of the system test, as 100% statement coverage will often leave many defects undetected in the system. Other evaluation criteria are important as well, such as maintainability and the relationship between design documents and code. Some typical test coverages:

- **Statement coverage** measures the amount of individual statements that has been encountered at least once in the code.
- **Branch coverage**, also named decision coverage, checks if every branch in the program has been executed. This means that boolean expressions has been evaluated to both true and false, all outcomes of a switch statement has been covered, and so on.
- **Condition coverage**, also named predicate coverage, is the extent that boolean sub-expression has been covered.
- **Path coverage** is the number of routes through a given part of the code that has been executed.

## 4.3 Test Levels

Testing can be performed at different levels in the code. A test level is a group of test activities that are managed and executed collectively. Testing at multiple abstraction levels makes it easier to detect faults at an earlier stage of the development process. Figure 4.1 shows the V-model of software testing [20]. The left hand side shows the layers of specification and design, while the right hand shows the corresponding test levels.

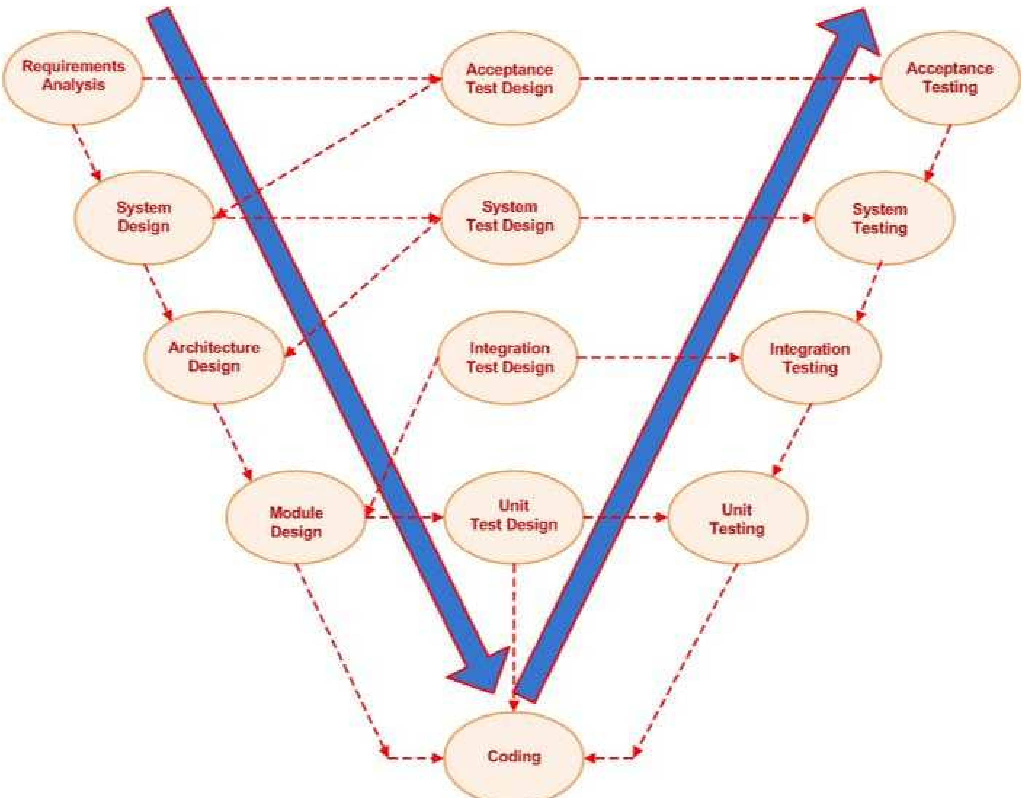


Figure 4.1: The V-model in software development

### 4.3.1 Unit Testing

In unit testing individual units of code, such as a method or a class, is tested. These tests are most often written by the developers themselves. The test cases can either be written after coding, which has been the traditional way, or before the coding, as done in Test Driven Development (TDD). A possible disadvantage of unit testing is that the test cases are written to suit the programmer's implementation, and not necessarily the specification of the system.

### 4.3.2 Integration Testing

Integration testing is done to test several units working together in collaboration. Some people also call this interface testing. The modules put together to be tested are most often coded by different developers, and it's thus important to test that the interaction between the modules works as specified.

There are several ways to perform integration testing. A bottom-up approach uses drivers, while a top-down approach uses stubs. Another approach to integration testing is the big bang method, where all the components are put together and tested at once. The main goal of the integration test level is to discover inconsistencies in the combination of units.

### 4.3.3 System Testing

System testing is performed on the system as a whole. The system is tested to check its compliance to the system's requirements. As no internal knowledge of the system is needed in this phase, a black box test method is used. Different kinds of tests focus on different aspects of the system: functional requirements, performance, scalability, reliability, usability and so on. System testing is performed by a test team, and the test cases are derived from the high level specification.

### 4.3.4 Acceptance Testing

Acceptance testing is the final test before the system is delivered to the customer. The test is performed in a simulated or real environment. There are two categories of acceptance testing. User Acceptance Testing (UAT) is performed by the customer, while Business Acceptance Testing (BAT) is undertaken within the development company to ensure that the system will pass the UAT.

### 4.3.5 Regression Testing

Regression testing is to rerun tests that passed before after doing changes to the code. Parts of the system that worked before may stop working as a side effect of a code change. If a system has a lot of tests that takes a long time to run, it is beneficial to only rerun the tests that test code that may have been affected by the changes. Regression testing is a time-consuming process, and should be automated.

## 4.4 Test Approach for Further Work

This section provides a discussion of RQ1, the selection of a test approach. The three possible approaches were defined in Chapter 2.1. The first approach is to generate abstract test cases directly from the boilerplates and the ontology. User involvement is needed both as a test oracle, and to translate the abstract tests into executable ones. The approach is visualized in Figure 4.2.

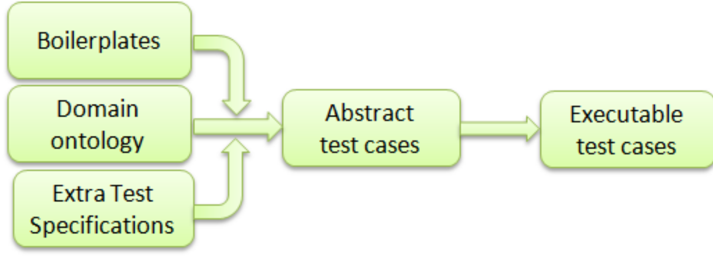


Figure 4.2: Generate abstract tests directly

The problem with this approach is the complexity of generating tests directly. An idea was to create a test pattern for each boilerplate. In other words, each boilerplate would have a statically defined way of being tested, where the boilerplate defined which test pattern would be used and the domain ontology was used to generate test data. The concepts in the domain ontology would need data or object properties for the test data to be created. This way had merit with the initial 30 boilerplates, but when boilerplate recursion and more boilerplates were added, the test pattern creation becomes a dynamic task.

The second approach is to create a behavioral test model based on the boilerplates and the ontology, and then use this test model as input to an existing MBT tool. The approach is displayed in Figure 4.3.

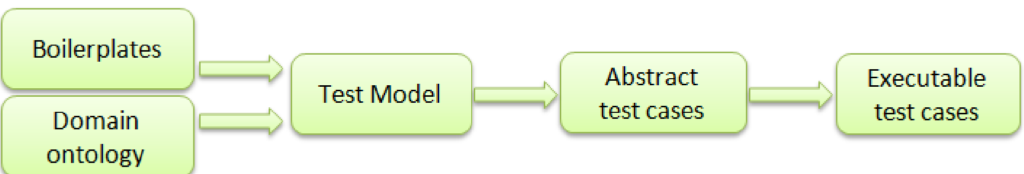


Figure 4.3: Generate test model

In order to discuss this approach, we will explain the model-based testing process. Model-based testing (MBT) has several dimensions, as shown in

Figure 4.4. MBT can be used for all test phases of a system, from unit testing to system testing. An important question of MBT approaches is how the model is specified. The model can be represented as states, sets, grammars and several other ways. A typical example is UML state diagrams, which could be extended with test annotations which is then used in the test case generation. After the test model is created, test specifications can be generated by traversing the model.

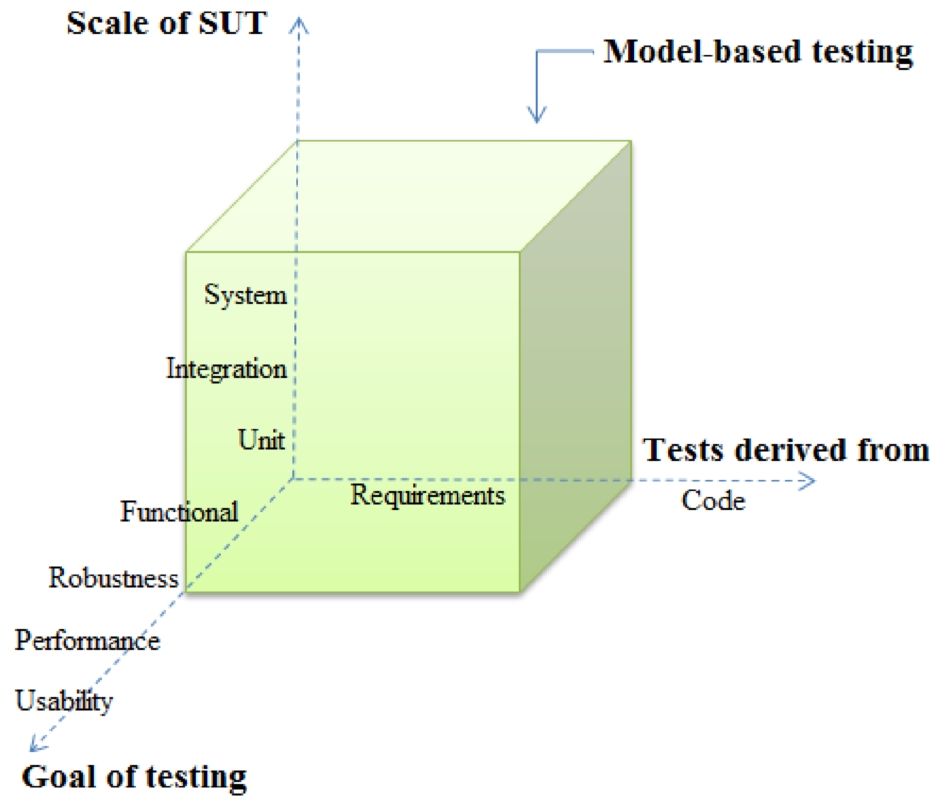


Figure 4.4: Dimensions of model-based testing

The major obstacle to the second approach is the construction of the model. It requires a lot of time and effort, and not all systems are easily modeled. Changes to the ontology or boilerplates which will lead to changes in the model will have to be done manually, and only then the tests can be regenerated automatically. This approach has merit if it is possible to do a partial automation of the test model construction based on boilerplates and the domain ontology. As discussed in [7], there has been three research projects the latest years: AGEDIS (2001-2004), D-MINT (2007-2009) and MOGENTES (2008-2010). These projects have looked at MBT tools and the construction

of such. Even though a lot of efforts have been made, the weakness of this second approach is the lack of a good MBT tool to work against.

The third approach suggested involves creating an assistance-tool that guides the user through the test creation process by using information from the domain ontology. Given an existing ontology and a set of boilerplate requirements, the tool could create partial tests based on the requirements, and have the user fill in the rest. Another possibility is that the tool identifies which domain concepts are used in a boilerplate, analyze the concepts and make test suggestions based on the attributes and relationships of each concept.

The third approach is the least automated, but has the advantage that the scope of the development task is closer to what can be achieved in a master thesis project. Most importantly, the approach does not exclude the other options of test automation. After all, if an automatic test specification technique is discovered during the development of the tool, then the technique can be incorporated as another way of aiding the user. If no ways of automatically generating tests are found, the tool will still be useful, but with more work put on the user. In an approach involving MBT tools, it is “all or nothing”, in the sense that if not a proper test model is used as a basis, then useful tests cannot be created.

The answer to RQ1 is that the third approach will be used, where the focus is on creating a tool that supports the user in the test creation process. In [7], possible test automation tools were described. Two tools, FitNesse and Cucumber, were evaluated, where Cucumber came out in front for our needs. The choice of an automation tool is important, as the test created in our approach will need to executed. A description of Cucumber and its implications for the approach will now be given.

## 4.5 Cucumber

Good tool support is essential for test automation. Different tools belong to different phases of the life-cycle, as shown in Figure 4.5 [21].

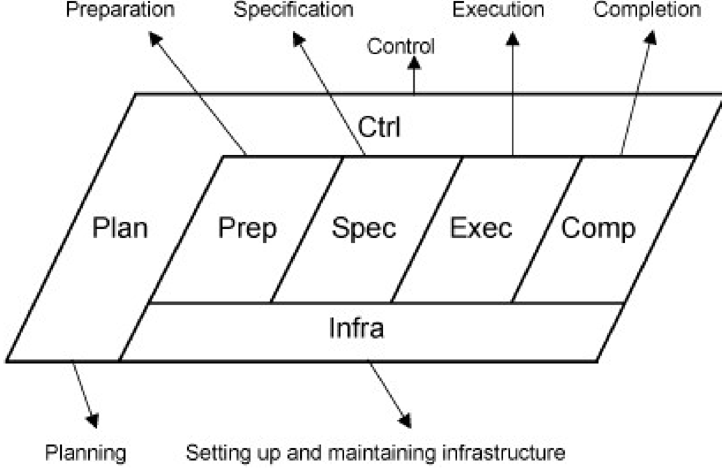


Figure 4.5: TMap test lifecycle

Cucumber is used in the *specification and execution* phases. In the approach selected for RQ1, a tool will use the information from a domain ontology to aid the user in a test specification process. The focus is on executable tests, meaning tests that can be run. Thus, the test format has to be writable and readable by customers without much IT experience, and at the same time be possible to automate. This is where Cucumber will be used.

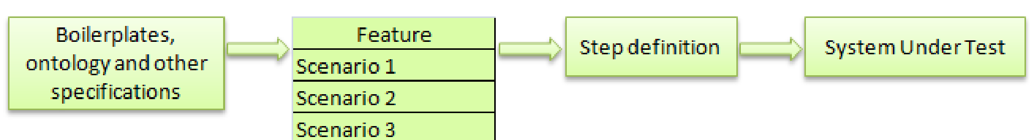


Figure 4.6: Process overview with Cucumber

The main elements of Cucumber are *features* and *scenarios*. A feature is a high-level requirement, and consists of one or many scenarios. The scenarios are acceptance tests for that requirement. The *step definitions*, the elements of a scenario, can be implemented in most programming languages. The choice of *execution tool* (e.g. JUnit) for the actual running of the tests is freely selectable. The test results are fed back to the user through Cucumber.

### 4.5.1 Boilerplate Example with Step Definitions

In Cucumber, tests (or behavior, as it is called in BDD) are written in plain text using a few simple keywords. Figure 4.7 shows a typical Cucumber example. It defines who the user is (a code-breaker), what they want to do (start a game) and why (be able to break the code). The general format in Cucumber is [22]:

- Feature: a title of the behavior (requirement)
- Story: a description of the requirement
- Scenario(s): examples of the behavior

Download cb/02/features/codebreaker_starts_game.feature

```
Feature: code-breaker starts game

  As a code-breaker
  I want to start a game
  So that I can break the code

  Scenario: start game
    Given I am not yet playing
    When I start a new game
    Then I should see "Welcome to Codebreaker!"
    And I should see "Enter guess:"
```

Figure 4.7: Cucumber start game

One way to use Cucumber with boilerplates is to have a separate feature for every boilerplate, where the story is the actual boilerplate requirement. Although this will go against the standard story-format, the story is just a text description and is not used for anything executable. It is, however, important to use the Given, When, Then, And, But keywords, which are keywords in the Gherkin domain-specific language [23]. *Given* is the context and preconditions for the scenario. *When* is what the feature is talking about - the actual action. *Then* is a check of the postconditions to see that the right thing happened in the When stage.

We will now look at how a boilerplate can be transformed into a scenario. The boilerplate requirement is: "If <the ignition is turned on by car key> while <either of the doors are opened> then <the alarm horn> shall <beep 3 times> within <2 seconds>. As a scenario, this can look like:

> *Given* either of the doors are opened
> *When* the ignition is turned on

*Then* the alarm horn beeps 3 times within 2 seconds

As we can see, it is relatively easy to transform the boilerplate requirement to a scenario test. This is because of the freedom you get when you write scenarios. In some situations, you have the same scenario many times, but only with different values being tested. By using a *scenario outline*, the scenario is written once and different values for the scenario are specified in a table.

Additionally, it is possible to write a *background* before the scenarios. A background has any number of Given steps. These steps belong to all the scenarios for the feature. This means that if you have Given steps common to all scenarios, you can put them into the background. The advantage of doing this is that the tests are clear about the data being set up. The disadvantage is that everything is common. You cannot vary some parts of the data for the different scenarios.

### 4.5.2 Wiki-style Cucumber

In order to use the extra information about concepts and relations which the domain ontology gives us, it would be useful to have a wiki for the Cucumber tests, as was done with FitNesse for Fit. At the time of writing, this has not been done, but since Cucumber is steadily changing it may be added if developers think it would be useful. However, it would be useful to incorporate the extra information from the ontologies directly where the tests are created, providing functionality such as the possibility to click on a concept in one requirement to see other requirements where the same concept is involved. The focus will thus be on creating the abstract tests and let the test engineers write the implementation of the test scripts. The actual programming language (Ruby, Java, C++ etc) will not matter since the abstract test cases are just text; the invoking of the code in step definitions is done using regular expressions.

# CHAPTER 5

# TECHNOLOGY PLATFORM

The overall goal is to create a test process with a tool that can assist in the specification of tests. This chapter takes a look at the different technology platforms that can be used. An important aspect is what current tools and methods already exist for the given technology platform.

## 5.1 Technology Platform Options

There are three main options when it comes to technology platforms. The first is to create a plug-in to Eclipse, the second is to create a standalone application, and the third is to create a web-based application.

### 5.1.1 Eclipse Plug-in

Creating an Eclipse plug-in has the advantage of being compatible with GNLQ, described in Chapter 3.3.4, which is also a plug-in to Eclipse. It is possible to extend GNLQ, or to create a separate plug-in. As GNLQ is still in the early development stage, less risk is taken by creating a separate plug-in and use the output (boilerplates and ontology) of GNLQ as input for the new plug-in.

If an Eclipse plug-in is developed, then the tests need to be specified in Eclipse. This is fine if the developers already use Eclipse, but it puts a restriction on the IDE of developers, and the customers may not be familiar with Eclipse. Fortunately, the Eclipse Rich Client Platform (RCP) can be used to create a standalone application of the plug-in, if that is needed [24].

Creating an Eclipse plug-in makes it possible to reuse existing features and functionality in Eclipse, including keyboard shortcuts and drag and drop.

### 5.1.2 Standalone Application

Developing a local-based client puts few restrictions on design and functionality. Any programming language can be used. The disadvantages is that the development have to start from scratch, although libraries may offer some of the functionality. The application needs to to be installed on every computer where it will be used.

### 5.1.3 Web Application

A web application removes the need of any installation. As the users can work with the same data, no effort is needed to share the tests. However, the complexity increases when multiple users can edit the same data on the same time. The software can be used independent of operating system, but cross-browser compatibility needs to be taken into consideration.

The major weakness of web applications is limited graphical functionality, as not everything that can be done in a standalone application can be done in a web browser. Another usability concern is the latency, which will depend on the network speed and the amount of data that is loaded. Care should be taken so that if a large domain ontology is used, it should not degrade the response time of the application.

### 5.1.4 Selection

In order to select one of the three platforms to use, it is necessary to investigate the tools and solutions that already exist for each platform. If it is possible to reuse tools or functionality, more time can be used implementing the core functionality of the new tool. As an example, the creation of a version control system for the test specifications would take too much time compared to the research benefits it would give.

Although the requirements for a new tool are described in Part 3: Development, the most important requirements were already known at the time that this investigation was performed. Users would have to write test specifications together, and be able to browse the requirements and tests. The functionality resembles what can be found in a typical requirement management tool, but with added functionality of test specification based on a

domain ontology. In addition to requirement management tools, the use of a wiki could provide the functionality we are looking for.

The functionality that could already exist is support for a) domain ontologies, b) collaboration and c) testing/Cucumber. The Eclipse based wiki called EclipseWiki[1] had potential to be used as a basis to build on. It did, however, lack in several aspects compared to what can be found in existing web based wikis. Gwtwiki[2] provides a Java Wikipedia API, but this is mainly used to convert Wikipedia syntax to HTML, and the other way around. Even though the wiki support inside of Eclipse seems to be lacking, a tool supporting Cucumber in Eclipse has been created. This tool, called QuBiT[3], is developed with Xtext and provides syntax highlighting and auto-complete on Cucumber keywords.

In the search for tools that supported either domain ontologies, collaboration or Cucumber, the potential of a wiki with support for ontologies, of which many exist, emerged. It would cover two of the three goals, and Cucumber-support could be added on top of the wiki. Most mature wikis are web-based, and the conclusion of the technology platform is that a web-based wiki with ontology support should be used.

[1] http://eclipsewiki.sourceforge.net/
[2] http://code.google.com/p/gwtwiki/
[3] https://github.com/QuBiT/cucumber-eclipse-plugin

## 5.2 Wiki Comparison

This section provides a discussion of wikis that can be used for further development. The wiki needs to have some kind of support for ontologies, and it has to be possible to add new functionality to the wiki. Several semantic wikis exist:

- **KiWi** - EU-funded project combining the wiki philosophy with methods of the Semantic Web.
- **Knoodl** - Tool for creating ontologies. Wikis can be created by the community from within Knoodl. Each community has its own wiki, integrating its specific vocabulary.
- **OntoWiki** - Open-source semantic wiki, acts as an ontology editor.
- **Semantic MediaWiki** - Extension of MediaWiki, integrates the RDF triples in the wikitext.
- **Wikidsmart** - Adds semantics to Confluence. Made by zAgile.

Kiwi, Knoodl and OntoWiki are wikis that support the managing of ontologies. MediaWiki and Confluence have a wider range of application, and do not have ontologies as their only goal. The two biggest contenders for further use are Wikidsmart, adding semantics to Confluence, and Semantic MediaWiki, which adds ontology support to MediaWiki.

The best-known semantic wiki software is probably Semantic MediaWiki, which has extensive documentation, maturation, extensions and possibilities for adding new functionality. Wikidsmart's main advantage is that it uses Confluence, which is tailored for software development companies. The ultimate goal should be to incorporate the tests that are created with any requirement management tool that is already used in the company. For many companies, this might be the products of Atlassian, namely Confluence and Jira.

The limiting factor of Wikidsmart is that it builds on top of Confluence, which in turn requires a license. A zero-cost license program is available for non-profit organizations and open source projects, but there are some requirements to be regarded as a qualified open source project, such as a public site, public availability to Confluence, and so forth. This is a serious impediment in a project like this, and makes the choice of a wiki platform easier. Ultimately, Semantic MediWiki offers more ontology support, and

the multitude of documentation and tools that already exist makes it more efficient to use this wiki server when creating a tool as a proof of concept. If the approach of using a semantic wiki for test creation has merit, effort should be made to incorporate it in enterprise wikis used by companies today.

Table 5.1, which is based on WikiMatrix[4], displays some of the differences between Confluence, MediaWiki and SMW+. SMW+ is a commercial extension of Semantic MediaWiki developed by Ontoprise GmbH.

| | **Confluence** | **MediaWiki** | **SMW plus** |
|---|---|---|---|
| Version | 3.5 | 1.16.5 | 1.5.3 b1 |
| Open source | No | Yes | Yes |
| Programming language | Java | PHP | PHP |
| Development status | Mature | Mature | Mature |
| Interface languages | 15 languages | 140 languages | 3 languages |
| Syntax Highlighting | Yes | Yes (plugin) | Yes (plugin) |
| Webserver | Tomcat, JBoss etc | Any PHP | PHP 5.0+ |

Table 5.1: Comparison of wikis

[4]www.wikimatrix.org

## 5.3 Semantic MediaWiki

The term wiki stems from Ward Cunningham, who in 1994 invented the first wiki, called WikiWikiWeb [25]. A wiki consists of a set of editable pages that are connected by hyperlinks. Information should be easy to find, and easy to edit, so that everyone may contribute. Users are the primary contributors, and the participation of the users determines the site's success. The wikis use *wikitext*, which is natural language with special syntax constructs for the layout of the text. Several markup languages for wikis exist.

### 5.3.1 MediaWiki

MediaWiki is mostly known for being the wiki used for Wikipedia. MediaWiki is a robust wiki engine that is capable of creating large wiki farms where one or more servers host multiple individual wikis [26]. The minimum hardware requirement is 256MB of RAM and 40MB of available storage space, although more is recommended if the site has a lot of traffic.

The features of MediaWiki include [26]:

- Easy navigation: Search, go-button, random page, special page, printable versions of articles
- Easy editing, formatting and referencing to other pages
- Look and feel changes, for the whole site or individual pages
- File uploading
- User management
- History of changes, possibilities to roll back changes

### 5.3.2 Ontologies in SMW

Semantic MediaWiki (SMW) is an extension MediaWiki, where semantic annotations are added to the original functionality. A limit of most wikis is how content is queried. E.g., if you want a list of the ten biggest cities with a female major over 50 years, you would have to update the list manually. With semantic annotations, it would be possible to create this list dynamically, so that the query is defined once and the content of the list is dynamic. This concept of structuring the data on the web comes from the vision of the *semantic web*. Formalizing information and adding metadata makes it possible for computers to reason over data.

A semantic wiki uses a domain ontology to structure the data. In "Semantic

wiki engines: a state of the art" [27], two approaches for semantic wikis are identified. The first is called *wikis for ontologies*, where wiki pages are concepts and links are properties. The problem is to keep the ontology consistent. Nothing prevents a user from entering two relations with similar semantics, such as "has-monarch" and "has-king" [27]. The second approach is called *ontologies for wikis*, where the main use is to create instances of concepts. Our use of Semantic MediaWiki will employ the second approach, where an existing ontology is used as a basis for creating new instances, for example adding a person "John Doe" who is an instance of the concept "Person".

An example of semantic annotations on MediaWiki is the following. On a page called "Norway", the population can be saved as [[Has population:: 4,946,488]]. In the SMW database, this is stored as: Norway (subject) Has population (predicate) 4,946,488 (object). This information can be used in queries to for example list all countries with more than five million inhabitants. When the page about Norway is updated with a new population, it will be reflected in the list.

The main obstacle with *ontology for wiki* is that its main purpose is creation of instances that belong to a certain concept from the ontology. In our case, the objective is to create tests (Cucumber scenarios) based on the information that can be found in the ontology. This is a task of using the concepts and relations, and not adding new instances. The only use for instances may be if concrete test data needs to be added. If a test use the concept "Person", an instance of person can be created which serve as test data, for example by having properties such as age, address and phone number.

### 5.3.3 Extensions

For MediaWiki, almost 1000 extensions exist. At the time of writing, 57 extensions are specific to Semantic MediaWiki. The Halo extension is perhaps the most significant one, providing an intuitive graphical interface for operations on semantic data, including a semantic toolbar, advanced annotation mode, auto-completion, graphical query interface and an ontology browser. As mentioned earlier, SMW+ is a bundle of extensions, where the Halo extension is included. The extensions provided in SMW+ are modified by Ontoprise in order to increase the efficiency when the extensions are working together. This has the weakness that updates made to extensions will not be reflected through SWM+ before Ontoprise makes the corresponding update.

Annotating data in semantic wikis requires more from the users than the plain markup language. Some functionality has been added to SMW to ease the editing. This includes forms and auto-completion, as well as a WYSIWIG editor. Semantic Forms is an extension to MediaWiki that allows users to add, edit and query data using forms. It is possible to create forms for adding new concepts and properties, and for adding instances for concepts in the ontology. The forms support autocompletion based on content from the wiki. Semantic Forms is one of the extensions that are often updated, and the latest version is not always provided by SMW+.

Out of a set of 216 known active, public SMW-based wikis, the ten most popular SMW-based extensions are [28]:

1. Semantic Forms - 85%
2. Semantic Result Formats - 33%
3. Semantic Drilldown - 30%
4. Semantic Maps - 25%
5. Semantic Compound Queries - 19%
6. Semantic Google Maps - 7%
7. Semantic Internal Objects - 6%
8. Semantic Forms Inputs - 6%
9. Semantic Tasks - 5%
10. Halo - 3%

# Part II

# Development of a Tool

# CHAPTER 6

# TOOL REQUIREMENTS

This part of the thesis describes the development of a semantic wiki tool called WikiTest. The main focus of this chapter is RQ3 (see Chapter 2): the identification of functionalities for a collaboration tool that can be used in a test specification process. The requirements that are found will be prioritized according to their research interest. The scope of the thesis does not make it possible to implement everything, thus only the most important functionality will be implemented.

## 6.1 Early Outlook

In the latest years, several articles have described how a wiki can be used in a software engineering process [29][30]. How to write and track requirements has been a common topic for the wikis. Little has been written about how wikis can be used for test specification.

It is useful to look at some of the experiences that others had when they used wikis as a collaboration tool in software engineering. As shown in "Extending and Integrating Wikis to Improve Software Documentation" [31], current wikis should be better integrated with IDEs. Other challenges of enterprise-wide wikis were found in a survey performed by Romberg [32]:

- **Usability:** The appearance of text completely changes between viewing and editing. Learning wiki-syntax is a slow process. No direct feedback is given, making errors more likely.
- **Productivity:** Some operations are more complicated than they should be, such as "find and replace", which reduce productivity.

- **Growth and overview:** Large wikis can become poorly structured if no one takes responsibility for keeping the wiki content organized.
- **Multiple companies:** Who will host the wiki and what happens to the information on the wiki when the project is over.
- **Offline access:** Although wireless internet access becomes more and more common, offline access may still be a problem for people doing extensive traveling.

In “Support for High-Quality Requirements Engineering in a Collaborative Setting” [33], a tool called SmartWiki was implemented. The tool is built on Semantic MediaWiki and is used to write requirements. It has been evaluated in multiple projects over a three year period. In this tool, the most interesting functionality is perhaps the heuristic feedback that is given to the user when he writes requirements. The text is analyzed, and if ambiguous words are found, the user is notified so that he can rewrite the requirement. Since the wiki was evaluated both in university teaching and industrial projects, it is interesting to see the challenges that were described. One challenge was that the setup and familiarization was found to be time consuming. Even though the wikitext syntax is simple, some training and hints should be given to enhance the productivity. Another challenge was that in industrial settings, MediaWiki did not always provide enough separation of user rights. Some people should only be able to see certain documents in the wiki.

The vision of this wiki, WikiTest, is to be a tool that can assist the customers and developers to create tests. The core functionality is to start with a domain ontology and requirements, let the user write tests and let the user be able to implement code for these tests.

## 6.2 Focus Group

A focus group, as described in Chapter 2.2, was used to elicit potential requirements for the tool. The focus group was held the 14th of April. Four professors and three PhD candidates from NTNU participated. The author of the thesis acted as a facilitator, which made it a total of eight participants.

In the start of the focus group, the following five topics were presented:

1. Domain ontologies
2. Boilerplates
3. Writing test specifications in Cucumber
4. Implementing and running tests in Cucumber
5. MediaWiki

The majority of the participants were already familiar with domain ontologies. A few of the participants had experience with acceptance testing tools such as FitNesse, but not with Cucumber. As part of the initial presentation, a few Cucumber tests were written and run to show the tool in use. An open discussion, centered around some key questions, was initiated after the presentation. One such question was what help and instructions would be needed to use the tool, especially for someone without much IT experience. Another question was how the ontology information could be used to write tests, and what functionality would be needed in the wiki. Each question discussed in the focus group will be described in the following sections.

### Domain Specific Languages

The first topic that came up was how tests could be written using a template. One suggestion was to define a grammar using Xtext, which makes it easy to create an editor with syntax highlighting and autocompletion. As an example, when you enter “a” and click space, “And” would come up as a suggestion (since it is a Cucumber keyword). The ontology would provide the “clean” objects, while the DSL would describe how the concepts from the ontology could be used as text. Creating a grammar from the ontology would make it possible to analyze the correctness of the tests. For example, if *steam* and *valve* are used in a sentence, it would be possible to verify based on the ontology that the context they are used in is valid.

> *Illustrative quotation from the focus group:*
> “The state has to be valid according to the domain model. For example, a person with Age, where the age is 17, would mean that

> the person is deducted to be a minor. On a similar note, how do we know what meaning Open has? We know that it comes after Valve. Valve qualifies for the use of Open. When you say "valve is open", is there something that checks that this is a legal state?"

Valid states was a hot theme during the focus group. The participants that raised their opinion, felt that checking if a state was legal would be more helpful than autocompleting keywords. One participant worried about customers writing something in the When step that would not make sense according to the model. When a tester would implement the test he will know a lot about testing, but perhaps not enough about the domain to know that the When step does not make sense according to the domain.

## Writing Tests

One participant suggested to use a "boilerplate" for the Given/When/Then combination. It could look like "Given a <concept> with <state>, When ...". Another suggestion was to use wiki forms. This would make it easier for the user to create the tests without making errors. There were some discussion about it, as some participants felt it was fine to write text using wikitext. The use of forms compared to wikitext with autocomplete may be a good topic to analyze in an experiment involving students.

> *Illustrative quotation:*
> "Often, the starting state is large and complex. This is followed by a small "trigger", and then a description of the wanted end state."

## Visualizing the Ontology

It may be beneficial to display the domain ontology to the user, but this will probably not scale well. It is easy to fill in test information for a small domain. When the domain grows large, it becomes more difficult to make sure the concepts are used correctly, and it is difficult to get an overview of which concepts can be used.

> *Illustrative quotation:*
> "Keeping an overview over a lot of components is difficult. An example is in a steam boiler domain with a lot of different valves. The user will need help to find the component he is looking for."

## Test Coverage

Checking if all nodes have been included in at least one test would be useful. We could use some kind of "model coverage" that shows the coverage for what is specified in the domain ontology. For example, if the system says that you have tests for nine out of ten requirements, it has to say which parts need more tests.

## When Enough is Enough

The question about when a test case is specified enough was raised. This quote represents the issue:

> *Illustrative quotation:*
> "We want two things. We want something that is simple, and we want something that can support us when we write tests. What happens today is that often an Excel sheet is used, where one row is a test and there is a column called "status". We want that every requirement have at minimum one test. If not, it is not a requirement.
>
> The biggest problem is to say when "enough is enough" in the tests. That is, when are the tests specific enough, when are the states properly specified. This is not specific to the use of a wiki for test specification, but it is a general problem when writing tests."

It would be beneficial to have some way to say when tests are specified good enough. It is a common problem that the initial state is not properly specified, and so it becomes up to the tester to judge how the initial state should look like. Often, there is no documentation of what states the system could actually be in, making it more problematic to create a proper test.

## Test Hierarchy

Creating states that are compositions of other states, and inheritance of states, would be useful. Tests often have similar starting states, with only a minor difference. It should be possible to reuse tests and states, and specify variation points for these tests and states. A steam boiler could have an "operating state", where the temperature is between a min and max value, the pressure is between the min and max value, and so forth. Then it would

be possible to use this operating state and alter it, such as adding too much water in the tank, creating an erroneous state from the normal state.

Reuse of states from another test would make it possible to have some kind of “network” around a starting node, where the starting state is the same with only minor variations. Thus you achieve a link between tests, and the tests that are not linked to any other tests should be further investigated. Perhaps those separate tests are special conditions that need extra care, i.e. extra tests to achieve a higher coverage.

## 6.3 User Stories

The requirements for the wiki tool span a wide range of functionality. Even though only some of the requirements will be implemented in the prototype, the identification of possible functionality is important to analyze the validity of a wiki-based test approach. The requirements are split into categories. Some of the requirements are general, which means there are multiple ways to implement the functionality. The best way to implement the requirement may vary from company to company, depending on their needs.

The requirements are written using the following format: As a <user> I want to <capability or action> so that <benefit or purpose>. This is a user story consisting of who, what and why. The generic term "user" will be used if the user type is common for customers, developers, testers and so forth.

**R1: Import ontology**
As an ontology-responsible, I want to import an existing domain ontology so that it can be used in the wiki.

**R2: Import requirements**
As a requirement-responsible, I want to import requirements so that already written requirements can be used in the wiki.

**R3: Export features**
As a developer, I want to be able to export all features in the wiki to .feature-files so that I can write test code for the features.

**R4: Export printable documents**
As a customer, I want to be able to export printable documentation of requirements and tests so that I can get it on paper.

**R5: Export ontology**
As an ontology-responsible, I want to be able to export the ontology to OWL or RDF format so that I can use the ontology in other systems.

**R6: Access control**
As an administrator, I want to have access control so that only certain users can view or edit certain features.

**R7: IDE integration**
As a developer, I want to integrate the wiki with an IDE so that I can make changes in the wiki from within my IDE.

**R8: View test results**
As a customer or project leader, I want to be able to watch the test results from all implemented scenarios so that I can see the progress of the project.

**R9: Domain suggestions**
As a user writing tests, I want to receive suggestions from the wiki of relevant domain concepts so that I can create better tests.

**R10: Step suggestions**
As a user writing tests, I want to receive a list of earlier written step definitions so that I can reuse them in my scenarios.

**R11: Analyze words**
As a user writing tests, I want to receive feedback if my tests are written ambiguously so that I can write avoid using ambiguous words.

**R12: Test pattern suggestions**
As a tester writing tests using boilerplates, I want to receive a list of typical ways to test the requirement so that I can reuse knowledge of how the boilerplate is usually tested.

**R13: Display tests**
As a user, I want the features and scenarios to be visually formatted so that they are easy to read.

**R14: List missing tests**
As a tester, I want to be able to get a list of all features without tests so that I can write tests for these features.

**R15: Domain concept information**
As a developer, I want to be able to see information about a domain concept used in a feature so that I can understand what the feature really means.

**R16: Hierarchy of features**
As a tester, I want to group features so that I can more easily manage a large amount of features.

**R17: Model coverage**
As a tester, I want to be able to see a test/model coverage so that I can find out which parts of the domain ontology are not used in any tests.

**R18: History**
As a tester, I want to see the change-log for a specific feature so that I can roll back the changes to an earlier version.

**R19: Watch-list**
As a tester, I want to be able to put a feature on my watch-list so that I can get information if anyone make changes to the feature.

# CHAPTER 7

# TOOL IMPLEMENTATION

This chapter describes the setup and implementation of WikiTest. The implementation is based on the most important requirements of Chapter 6.3. Each section in this chapter describes a different implementation topic. The first section “Setup and Installation” is not directly related to any user story, but has to be in place for the user stories to be implemented.

When a web-based wiki was selected as a technology platform, a large amount of wikis could be used. One of the reasons why Semantic MediaWiki was selected as the wiki engine was because of the large number of extensions. Each extension has its use and place. This reduces the need to write a lot of code; the work is to identify what existing functionality can be reused.

## 7.1 Setup and Installation

MediaWiki and its extensions are under constant development. Especially the semantic extension, SMW, is often updated. This section gives some pointers to the installation of MediaWiki and the extensions used for test specification. A point to note is that when the versions change, the installation process may change as well. If a wiki-based test process is to be used in the industry, it will be important that the approach is general and that the newest versions of MediaWiki and extensions can be used.

Two setups will be described. One of the setups will include only a few extensions. This is the minor one, which will be used in the experiment described in Chapter 8. The other setup use a wider range of extensions

to provide more functionality, but is more tedious to install as well. The extensions are shown in Table 7.1 and Table 7.2.

| Software | Version |
|---|---|
| MediaWiki | 1.16.2 |
| PHP | 5.2.4 |
| MySQL | 5.0.51a |
| Semantic MediaWiki | 1.5.6 |
| Semantic Forms | 2.1.2 |
| ParserFunctions | 1.3.0 |

Table 7.1: Software in the minor installation

| Software | Version |
|---|---|
| MediaWiki | 1.16.2 |
| PHP | 5.3.5 |
| MySQL | 5.5.9 |
| Semantic MediaWiki | 1.5.2_1 |
| Semantic Forms | 2.0.0_1 |
| ParserFunctions | 1.3.0 |
| ScriptManager Extension | 1.0.1_0 |
| SMWHalo Extension | 1.5.2_2 |
| ARCLibrary | 1.0.0_0 |
| Semantic Forms Inputs | 0.3 |
| Semantic Gardening extension | 1.3.3_0 |
| WYSIWYG extension | 1.4.0_3 |

Table 7.2: Software in the full installation

The main difference lies in the inclusion of the SMWHalo extension. The minor version was used on a dedicated server at NTNU (folk.ntnu.no), while the full version was deployed locally on a laptop. On the full version, SMWHalo was used, which has adjusted Semantic Forms and Semantic MediaWiki. The adjustments are the reason for the underscore in the version number of these extensions. SMWHalo provides interesting functionality, such as an ontology browser, but is difficult to install on the server at NTNU, and it did not allow the newest version of Semantic Forms to be used.

The first thing to do is to install MediaWiki [1]. A web server such as Apache or IIS is needed, as well as PHP version 5.2.3 or later. As database, either MySQL with version 4.0 or higher, or PostgreSQL with version 8.1 or higher, may be used. MediaWiki is downloaded and extracted on the web server. When you access the web site for the first time, an installation script guides you through the rest of the installation.

Installing Semantic MediaWiki consists of three steps[2]. The first is to download and extract the files to the extension folder of the wiki you created when you downloaded MediaWiki. The second is to insert the two first lines given in Listing 7.1 into the bottom of `LocalSettings.php`. The third is to enter the wiki, go to the `Special:SMWAdmin` page and click on "database installation and upgrade", and then “data repair and upgrade”. This creates the database tables that are needed to use SMW. For the minimalistic installation, Semantic Forms and ParserFunctions is yet to be installed. The installation procedure is the same as for SMW; download and extract it to the MediaWiki extension folder.

Listing 7.1: LocalSettings.php

```
include_once("$IP/extensions/SemanticMediaWiki/SemanticMediaWiki.php");
enableSemantics('example.org');
include_once("$IP/extensions/SemanticForms/SemanticForms.php");
require_once("$IP/extensions/ParserFunctions/ParserFunctions.php");
```

This concludes the installation of the minimalistic version with the fewest extensions. In order to use SMWHalo, it is recommended to use the Deployment framework [3]. This is a framework that, once installed, makes it possible to install extensions from a command line.

[1]http://www.mediawiki.org/wiki/Download
[2]http://semantic-mediawiki.org/wiki/Help:Installation
[3]smwforum.ontoprise.com/smwforum/index.php/Help:Installing_the_Deployment_Framework_1.3

## 7.2 Importing a Domain Ontology

This is related to requirement R1 *import ontology* and R2 *import requirements*. Semantic MediaWiki uses a domain ontology to structure information. New concepts may be added from within the wiki, but a more common situation for a company using a semantic wiki is to import an existing ontology. Semantic MediaWiki already have support for the import of an ontology through the `Special:Import ontology` page, but it has been found to be buggy and is disabled in version 1.0 [34]. An extension called Semantic Gardening, which requires SMWHalo, can instead be used to import ontologies.

To upload files on the wiki, MIME type verification should be turned off. This is done by adding `$wgVerifyMimeType=false;` to `LocalSetting.php`. In addition, OWL files should be added to the file extensions: `$wgFileExtensions[] = "owl";`. Afterwards, the ontology can be imported using the gardening bot, as shown in Figure 7.1.

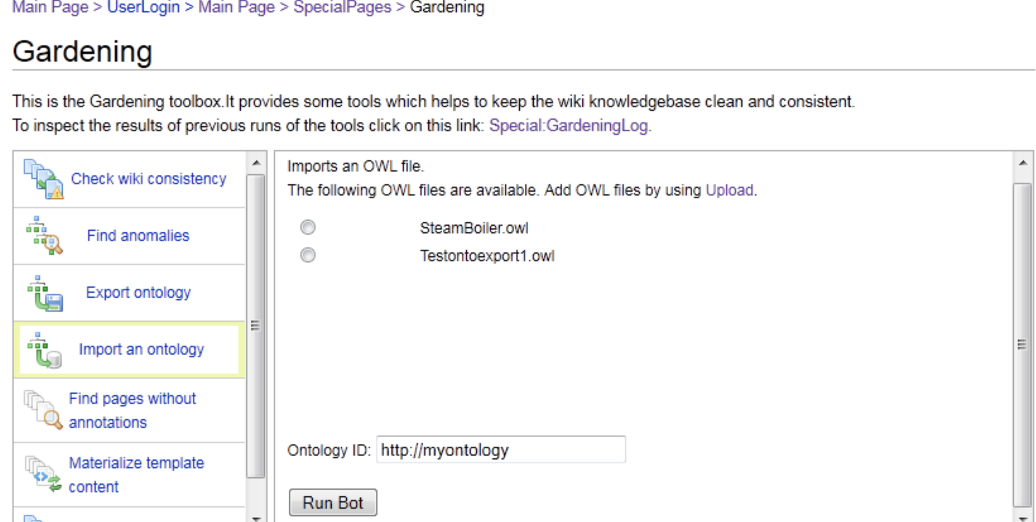


Figure 7.1: Semantic Gardening import

If the minimalistic approach without SMWHalo is used, there are two options. If the ontology is large, the best approach is to use the Python Wikipedia Bot[4]. An example of code using the bot is given in [34]. If the ontology is small, the easiest is to import it manually. A mapping from the ontology to WikiTest is:

[4]http://en.wikipedia.org/wiki/Wikipedia:Creating_a_bot

- Add all domain concepts to the main namespace of the wiki (e.g. wiki/Foo), and annotate each page with *category* and *label*.
- Add all domain properties to the property namespace (e.g. wiki/Property:Foo) and annotate each page with a label.

The labels act as a way to mark each domain concept and property that is imported from the domain ontology. This makes it possible to retrieve all pages that come from the domain ontology. Note that there are pages in the *property* namespace of the wiki that do not come from the domain ontology. "Modification date" is one such special property that is used by the wiki itself.

Requirements can be imported in a similar fashion, either by using a script if the amount of requirements is large, or manually if the amount of requirements is low. Even though the initial basis for the tool was to use requirements formulated in boilerplates, this is not a prerequisite in the wiki.

## 7.3 Creating a Feature

A Cucumber feature will have its own page in the wiki. At a minimum, it will consist of a feature title, a requirement description and at least one scenario. It may also consist of other parts, such as a Cucumber background. We want to make a form from which the user can create features. This form needs to include templates for specifying the requirement, the background and the scenarios.

Figure 7.2 shows the general idea of how forms, templates and pages are connected. The form is made up of three templates: one for the feature, one for the background and one for the scenario(s). Form definitions, templates and properties are parsed on the fly to create the form. In the figure, the "control steam pressure" page is an instantiation of the feature form, where the requirement and one scenario is filled out.

Form definitions are stored in the `Form` namespace of the wiki, templates are stored in the `Template` namespace, categories are stored in the `Category` namespace and properties are stored in the `Property` namespace. The instantiation of the feature form, e.g. "control steam pressure", is stored in the main namespace. Since the page is made using the form, it will be in the "Feature" category. Going to `wiki/Category:Feature` will list all feature pages.

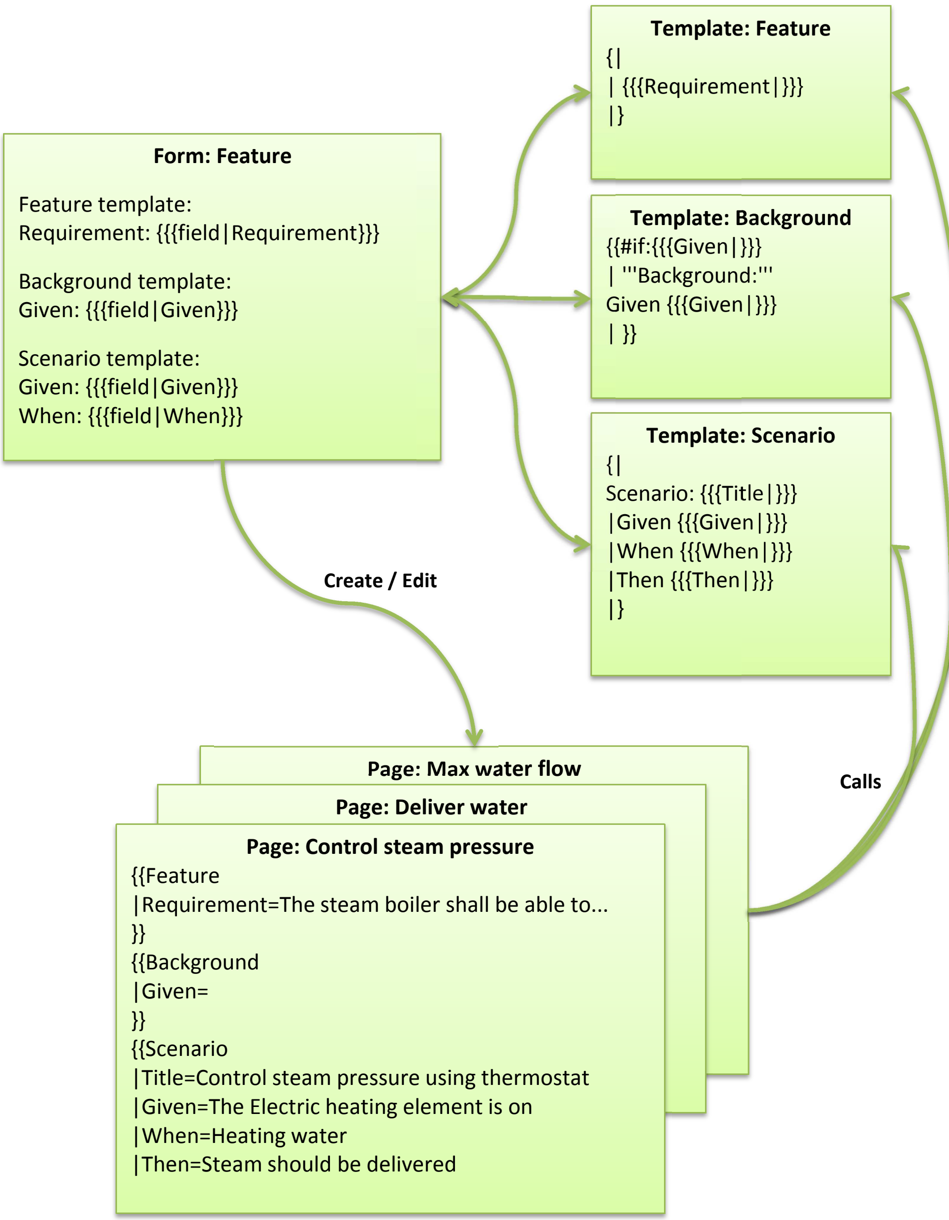


Figure 7.2: Composition of the feature form

Forms are associated with a category or a namespace to allow editing with the form instead of the source code (wikitext). This means that the feature can be edited in the same manner as it was created.

The complete code for the feature form is given in Appendix A.1. The syntax depends on which version of Semantic Form is used. Here, Semenatic Forms 2.1.2 is used. The feature, background and scenario templates are given in Listing A.2, A.3 and A.4. The most noteworthy is the use of a parser function, `#if :{{{Given|}}}`, in the background feature. This is used to only render the background if any Given steps are written. In the scenario template, the syntax for formatting the step definitions is:

```
|Given {{#arraymap:{{{Given|}}}|\n|x|[[Given::x]]|\n\n  And\n}}
```

This displays the steps so that if more than one step is written, the next one is started on a new line with the And keyword in front of it. Another implementation approach needs to be taken if the But keyword shall be supported as well.

An example of how the wikitext of a feature page will look is shown in Listing 7.2. The WikiTest visualization of this feature is shown in 7.3.

Listing 7.2: Steam boiler feature

```
{{Feature
|Requirement=The steam boiler shall be able to
 control steam pressure using thermostat of electrical heating element
}}
{{Background}}
{{Scenario
|Title=Control steam pressure using thermostat
|Given=the electric heatinge element is on
|When=heating water
|Then=steam should be delivered
}}
```

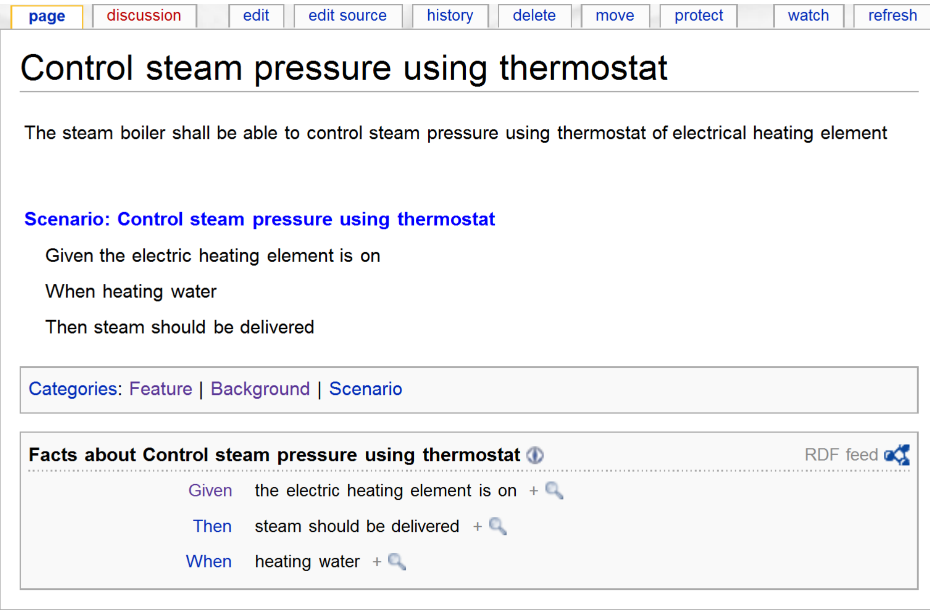


Figure 7.3: Visualization of an example feature

Editing the wikitext directly would be prone to errors. Figure 7.4 shows how the edit would look like when a form is used.

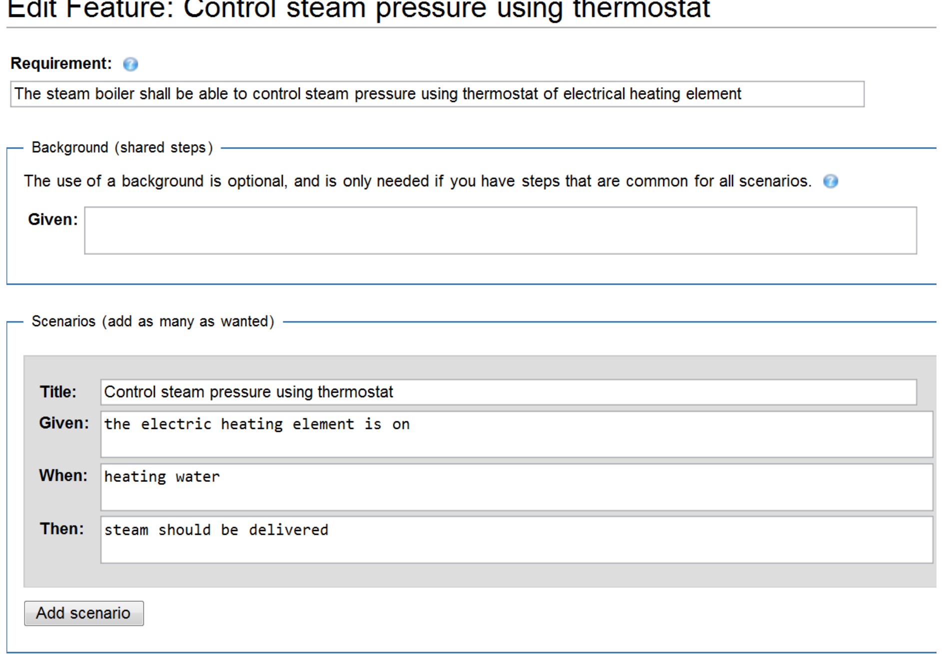


Figure 7.4: Edit a feature using a form

## 7.4 Autocomplete

A semantic form has fields where the user can enter values. Each field can be connected to a semantic property, which is specified when the form is created. A scenario has the fields Given, When and Then. The fields are connected to properties with the same name. When a field has a property, it can have autocomplete based on previously written values of the property. This section about autocompletion is related to the two requirements R9 and R10, which cover domain suggestions and step suggestions.

The difference in WikiTest, compared to a content-driven wiki, is that each of the fields should not have autocomplete only based on the property, but also autocomplete based on domain concepts. The autocomplete provided by Semantic Forms 2.1.2 is located in `SF_Utils.php`, shown in Listing 7.3. For Semantic Forms 2.0.0_1, used by SMWHalo, the same code can be found in `SF_FormsInput.php`.

Listing 7.3: SF_Utils.php

```
/**
 * Creates an array of values that match the specified source name
 * and type, for use by both Javascript autocompletion and comboboxes.
 */
static function getAutocompleteValues( $source_name, $source_type ) {
  $names_array = array();
  // the query depends on whether this is a property, category, concept
  // or namespace
  if ( $source_type == 'property' ||
       $source_type == 'attribute' || $source_type == 'relation' ) {
    $names_array = self::getAllValuesForProperty( $source_name );
  } elseif ( $source_type == 'category' ) {
     $names_array = self::getAllPagesForCategory( $source_name, 10 );
  } elseif ( $source_type == 'concept' ) {
    $names_array = self::getAllPagesForConcept( $source_name );
  } else { // i.e., $source_type == 'namespace'
    // switch back to blank for main namespace
    if ( $source_name == "Main" )
      $source_name = "";
    $names_array = self::getAllPagesForNamespace( $source_name );
  }
  return $names_array;
}
```

To account for both property values and domain concepts, the method is extended with the code given in Listing 7.4.

Listing 7.4: Modified SF_Utils.php

```
if ( $source_type == 'property' ||
     $source_type == 'attribute' || $source_type == 'relation' ) {
  $properties_names_array = array();
  $domain_names_array = array();
  $properties_names_array = self::getAllValuesForProperty( $source_name );
  if ($source_name != 'Label') {
    $domain_names_array = self::getAllValuesForProperty( 'Label' );
  }
  $names_array = array_merge($properties_names_array,$domain_names_array);
}
```

Ulitmately, the autocomplete should be “intelligent”, in the sense that only related concepts are proposed. This would mean the whole sentence would be analyzed before suggesting the next word. Perhaps it would be necessary to analyze the whole page, so that the suggestion for a Then step would take the values of Given and When into consideration.

An intelligent autocomplete was not implemented in this WikiTest version. A simple improvement of the autocompletion would be to use a breadth first search starting from the last entered word. Thus, if the last word the user entered was “boiler”, then all relationships the boiler has would be given as suggestions, followed by all concepts of these relationships. More words could be found by doing the same procedure for the related concepts of the boiler. Just a simple improvement like this would make the autocomplete substantially more valuable for the user.

## 7.5 Creating a Domain Concept

If a domain concept is forgotten during the creation of the domain ontology, it can be added later. In the wiki, this is done in the same manner as for creating a feature. A template looking like the one in Listing 7.5 shows the simplicity of how this can be done in the wiki. The domain class will get its own page, which will have a label and be marked as being in the domain class category. The label is an annotation that all domain classes and properties has. It will usually be named the same as the domain class page, but the user is given the freedom to use a different label, such as an abbreviation, if wanted.

Listing 7.5: Template:CreateClass

```
<noinclude>
This is the "CreateClass" template.
It should be called in the following format:
<pre>
{{CreateClass
|Label=
}}
</pre>
Edit the page to see the template text.
</noinclude><includeonly>
''Label: [[Label::{{{Label|}}}]]''
[[Category:Domain class]]
</includeonly>
```

The template is used as part of the form shown in Listing 7.6. In addition to HTML styling for the form, the only extra element specified here is a free text input field in the bottom. This text field can be used to describe the domain concept. It could be a good practice to write a thorough explanation about each domain concept, especially in domains with difficult concepts, or where the turnover of developers is high.

Listing 7.6: Form:CreateClass

```
<noinclude>
This is the "CreateClass" form.
To create a page with this form, enter the page name below;
if a page with that name already exists, you will be sent to a form to
edit that page.

{{#forminput:form=CreateClass|size=50|
button text=Create or edit a domain class|autocomplete on namespace=Main}}

</noinclude><includeonly>
<div id="wikiPreview" style="display: none; padding-bottom: 25px;
margin-bottom: 25px; border-bottom: 1px solid #AAAAAA;"></div>
{{{for template|CreateClass|label=Name of the class
 in the domain ontology}}}
The label will usually be the same name as the name of the class
 you are creating.
{| class="formtable"
! Label:
| {{{field|Label|mandatory}}}
|}
{{{end template}}}

'''Information about the domain class (free text):'''

{{{standard input|free text|rows=10}}}


{{{standard input|save}}} {{{standard input|preview}}}
 {{{standard input|cancel}}}
</includeonly>
```

## 7.6 Navigation

The first thing a user sees when he access the wiki is the main page and the sidemenu. The typical user will use hyperlinks to navigate in the wiki. The alternative is the search field, but it is likely to be used only if the user already know what to search for. Providing good links which represent the functionality of the site has a significant importance of the usefulness of the wiki. Information has no use if the user cannot find it.

The `Main_Page` is part of the main namespace and can be edited directly from the page itself by clicking the edit button. A simplistic main page may provide links to lists of features, categories and properties. From these lists, the user can navigate to a specific page. The main page in the experiment performed in Chapter 8 had the following content:

Listing 7.7: Main page content

```
Use the menu on the left to navigate around in the wiki.
You can search for domain concepts using the search field.

== View ==
[[Features|View all features]]
[[:Category:Domain class|View all domain classes]]
[[:Property:Label|View all domain properties (relationships)]].
You can click on a property to see which domain classes use that property.

== Create ==
[[:Form:Feature|Create a new feature]]
```

Even more important is the menu on the left side of the wiki, which is called the *sidebar*. It provides a search field and links that are accessible on all pages of the wiki. In the experiment, the sidebar consisted of a search field, a group of “view” links, a group of “create” links, and a “toolbox”. The wikitext is given in Listing 7.8.

Listing 7.8: MediaWiki:Sidebar

```
* SEARCH

* view
** mainpage|mainpage-description
** Features|Features
** Category:Domain_class|Domain classes
** Property:Label|Domain properties
** Special:Properties|All properties
** Missing_tests|Missing tests
* create
** Form:Feature|Feature
** Form:CreateClass|Domain class
** Special:CreateProperty|Domain property

* TOOLBOX
```

The “view” group consists of links to pages that provide list of elements, such as all features or missing tests. The “create” group consists of forms that can be used to create new content in the site. The toolbox is a special MediaWiki command. It provides functionality such as “what links here”, related changes, special pages, printable version and more. In addition, a “browse properties” is provided by the toolbox, which makes it possible to see the semantic properties of that page.

## 7.7 Missing Tests

For large domains with hundreds of requirements, it is useful to find all requirements that do not have any corresponding acceptance tests. This is an implementation of requirement R15, *list missing tests*. In order to list all features without tests, we make a template using the extension ParserFunctions. The “missing test” page shown in Listing 7.9 uses a query to find all pages of the Feature category. The output of this query is formatted using the MissingTestsTemplate, shown in Listing 7.10, which renders all pages lacking the Then step. The assumption is that a feature page with a missing Then step has not a (properly) defined scenarios.

Listing 7.9: Page:MissingTests

```
This page lists features without any scenarios.
In order to make sure every requirement has at least one test,
every feature should have a scenario.

== Features without scenarios ==

{{#ask: [[Category:Feature]]
| ?Then
| format=template
| template=MissingTests
}}
```

Listing 7.10: Template:MissingTests

```
{{#if: {{{2}}} | | <p> {{{1}}} </p> }}
```

## 7.8 Ontology Browser

SMWHalo provides an *Ontology Browser*, which is shown in Figure 7.5. It is one of the advantages of using the SMWHalo extension, but comes at the expense of not always being able to use the newest versions of other extensions, such as Semantic Forms. The ontology browser is an already existing functionality that covers requirement R16, *domain concept information*, as it can be used to find out more about a concept in the ontology.

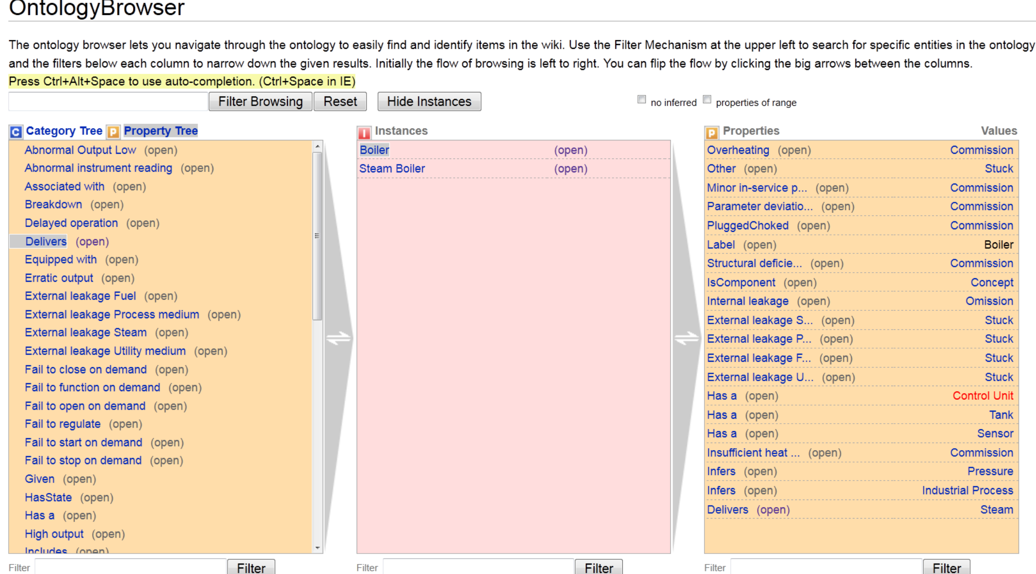


Figure 7.5: Ontology browser

The ontology browser provides a list of all the properties in the wiki. Selecting a property, such as “delivers”, gives a list of all instances that use this property. This way, the content of the ontology can be navigated. A potential improvement is to add a graphical display of the nearest ontology concepts for a given concept.

# Part III

# Experiment

# CHAPTER 8

# DEFINITION

This chapter describes the goal definition of the experiment. The goal definition of the experiment is established using the Goal/Question/Metric approach, which is explained in the first section of the chapter. The goal, questions and metrics are defined subsequently. Section 8.5 provides a summary of the experiment definition using the GQM approach.

## 8.1 The GQM Process

Goal/Question/Metric (GQM) is an approach created by V. Basili and D. Weiss during the early 1980s [35]. The approach is used to guide the definition of measurements in software engineering.

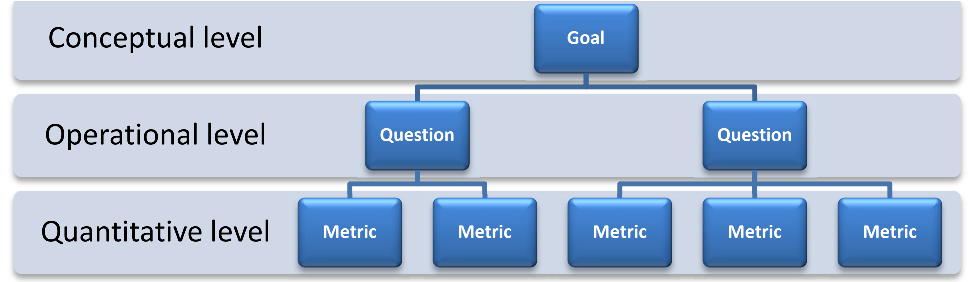


Figure 8.1: The GQM hierarchy

GQM is a goal-driven systematic technique. It is divided into three levels [36]. The uppermost level is the conceptual level - the *goal*. The goal is about some object, such as product or a process. Each goal has a number of *questions*,

which is the operational level. The questions try to characterize the object based on a quality characteristic of the object. Each question is associated with one or more *metrics*, which is the collected data used to answer the question [37]. The measurements can either be objective or subjective.

## 8.2 Goal

GQM uses a template for the formulation of goals. Several GQM templates exist, but they all have the same basic structure. The template contains information about the object to study, the purpose, the focus, the stakeholders and the context. A goal template for GQM is [8]:

> Analyze **object(s) of study**
> for the purpose of **purpose**
> with respect to their **quality focus**
> from the point of view of the **perspective**
> in the context of **context**.

The experiment goal is linked to RQ4, defined in Chapter 2, and is used to create the hypotheses defined in Chapter 9.2. The experiment goal, with a slight modification of the goal template, will be:

> Analyze **WikiTest**
> for the purpose of **evaluating the tool**
> with respect to **its usefulness for writing acceptance tests**
> from the point of view of **customers and developers**
> in the context of **acceptance tests for requirements with an underlying domain ontology**.

The end goal is to determine the usefulness of the tool for customers and developers who want to specify acceptance tests that can be automated. By "usefulness" we mean quality characteristics such as usability, efficiency, correctness and completeness of tests. These quality characteristics were prioritized when eliciting the requirements in the third research question, RQ3, defined in Chapter 2. An important point is that the prerequisite of the tool is that a domain ontology exists.

## 8.3 Questions

Four questions are identified which are further defined in hypotheses H1-H11, which can be found in Chapter 9.2. Each question focuses on a specific quality characteristic, with one or more corresponding metrics. An overview of the questions and metrics are given in Table 8.1, at the end of this chapter.

**Question 1: Test case quality**

> What is the quality of the created tests in terms of completeness and correctness?

A good test will be complete, correct, consistent, unambiguous and verifiable. This question aims to find out if these characteristics are different in the tests created depending on the method used to create them; if a domain ontology is used or if the WikiTest tool is used.

General experience with testing, tools and programming knowledge may have an influence on the quality of tests. The following metrics are used to assess the experience of the participants in the experiment:

- M1: Programming experience
- M2: Wiki experience
- M3: Ontology experience
- M4: Automatic test tool experience
- M5: Cucumber experience

The quality of the tests will be judged syntacticly and semantically by five different metrics.

- M6: Is the feature completely covered by scenarios?
- M7: Is the scenario complete?
- M8: Is the scenario syntacticly correct?
- M9: Is the scenario unambiguous?
- M10: Is the scenario verifiable?

For each of the metrics M6 to M10, a subjective score is given by the author of the thesis of either 2 (correct), 1 (partly correct) or 0 (not correct). Weighting each of the five metrics equally, the overall test case quality will be defined as the sum of these metrics.

#### Question 2: Understandability

> Is WikiTest easy to understand?

This questions seeks to answer how easy or difficult WikiTest is to use, and what extra help the user feels he needs. In addition to WikiTest itself, it will be important to find out how easy it is to understand the Cucumber way of specifying tests, and how easy it is to use a domain ontology. If Cucumber or ontologies are difficult to grasp, then the WikiTest tool should add features that make them easier to understand.

- M11: Does using a domain ontology make it easier to understand the domain?
- M12: Is it easy to understand how to write tests using the Cucumber-style?
- M13: Is it easy to understand WikiTest?

The measurements are taken by using a post-experiment survey using a five point Likert scale, asking the participants of the perceived ease of use. In addition, the participants will be asked if more help was needed in order to understand the domain ontology, the Cucumber-style or the WikiTest tool.

#### Question 3: Usability

> Is WikiText pleasant to use?

Usability, as well as understandability, will be important if business customers are to be able to write tests. It will be important to determine how easy it is to create, view, edit and delete test specifications. In addition, it will be important to assess how pleasant WikiTest is to use, and what functionality of WikiTest needs to be improved. The measurements are made by using a post-experiment survey using a five point Likert scale.

- M14: Is it easy to get an overview of all the tests using the tool?
- M15: Is it pleasant to use the WikiTest tool?

### Question 4: Efficiency

What is the efficiency of test creation when using WikiTest?

The efficiency of test creation is important. Using a domain ontology may lead to extra work, at least for small test specifications where the tester is likely to know the domain without the use of an ontology. The same argument goes for WikiTest; it is probably faster to write a few tests in any text editor, but once the amount of tests is large, the wiki-tool may be faster. In the experiment, only a few features with corresponding tests will be used.

- M16: How much time is used in total for all tests?
- M17: How much time is used per feature on average?
- M18: How much time is used per scenario on average?

The measurements will be calculated by taking the time from when the pre-experiment survey is submitted to the time the post-experiment survey is started. Metric M17 is found by taking M16 and dividing it by the number of features. Metric M18 is found by taking M16 and dividing it by the number of scenarios.

## 8.4 Metrics

**Metric 1 - Programming experience**

Measurement procedure: Perform a survey before the experiment.
Question: What kind of programming experience do you have?
Alternatives:

1. Only programming classes
2. Programming classes and hobby projects (creating a web site etc)
3. Worked 2 to 8 weeks in a software company
4. Worked more than 8 weeks in a software company

Expected result: More than 80% will have some experience (alternative 1 or 2). Less than 20% will have any industrial experience.

**Metric 2 - Wiki experience**

Measurement procedure: Perform a survey before the experiment.
Question: What is your experience with Wikis (for example Wikipedia)?
Alternatives:

1. None
2. Read articles, for example on Wikipedia
3. Edited at least one article on a wiki
4. Have had my own wiki server

Expected result: More than 80% will answer that they have read wiki articles.

**Metric 3 - Ontology experience**

Measurement procedure: Perform a survey before the experiment.
Question: Have you had any experience with domain ontologies?
Alternatives:

1. None
2. Heard about it
3. I have used a domain ontology
4. I have created or edited a domain ontology

Expected result: It is not likely that more than a few, less than 20%, knows anything about domain ontologies.

**Metric 4 - Test tool experience**

Measurement procedure: Perform a survey before the experiment.
Question 1: Have you written an automatic test (for example using JUnit)?
Question 2: Have you used an acceptance testing tool (for example FitNesse)?

Expected result: All the participants will have taken a Java programming course, where JUnit has already been written. Some people, 25-50%, may have written automatic tests. Most likely there will not be more than 20% that have used an automatic acceptance tool.

**Metric 5 - Cucumber experience**

Measurement procedure: Perform a survey before the experiment.
Question: Do you have any experience with the test tool Cucumber?
Alternatives:

1. No
2. Heard about it
3. Used it to write a feature
4. Used it to write a feature and implemented the executable test

Expected result: More than 90% will not have used or heard about Cucumber before the experiment.

**Metric 6 - Feature coverage**

Measurement procedure: Subjective evaluation of the features after the experiment by the author of the thesis.
Question: Is the feature completely covered by scenarios?
Alternatives:

1. Yes
2. Partly
3. No

Expected result: A complete set of scenarios for a feature is difficult to achieve even for domain experts. Although a non-strict grading scheme will be used, most features will probably be classified as "partly" complete. The average grade is expected to be between 1,0 and 1,5.

**Metric 7 - Scenario completeness**

Measurement procedure: Subjective evaluation of the scenarios after the experiment by the author of the thesis.
Question: Is the scenario complete?
Alternatives:

1. Yes
2. Partly
3. No

Expected result: Creating complete scenarios may be easier to achieve than creating complete features, as it requires less overview of the whole

domain. Specifying an appropriate initial state (the Given step in Cucumber), is expected to be the most difficult. The average is expected to be between 1,3 and 1,8.

**Metric 8 - Syntactic correct scenario**

Measurement procedure: Subjective evaluation of the scenarios after the experiment by the author of the thesis.
Question: Is the scenario syntacticly correct?
Alternatives:

1. Yes
2. Partly
3. No

Expected result: Syntactic correctness will be judged according to the Cucumber-syntax, which involves using the right keywords at the right places. This should be fairly intuitive, and it is expected that most answers will be given a full score. The expected average is more than 1,8.

**Metric 9 - Unambiguous scenario**

Measurement procedure: Subjective evaluation of the scenarios after the experiment by the author of the thesis.
Question: Is the scenario unambiguous?
Alternatives:

1. Yes
2. Partly
3. No

Expected result: Again, the grading will not be strict as the participants have limited knowledge about the domain. However, scenarios which are clearly vague or can be interpreted in a number of ways will get a reduced score (alternative 2 or 3). It is expected that the majority of the scenarios will not be ambiguous, with the average lying somewhere between 1,6 and 1,9.

**Metric 10 - Verifiable scenario**

Measurement procedure: Subjective evaluation of the scenarios after the experiment by the author of the thesis.
Question: Is is possible to actually verify the scenario?
Alternatives:

1. Yes
2. Partly
3. No

Expected result: Only a few scenarios will not be verifiable. Expected average is more than 1,8.

**Metric 11 - Domain ontology understandability**

Measurement procedure: Survey after the experiment.
Likert statement: Using a domain ontology made it easier to understand the domain?
Alternatives:

1. Strongly disagree
2. Disagree
3. Neutral
4. Agree
5. Strongly agree

Expected result: Using a domain ontology should make it a little easier to understand the domain (alternative 4). The average will be between “neutral” and “agree” (3 and 4).

**Metric 12 - Cucumber understandability**

Measurement procedure: Survey after the experiment.
Likert statement: Writing tests with the Cucumber-style (Given/When/Then) was easy
Alternatives:

1. Strongly disagree
2. Disagree
3. Neutral
4. Agree
5. Strongly agree

Expected result: Agree or strongly agree, as the Cucumber-style is not complex by any means.

**Metric 13 - WikiTest understandability**

Measurement procedure: Survey after the experiment.
Likert statement: The wiki was easy to use
Alternatives:

1. Strongly disagree
2. Disagree
3. Neutral
4. Agree
5. Strongly agree

Expected result: Most participants will agree, meaning the average should be close to “agree”.

**Metric 14 - Ease of browsing tests**

Measurement procedure: Survey after the experiment.
Likert statement: Using a wiki (or text editor) made it easy to get an overview of the tests
Alternatives:

1. Strongly disagree
2. Disagree
3. Neutral
4. Agree
5. Strongly agree

Expected result: Using a text editor may be easier when the amount of requirements and tests is low. It is expected that the average will be between “neutral” and “agree”.

**Metric 15 - Pleasant to use**

Measurement procedure: Survey after the experiment.
Likert statement 1: The user interface of the wiki was pleasant to use
Likert statement 2: The user interface of a text editor (Microsoft Word etc) made it pleasant to write tests
Alternatives:

1. Strongly disagree
2. Disagree
3. Neutral
4. Agree
5. Strongly agree

Expected result: Both the group using the wiki and the group using the text editor will be “neutral” or “agree”.

**Metrics 16 to 18 - Time used**

Measurement procedure: Use the time stamp of when the pre-experiment survey is submitted to the time the post-experiment survey is started.
Metric 16: Total time used
Metric 17: Time used per feature
Metric 18: Time used per scenario
Expected result: Using the text editor may be faster than using a wiki, but the difference is expected to not be over 20% for any of the metrics.

## 8.5 GQM Summary

| | | |
|---|---|---|
| GOAL | OBJECT | WIKITEST |
| | PURPOSE | EVALUATE |
| | FOCUS | USEFULNESS FOR TEST CREATION |
| | VIEWPOINT | CUSTOMERS AND DEVELOPERS |
| | CONTEXT | ACCEPTANCE TESTING USING A DOMAIN ONTOLOGY |
| **Question** | **Q1** | **Test case quality** |
| Metrics | M1 | Programming experience |
| | M2 | Wiki experience |
| | M3 | Ontology experience |
| | M4 | Test tool experience |
| | M5 | Cucumber experience |
| | M6 | Feature coverage |
| | M7 | Scenario completeness |
| | M8 | Syntactic correct scenario |
| | M9 | Unambiguous scenario |
| | M10 | Verifiable scenario |
| **Question** | **Q2** | **Understandability** |
| Metrics | M11 | Domain ontology understandability |
| | M12 | Cucumber understandability |
| | M13 | WikiTest understandability |
| **Question** | **Q3** | **Usability** |
| Metrics | M14 | Ease of browsing tests |
| | M15 | Pleasant to use |
| **Question** | **Q4** | **Efficiency** |
| Metrics | M16 | Time in total |
| | M17 | Time per feature |
| | M18 | Time per scenario |

Table 8.1: Summary of goal, questions and metrics

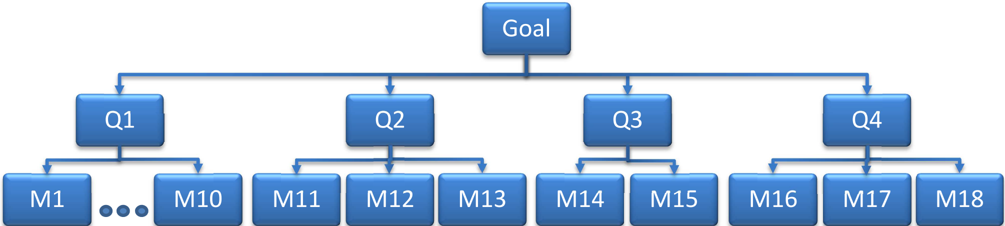


Figure 8.2: GQM tree

# CHAPTER 9

# PLANNING

This chapter describes the planning of the experiment based on the goals defined in Chapter 8. This includes selecting the experiment environment, subjects, the hypothesis and instrumentation. In addition, threats to validity are investigated.

## 9.1 Context Selection

The closer the experiment setting is to an industrial setting, the more valid the results will be. However, this is costly and time consuming compared to a simpler experiment involving less risks as the experiment is likely to be easier to control. The experiment will be conducted off-line, as opposed to on-line, which means it is not performed in a real industrial setting. Having the experiment off-line is more natural when the WikiTest tool is still in an early phase of the development. An on-line experiment could be a natural continuation if the approach is to be evaluated in a larger setting.

The experiment will be carried out using students. The reason is the same as having the experiment off-line; it is cheaper and involves less risk which is applicable for early phase evaluation. The main focus will be on customers of software projects, who may not have much programming experience, if any at all. Second year computer science students will be used.

The problem given in the experiment will be about a steam boiler. The advantage is that the domain is simple and thus comprehensible, but at the same time the domain is large enough to provide room for errors and different

interpretations. Another advantage is that the domain ontology for the steam boiler already exists. The ontology was not created just for the experiment, which means the ontology has not been created based on the knowledge that students would use it in an experiment. A disadvantage is that the problem domain may be difficult to understand in the short time the experiment will last, which could have been avoided by using a familiar problem, for example the creation of a web shop.

## 9.2 Hypotheses Formulation

A hypothesis is a statement that seek to explain some phenomenon or event. The hypothesis is stated so that if the null hypothesis can be rejected, then conclusions can be drawn [38]. The null hypothesis is what is assumed to be true. We have chosen a significance level of 5%. If the p-value is less than 5%, we will reject the null hypothesis.

The hypotheses are used to answer research question number four, RQ4, which is defined in Chapter 2. The hypotheses are used to investigate different characteristics of the test creation process, such as test case quality, efficiency, understandability and usability. These characteristics were derived in the GQM process in Chapter 8.

The null hypothesis will be listed first, followed by the alternative hypothesis that we are investigating. For H1, the null hypothesis states WikiTest has no effect on test quality. An alternative hypothesis, $H1_1$, states that the use of WikiTest leads to higher test quality. There is a second alternative hypothesis $H1_2$, which states that the use of WikiTest leads to *lower* test quality. Both $H1_1$ and $H1_2$ needs to be included to make up the whole sample space. We will, however, only show $H1_1$ to specify that this is what we are looking for. The procedure that will be used is to first see if there is a significant difference, and then if the mean value of WikiTest is higher than the alternative. If there is a significant difference, but where the difference of the mean values are the opposite of what we are investigating, then we cannot draw a conclusion with a one-tailed test.

### Question 1: Test case quality

$H1_0$: Using WikiTest has no effect on test quality
$H1_1$: Using WikiTest leads to higher test quality

$H2_0$: Using a domain ontology has no effect on test quality
$H2_1$: Using a domain ontology leads to higher test quality

$H3_0$: Experience does not lead to a difference in test quality
$H3_1$: More experience leads to higher test quality

$H4_0$: There is no difference in syntactic correctness when using WikiTest or a text editor
$H4_1$: The syntactic correctness is higher using WikiTest compared to a text editor

$H5_0$: Using WikiTest or a text editor leads to the same amount of people writing Backgrounds
$H5_1$: More people will write Backgrounds when using WikiTest compared to a text editor

## Question 2: Understandability

$H6_0$: Understanding the domain is equally easy using either WikiTest or Protégé
$H6_1$: It is easier to understand the domain using WikiTest compared to Protégé

## Question 3: Usability

$H7_0$: There is no difference in how easy it is to get an overview of the tests when using WikiTest or a text editor
$H7_1$: WikiTest makes it easier to get an overview of the tests compared to a text editor

$H8_0$: There is no difference between WikiTest and a text editor in how pleasant it is to write tests
$H8_1$: WikiTest makes it more pleasant to write tests compared to a text editor

$H9_0$: There is difference in ease of use between WikiTest and Protégé
$H9_1$: WikiTest is easier to use than Protégé

### Question 4: Efficiency

$H10_0$: There is no difference in test efficiency between a text editor and WikiTest
$H10_1$: There is a difference in test efficiency between a text editor and WikiTest

$H11_0$: Using a domain ontology has no effect on test efficiency
$H11_1$: Using a domain ontology leads to a difference in test efficiency

## 9.3 Variables Selection

The independent variables are the use of a domain ontology, and the use of WikiTest or a text editor. The dependent variables are the test quality and efficiency.

## 9.4 Selection of Subjects

Subject selection is closely connected to the generality of the experiment. In this experiment, the subjects are all second year computer science students. The students participating have signed up voluntarily as part of collecting money for a student trip. About 30 students will participate, 10 for each treatment.

## 9.5 Experiment Design

The experiment consists of one factor with three treatments. The factor is the tool used to create the tests. The treatments are:

1. **None.** Only a text editor is used to write the tests.
2. **Ontology.** Protégé will be used to view the domain ontology and a text editor will be used to write the tests.
3. **WikiTest.** WikiTest will be used both for the domain ontology and writing the tests.

Treatment one and two use a text editor, while treatment three uses a Wiki. Treatment two and three use a domain ontology, while treatment one does not have this information.

The students will be assigned randomly to each treatment. In addition, the experiment design principle called *balancing* is used so that each treatment has the same number of subjects.

ANOVA [39] will be used to compare the groups. t-test [8] will be used when comparing the wiki to a text editor, WikiTest to Protégé and the domain ontology versus no ontology. for the evaluation of the hypotheses.

## 9.6 Instrumentation

Two surveys will be performed for each subject; one before the experiment and one after. The surveys will be answered using an online questionnaire. This makes it easier to analyze the results. In addition, using an online survey tool makes it possible to log the time from the pre-experiment questionnaire is submitted to the time the post-experiment questionnaire is started. This gives us information about how much time it takes to use the two approaches.

The instruction set will provide information about:

- Assignment description and tasks
- Steam boiler information
- Cucumber information

The ABB boiler pilot application has six functional boilerplate requirements and four safety requirements, for a total of ten requirements. The task will contain:

- Two functional requirements. These are complete features where the scenarios are filled in.
- Two functional and two safety requirements where only the requirement text is written at the top of a feature, the student will fill in the rest.
- One functional and one safety requirement with poorly defined tests, which the student shall try to improve.
- For the last functional and safety requirement (one of each) the complete feature has to be filled in by the student, simulating a situation where a requirement is discovered during the test specification.

For the two groups using a domain ontology, a few domain specific questions will be asked. These control questions are used to make the students a little more familiar with the ontology.

## 9.7 Validity Evaluation

Incorrect data are dangerous, even more so than having no data at all. As for experiments, the validity of the results is crucial if any conclusions can be made. Thus, it is important to identify any threats to validity already at the planning phase of the experiment.

Wohlin et al. provide a checklist of validity threats [8]. This list will be used to investigate the validity threats for this particular experiment. The threats in the list are divided into four categories, depending on what kind of threat it is. *Conclusion validity* refers to the ability to justify the relationships that are found in the experiment. *Internal validity* refers to the validity of the cause-effect relationship. *Construct validity* is the link between theory and the observation. *External validity* is the degree to which the results can be generalized to a larger scope.

Each threat will be given a priority of either high, medium or low. The priority is given based on the threat's importance for this experiment only, and is not an evaluation of the priority of the threats in general. A threat with high priority is a threat that is particular important due to the nature of the experiment, which means that the threat will need to be mitigated or accepted as an aspect that may skew the results of the experiment. A threat with medium priority is a lesser threat, but will still be important to consider. This could *either* be that the probability of the threat occurring is low, *or* that the impact it has when it occurs is low. A low priority threat is any threat not applicable to this experiment, or where *both* the chance of it occurring *and* the impact it has when it occurs, is low.

### 9.7.1 Conclusion Validity

**Low statistical power (High)**
Statistical power is the probability that the test will reject the null hypothesis if the null hypothesis is false. If we have low statistical power, the risk of not being able to reject an erroneous hypothesis increases. This threat is often high in master thesis experiments involving students, where the number of subjects involved may be low. When comparing the difference between using a domain ontology and not using it, it is possible to aggregate group B and C, which both use the ontology, and compare it to group A, which does not use the ontology. When comparing the difference between using a text editor and WikiTest, it is possible to aggregate group A and B, which both use a text editor, and compare it to group C, which use WikiTest. This aggregation

will make it possible to get higher statistical power, as compared to when the three groups are compared individually.

**Fishing and the error rate (Medium)**
Fishing in a statistical context refers to playing around with data until you turn up something that supports your hypothesis. When conducting investigations using the rather arbitrary significance level 0.05, it means that 1 out of 20 investigations will give a significant result just due to luck. When multiple investigations are performed, the chance increases of finding at least one investigation which rejects the null hypothesis. This may happen in this experiment, due to the high number of hypothesis that are tested.

**Reliability of measures (Medium)**
Reliability of measures refers to the ability to get the same outcome if the experiment is performed multiple times. The instrumentation papers and questionnaires are static and would be the same for multiple experiments, which provides the same foundation for each experiment. However, human judgment is used to grade the quality of the tests, and this measure may not be reliable if different people do the grading. The reliability of grading the tests written in the experiment is mitigated by creating a set of grading instructions for the different aspects of test case quality. As an example, the syntactic grade is reduced by 1 point if one error in the use of Cucumber keywords and syntax is wrong, and reduced with 2 points if more than one error is found.

**Violated assumptions of statistical tests (Low)**
Some might argue that transforming the five point ordinal Likert scale to a five point ratio scale should be avoided. Doing this seems to be common in many research papers nowadays, as pointed out by Blaikie in [40]. Statisticians such as John Tukey argue that an over-purified view of what measurements are like should not dictate how data is to be analyzed [41]. The use of a ratio scale should not pose a big threat to the validity of the experiment.

**Reliability of treatment implementation (Low)**
The instructions and work flow are about the same for the three treatments applied in the experiment, and should not pose much of a threat.

**Random irrelevancies in experimental setting (Low)**
The experiment will be performed in a lecture hall at NTNU, where noise and distractions should be at a minimum. A potential factor is that each subject

brings his own PC, where things such as virus may disturb the experiment. This is not regarded as a big threat, as the process of writing tests does not suffer much from a sudden, short interrupt.

**Random heterogeneity of subjects (Low)**
Some extent of heterogeneity will always exist in experiment, but using students at the second year of computer science at NTNU provides a group that will have taken the same courses, providing the same minimum foundation for everyone. Certainly, there is a chance that some students may have extensive programming or test experience, which the rest does not have. This threat will be mitigated by the use of a pre-experiment survey where each subject states his experience level within programming, automated testing, acceptance testing, testing with Cucumber, and so forth. If just one or two participants have much more experience than the rest, the results for these participants could be excluded to reduce this threat

### 9.7.2 Internal Validity

**Instrumentation (High)**
The instrumentation is important for the experiment as a whole. Each subject will have quite a lot of material to read, and it is important that this information is not written in a way that helps one of the groups more than the others. Group B and C will need to answer some domain specific questions to make sure that they at least try to use the domain ontology. These control questions will not cover topics that may help these two groups writing the tests afterwards. The instrumentation is a core part of the experiment and the quality of the instrumentation is regarded as a high priority.

**Selection (Medium)**
The subjects are volunteers, which means they may be more motivated than the average student. A way to mitigate parts of this threat is to not give out more information than necessary before the experiment is started. Another problem with the selection is that the students may not be an accurate representation of *customers* of IT projects. On the other hand, the students are a good representation of junior programmers.

**Ambiguity about direction of causal influence (Medium)**
The nature of the experiment is to compare WikiTest to a text editor, and to compare the use of a domain ontology to not using an ontology. In the midst of doing this comparison, there may be causal influences that are not properly identified before the experiment is started. It is a threat, but not a

critical one, as the main goal is to assess the usefulness of WikiTest.

**Diffusion or imitation of treatments (Medium)**
There is a threat that one group learns about the treatment of another group, such as if the participants sit too close and can look at each others work. This is not a threat when it comes to the test quality, as it does not matter if a subject writing tests in a text editor learns about the wiki. On the other hand, it may affect the response of that subject in the post-experiment questionnaire. Knowing the another group used a wiki, the subject may answer more negatively (or positively) on questions regarding the use of a text editor.

**Compensatory rivalry (Medium)**
This is related to diffusion or imitation of treatments. A subject first has to learn about another treatment for it to be a threat. There is a threat that subjects using the wiki may deduct that this is the new method, and give it a higher or lower score that he would have done under different circumstances.

**Interaction with selection (Medium)**
There is a chance of selection-maturation where group A that does not use a domain ontology has the least amount of material to read, and may be able to spend more time writing tests. This threat will be investigated by asking the subjects if they had enough time to write tests.

**History (Low)**
Not applicable since the treatments are applied to everyone at the same time.

**Maturation (Low)**
There is a chance that the subjects learn how to write better tests as they wrote more tests, thus the tests written last may have higher test quality. This does not pose a threat, however, as this is a common factor for all the three groups.

**Testing (Low)**
The test is not repeated, and no feedback will be given to the subjects during the experiment, so this does not pose a threat.

**Statistical regression (Low)**
Not applicable as the subjects are not classified based on any previous experiment.

**Mortality (Low)**
Since the subjects are paid only if the attend the experiment, they are not likely to drop out during the execution of the experiment, as could happen if the experiment was conducted purely online without any form for reward.

**Compensatory equalization of treatments (Low)**
This is a low threat as there is no extra compensation to any of the treatments.

**Resentful demoralization (Low)**
This resembles compensatory rivalry, but where the subject put less effort into doing a good job to lower the score of his treatment. To some extent, it is possible to identify this by looking at the number of tests the subject wrote. If he wrote very few, and still said he had enough time, it may be an indicator that the subject did not put in the same effort as others.

### 9.7.3 Construct Validity

**Mono-method bias (High)**
Mono-method bias refers to the way the measurements are performed. It is a high threat in some parts of this experiment, such as the human judgment of test case quality, which can not be cross-referenced with other measurements. For other parts, such as asking the subjects if the interface is pleasant to use, it can be cross-referenced by asking if more help was needed with the tool.

**Hypothesis guessing (High)**
Hypothesis guessing is a high threat as the subjects know they are attending an experiment where not everyone will do the same (the different instructions depending on the group) and they may guess what the hypotheses is. Especially in the post-experiment questionnaire, the questions may give a pointer to what the hypotheses is. As the types of questions in the survey is quite disperse, it should be hard to guess the hypotheses.

**Experimenter expectancies (High)**
Most of the material for the experiment is made by the author of this thesis, which is a threat as the instrumentation could be tailored to give a specific result. Fortunately, the domain ontology and the requirement specification are made in another context (CESAR). In other words, the domain ontology and requirements were not created to be used only for this experiment. It would be possible to further reduce this threat by involving other people, who do not know anything about the hypotheses, in the grading of the test

case quality.

**Inadequate preoperational explication of constructs (Medium)**
Test case quality is divided into five areas to make it more explicit. It is possible that some constructs are not sufficiently defined in the planning stage and must be defined after the experiment. This will have a bad impact on the validity of the experiment. Anything defined after the actual experiment will have to be commented.

**Mono-operation bias (Medium)**
Mono-operation bias is a threat if if only a single independent variable is used. A solution is to implement multiple versions, e.g. different problem domains, to make sure the results are valid for other domains than the steam boiler.

**Confounding constructs and levels of constructs (Medium)**
This threat is identified by asking the subjects about their experience. It will be a problem if a majority of the subjects in one group has much more experience than the subjects in another group, as it will make it difficult to see if the difference is because of the experience or the treatment.

**Interaction of different treatments (Low)**
Not applicable as only one treatment is used.

**Interaction of testing and treatment (Low)**
The subjects will only know that they are to write tests for a given set of requirements, and that these tests should follow a specific syntax. They will not know if the goal is to write the most tests, be finished the fastest or to use the most domain concepts.

**Restricted generalizability across constructs (Low)**
This threat is concerned with side effects. The most important one is test efficiency, which will be measured to see if there is a large different between the treatments. Having open questions in the post-experiment questionnaire makes it possible for the subjects to write about concerns which was not explicitly covered earlier in the experiment.

**Evaluation apprehension (Low)**
The fear of being evaluated should not be a threat as the subjects are volunteers, anonymous and used to be graded based on their work.

### 9.7.4 External Validity

**Interaction of selection and treatment (Medium)**
The selection of people used as subjects have an impact on the ability to generalize the results. This is a medium threat as the students are an approximate representation of programmers, but they do not reflect the views of *customers* of IT projects.

**Interaction of setting and treatment (Low)**
The setting is different from real use as the subjects work alone, and not together as is likely to happen when the tests are created. The domain ontology and the steam boiler requirements document from the industry helps to create a more realistic setting.

**Interaction of history and treatment (Low)**
The day the experiment is performed should not be a high threat to the validity.

### 9.7.5 Validity Summary

There is a trade-off between the four types of validity. Putting more effort on one may reduce another. As Table 9.1 shows, there are five threats of high risk in the experiment. Three of these are construct validity threats, which is about the ability to generalize the results. This is mostly because experiments in general are easy to control, but the more they are controlled the more artificial they become, which in turn makes it harder to generalize the results.

| Type | Threat |
| --- | --- |
| Conclusion | Low statistical power |
| Internal | Instrumentation |
| Construct | Mono-method bias |
| | Hypothesis guessing |
| | Experimenter expectancies |

Table 9.1: High priority validity threats

Low statistical power may make it difficult to get any significant results. This is a threat because it may be difficult to get enough subjects to participate. This risk will be reduced by involving the second year computer science excursion committee early on so they can promote the experiment early. It is also possible to include excursion committees from other field of studies to

increase the number of subjects, although this will increase the heterogeneity of the subjects.

Instrumentation is critical for the experiment, and care needs to be taken when creating the tasks and instructions. The threat of mono-method bias and experimenter expectancies can be reduced by being explicit about how the test quality will be graded, and involve another person outside the experiment to go over the experiment plans. Hypothesis guessing is natural for subjects to do when they attend the experiment, but since the post-experiment questionnaire touch a wide range of topics, it should difficult to guess the hypotheses.

Table 9.2 displays the validity threats classified as medium risks in this experiment. There is a total of eleven such threats, with all validity types represented.

| **Type** | **Threat** |
|---|---|
| Conclusion | Fishing and the error rate |
| | Reliability of measures |
| Internal | Selection |
| | Ambiguity about direction of causal influence |
| | Diffusion or imitation of treatments |
| | Compensatory rivalry |
| | Interaction with selection |
| Construct | Inadequate preoperational explication of constructs |
| | Mono-operation bias |
| | Confounding constructs and levels of constructs |
| External | Interaction of selection and treatment |

Table 9.2: Medium priority validity threats

# CHAPTER 10

# OPERATION

In this chapter, the operation of the experiment is explained. The operation consists of three steps: preparation, execution and data validation.

## 10.1 Preparation

The preparation is based on the planning described in Chapter 9. The participants were recruited through a class excursion committee for the second year students taking Computer Science at NTNU. The committee is given some money, according to a standard rate at NTNU, for each student that attends the experiment. Before the experiment, the only thing they knew was that the experiment's theme was that it had to with testing, and that they had to bring their own PC.

## 10.2 Execution

The experiment was held on the 4th of May in a classroom at NTNU with a total of 38 students. The experiment lasted from 14:15 to 16:00, a total time of 105 minutes. About 15 minutes were used on a presentation describing how the experiment would be executed, as well as some information about the steam boiler and Cucumber. The same information was given to each participant through hand-outs; the participants did not have to memorize the information given in the presentation. The hand-outs had a description of the assignment and a link to a webpage the participant should access. Each group had a different webpage, with the specific instructions for that group.

The online survey tool Surveygizmo (http://www.surveygizmo.com/) was used for the pre-experiment and post-experiment questionnaires. Group A and B wrote the features in text files, while group C used WikiTest to write the features. Each participant in group C had their own wik, where the ontology data was set up for each participant. If the purpose of the experiment was to assess the capabilities of collaboration, it would make sense to use the same wiki for all the subjects in group C.

## 10.3 Data Validation

38 students participated: 12 in group A, 13 in group B and 13 in group C. A possible source of error is that some participants have misunderstood the instructions, not delivered the files or not taken it seriously.

One participant in group C reported problems with his PC during the experiment, and the participant also wrote about it in the post-experiment comments. This student had only time to write two features, and did not deliver the pre-experiment questionnaire. Another participant, from group B, did either not understand the instructions, or (more probably) did not take the experiment seriously. His delivery consisted of two features, where one of them was written as a poem not relevant to the experiment. This participant answered the domain questions, but had not tried to create any features properly. For the data of the two participants, the following options were possible:

1. Exclude only the written features, but keep the comments from the questionnaires
2. Exclude the written features and post-experiment questionnaire, but keep the pre-experiment questionnaire
3. Exclude everything

The post-experiment questionnaire was based on the work done creating the Features, and as such doing little work creating Features gives less experience to talk about. The pre-experiment questionnaire might be valid, but it has little use by itself. The participant of group C did not answer the pre-experiment questionnaire, while the participant from group B did (although he did not take the latter part of the experiment serious). As such, the best option is likely the third one. All the data from the two participants were excluded from the analysis.

# CHAPTER 11

# DATA ANALYSIS

This chapter contains the analysis of the data collected in the experiment. First, descriptive statistics for each measurement is given. Then, the hypotheses are tested using statistical tests. A discussion of the data calculations and their implications is given in Chapter 12.

## 11.1 Measurements

This section will describe the data collected for the metrics defined in Section 8.4. For group A, 12 out of 12 delivered the pre-experiment questionnaire. For group B, 10 out of 12 delivered, and for group C, 11 out of 12 delivered. The measurements are visualized in diagrams to make it easier to get an overview of the data. Each measurement will be compared to the expected value of the metric given in Chapter 8.4.

### 11.1.1 Metric 1 - Programming experience

Figure 11.1 shows the programming experience of the subjects distributed over the three groups, where each group has 12 subjects. As can be seen, no participant has more than 8 weeks of industrial experience, and only four participants has more than 2 weeks of industrial experience. It was expected that over 80% would belong to option one or two, and the actual result was about 88%. Although the industrial experience does not give an exact measure of the subjects actual programming experience, it is enough to get an idea of the experience of the subjects as a whole.

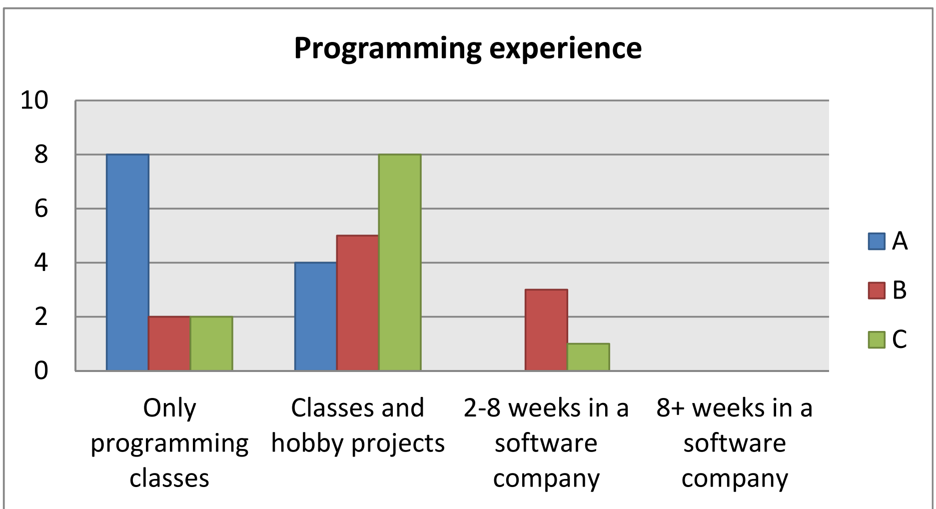


Figure 11.1: Programming experience

Group A has a larger amount of subjects with only programming experience from school compared to the other groups. If experience is a factor, then that may have an effect when comparing the use of domain ontology (B and C) to not using a domain ontology (A). When comparing a text editor (A and B) to the use of a wiki (C), the distributions are similar.

### 11.1.2 Metric 2 - Wiki experience

It was expected that more than 80% would have read wiki articles. The actual experience of the subjects in group C was higher, as 36% had done more that just read articles. Two subjects had even had their own wiki server, which is more than expected. From a generalizability point of view, the wiki experience of the subjects is likely to resemble the general experience of programmers, more so than the average IT customer who is expected to have less experience.

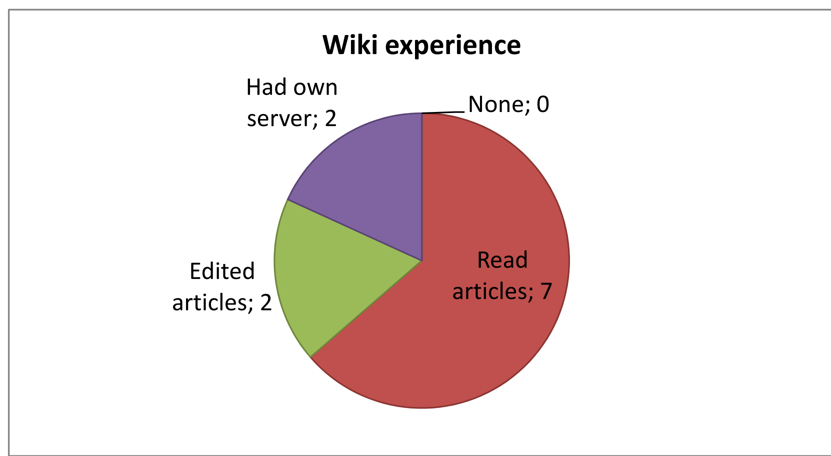


Figure 11.2: Wiki experience

### 11.1.3 Metric 3 - Ontology experience

The expected ontology experience was that only a few would know anything about it. The pre-experiment survey showed that only 1 of 10 subjects in group B had heard about it, while none out of 11 subjects in group C had any knowledge about it. This is in conformance with the expectancies which assumed that less than 20% would have any experience with ontologies.

### 11.1.4 Metric 4 - Test tool experience

None of the subjects had ever used any acceptance testing tool, such as FitNesse. Figure 11.3 shows that 13 subjects had written an automatic test, for example using JUnit, while 20 had not. The distribution between the three groups is fairly equal. It was expected that 25-50% would have written tests, and less than 20% would have used an automatic acceptance tool. The results were as expected, as 43% had written tests before, while 0% had used acceptance testing tools.

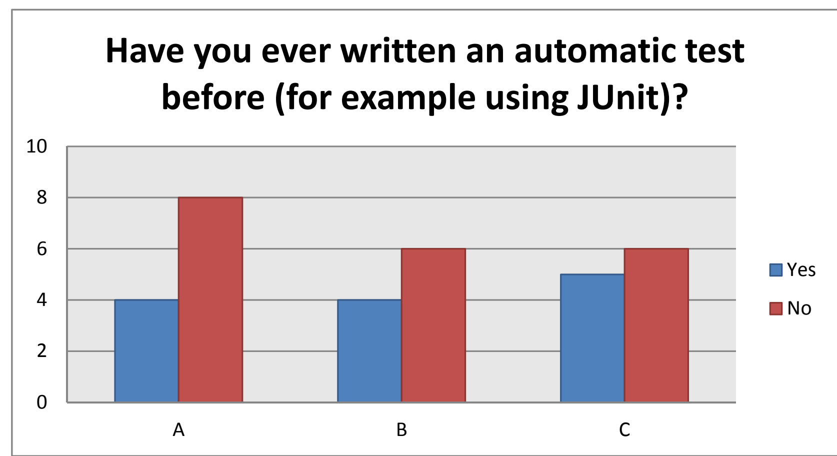


Figure 11.3: Written automatic test before

### 11.1.5 Metric 5 - Cucumber experience

As Figure 11.4 shows, only a few had even heard about Cucumber before the experiment. This was expected, and should be a good representation of an IT customer. The downside is that there is less use analyzing the test quality based on Cucumber experience when the large majority has no experience with the tool. As with the programming experience, the experience of the subjects is quite homogeneous.

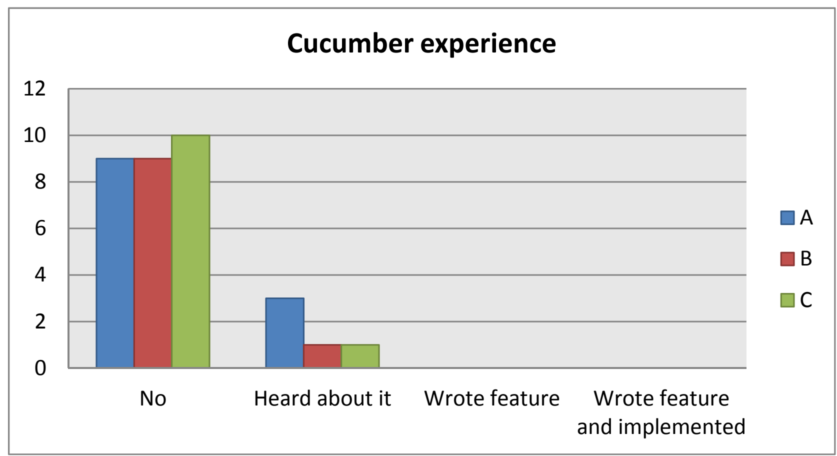


Figure 11.4: Experience with Cucumber

### 11.1.6 Metric 6 - Feature coverage

Figure 11.5 shows a box plot of the feature coverage for the three groups. For each subject, the feature coverage is found by giving each feature a score of 0 (the scenarios do not cover the requirement), 1 (the scenarios partly cover the requirement) or 2 (the scenarios cover the requirement). Then, for a particular participant the score for each feature is summarized and divided by the number of features that the participant have tried to write. This means that the features that do not have any scenarios are not included in the average. In the definition of the feature coverage metric, it was expected that the majority of the features would be partly complete, with an average between 1.0 and 1.5. It turned out that most subjects managed to write scenarios that covered the scope of the requirement, and that the expectancy was too low.

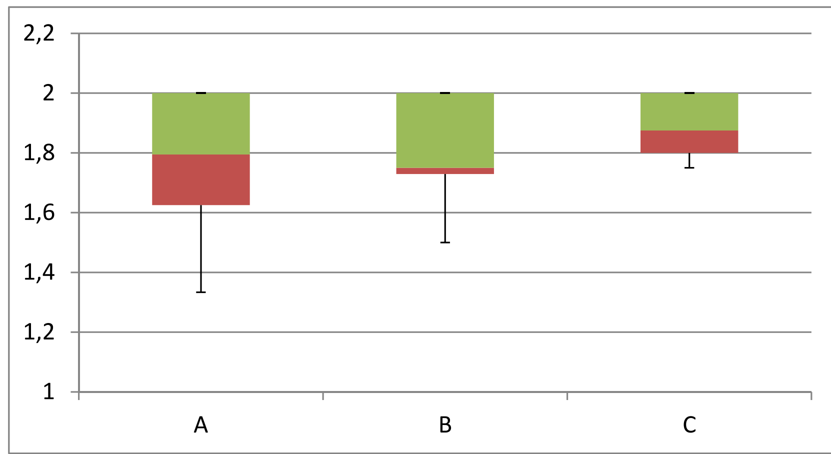


Figure 11.5: Feature coverage

### 11.1.7 Metric 7 - Scenario completeness

Specifying complete scenarios was more difficult for the participants than feature coverage, although all the groups had an average within the expected interval from 1.3 to 1.8. The reason why the feature coverage gave higher results than expected, and scenario completeness did not, might be because the steam boiler requirements were detailed (as opposed to general). This makes it easier to get complete coverage for the feature, but does not make it easier to specify a complete scenario.

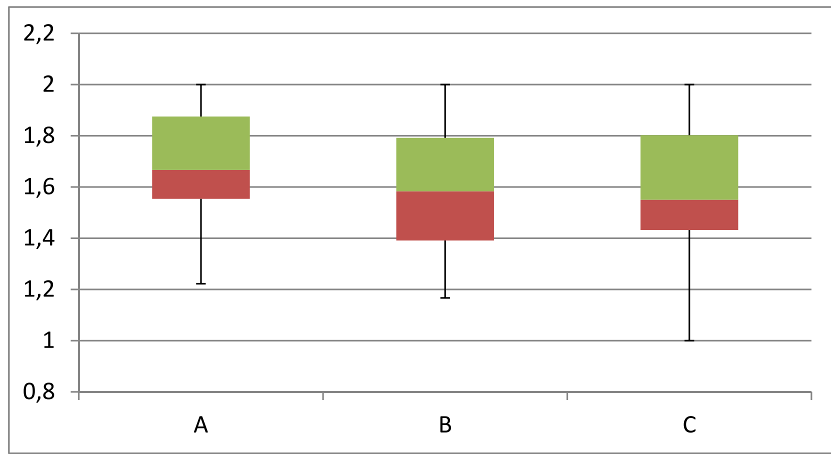


Figure 11.6: Scenario completeness

### 11.1.8 Metric 8 - Syntactic correct scenario

Writing features and scenarios with correct syntax was easy for the majority. In group A, 8 out of 12 got a full score. In group B, 7 out of 12 got a full score. In group C, 9 out of 12 got a full score. The results agreed with the expected average of more than 1.8.

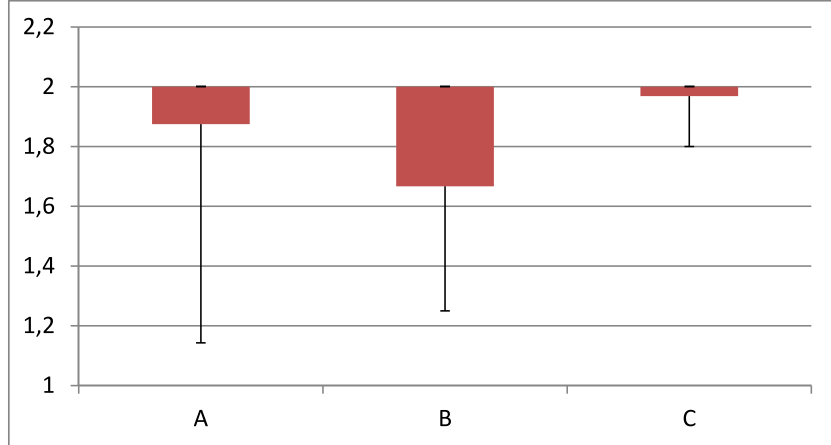


Figure 11.7: Syntactic correctness

### 11.1.9 Metric 9 - Unambiguous scenario

Figure 11.8 shows the ambiguity score for the groups, where the higher score is the less ambiguous the scenarios are. The mean for group A, B and C is 1.82, 1.92 and 1.98, respectively. The expected mean was between 1.6 and 1.9, so two of the groups had a higher mean than expected.

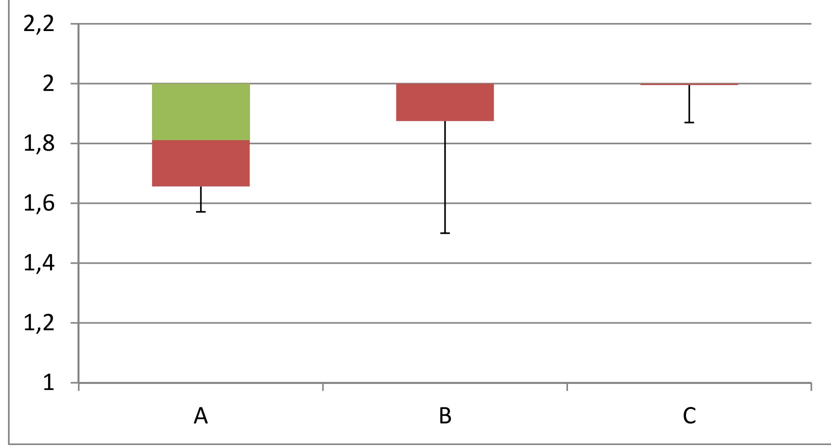


Figure 11.8: Ambiguity score

### 11.1.10 Metric 10 - Verifiable scenario

The verifiability of the scenarios for each group is given in Figure 11.9. The mean for group A, B and C is 1.80, 1.78 and 1.91, respectively. Group B had a mean less than the expected minimum value, which was 1.8 points.

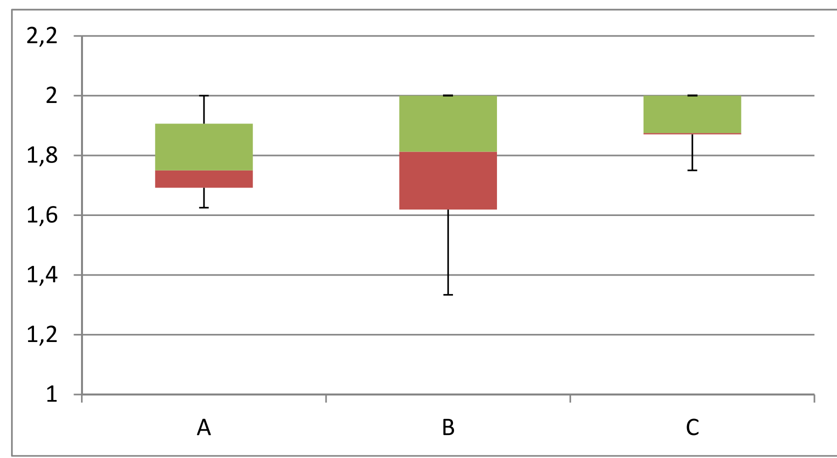


Figure 11.9: Verifiability

### 11.1.11 Metric 11 - Ontology understandability

This metric is collected in a survey after the experiment and is only relevant for group B and C. More people in group B think that using a domain ontology helped to understand the steam boiler domain. In addition, Figure 11.11 shows how many think the ontology makes it easier to write tests.

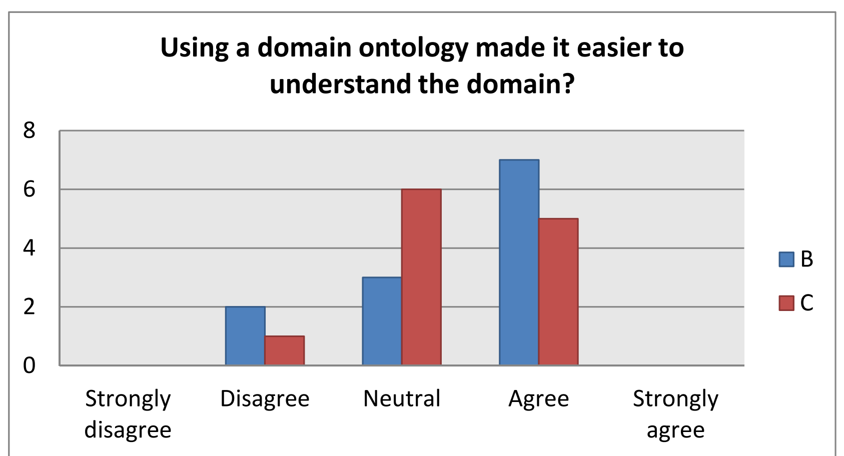


Figure 11.10: Ontology makes the domain easier to understand

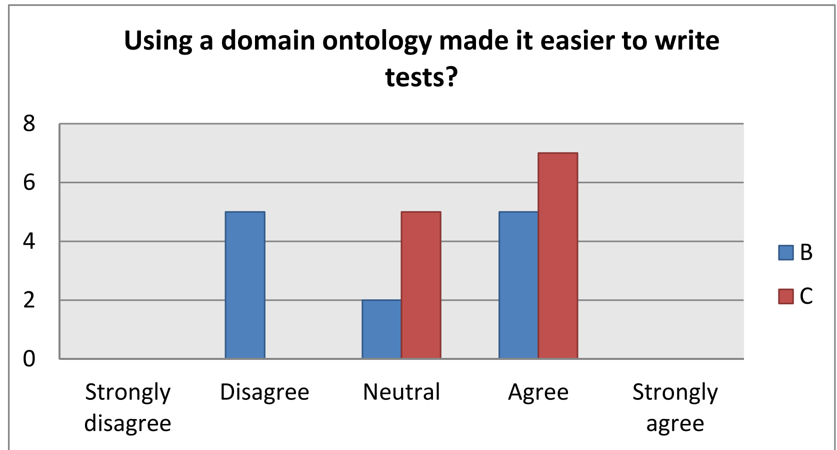


Figure 11.11: Ontology makes it easier to write tests

It is also interesting to see how many would like more help about the steam boiler for all the three groups. Figure 11.12 shows that no one strongly disagree that they would like more help. Four subjects strongly agree that they would like more help, two from group A and two from group B.

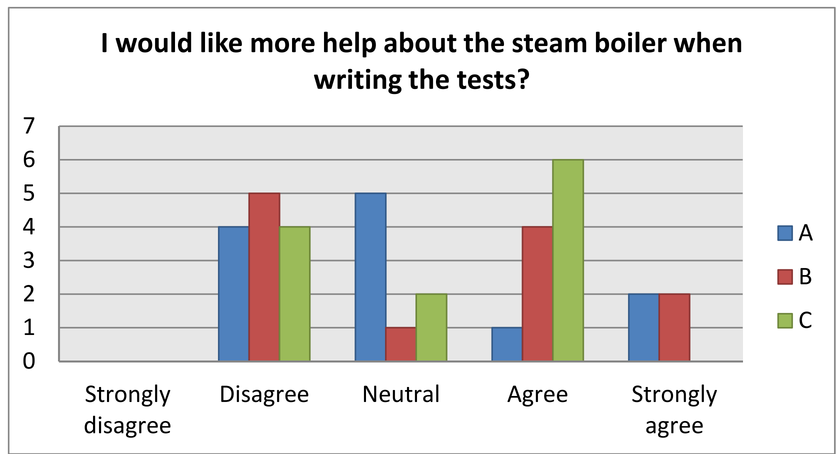


Figure 11.12: Steam boiler help

### 11.1.12 Metric 12 - Cucumber understandability

It was expected that the Cucumber syntax would be easy to understand and use. As shown in Figure 11.13, group C had a higher variation in their answers than the other groups. It is possible that the effect of using the wiki creates a higher variance within the group compared to a text editor. The cause of this should be explored further, such as by reading the post-experiment comments from the participants of the experiment.

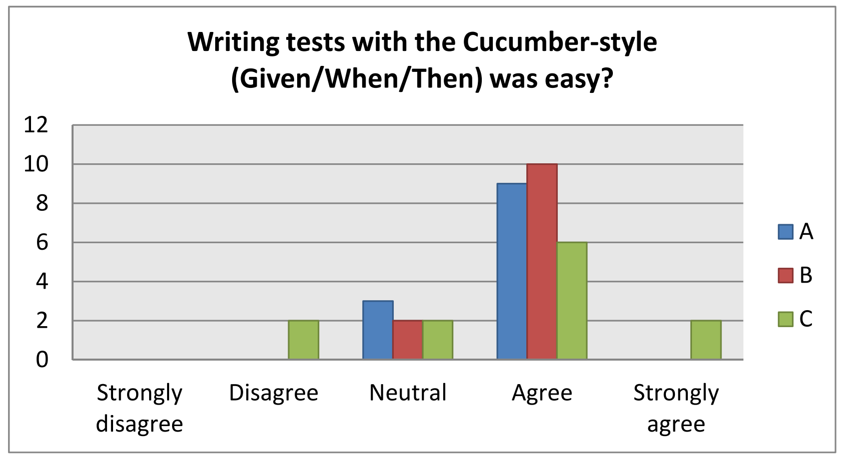


Figure 11.13: Difficulty of using the Cucumber syntax

Figure 11.14 shows how many would like more help about Cucumber, and Figure 11.15 shows how many would rather write tests in a free format, as opposed to using Cucumber.

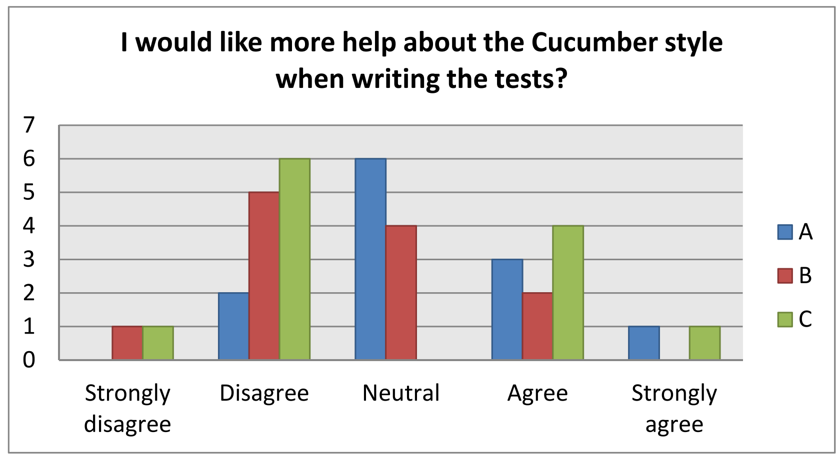


Figure 11.14: Cucumber help

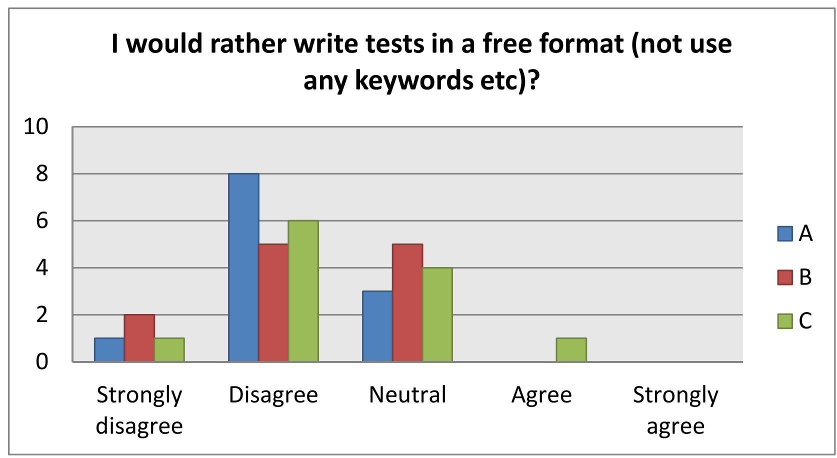


Figure 11.15: Rather write tests in free format

### 11.1.13 Metric 13 - WikiTest understandability

The understandability of WikiTest is specific to group C, and the pre-experiment expectation is that most subjects will agree. The same question about the perceived ease of use was asked to group B, but then in the context of Protégé.

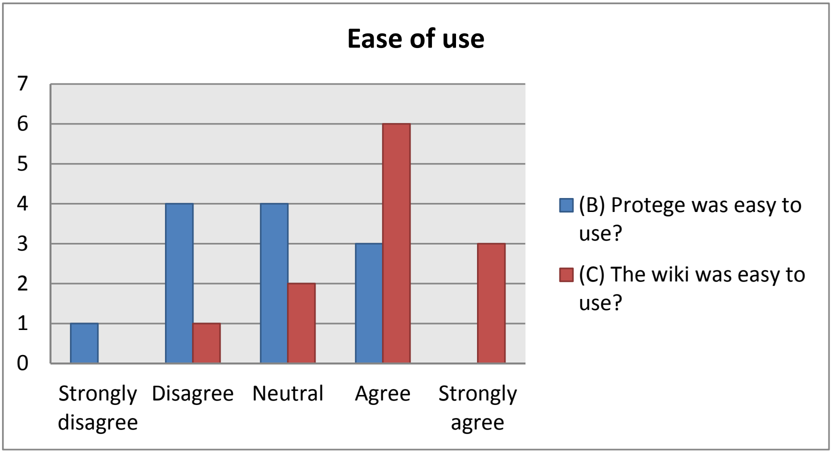


Figure 11.16: Ease of use

Figure 11.17 shows how many in group C would like more help about the wiki. This is another way of finding out if WikiTest was easy to use.

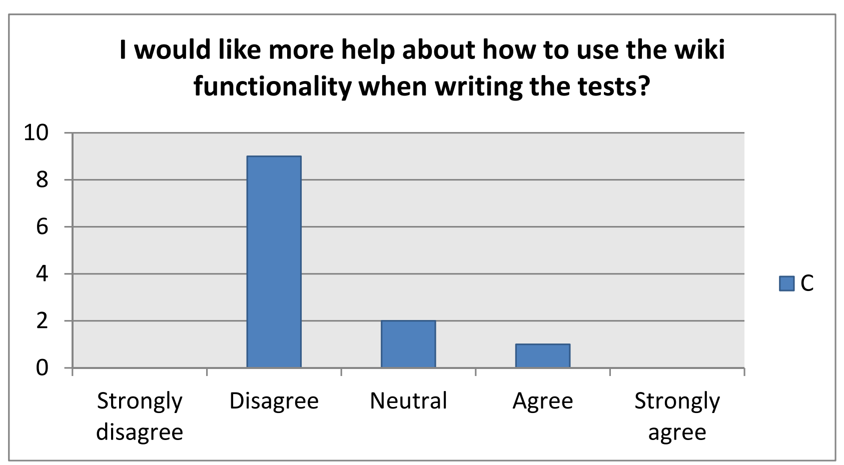


Figure 11.17: More help for WikiTest

### 11.1.14 Metric 14 - Ease of browsing tests

The metric "ease of browsing tests" is used to investigate how easy the text editor or wiki is to use when the amount of features grows large. As this experiment only used 10 features, the measurement might not represent its intention. Group A and C were within the expected interval value between 3.0 and 4.0. Group B had a lower score than expected, with an average at 2.8.

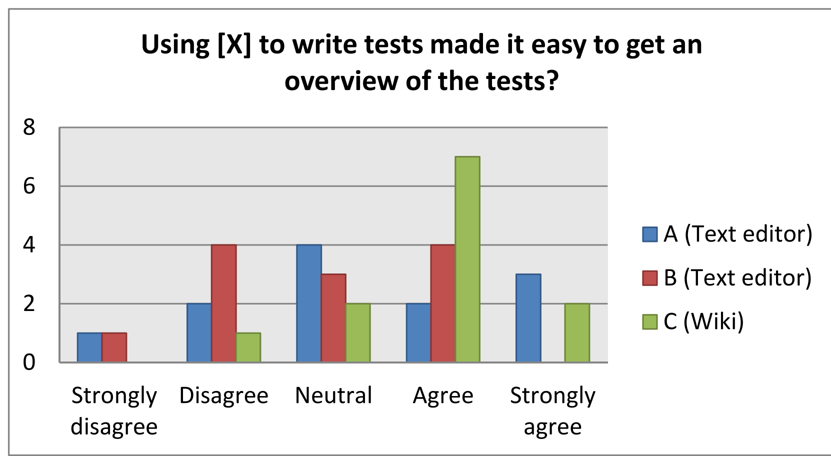


Figure 11.18: Overview of tests

### 11.1.15 Metric 15 - Pleasant to use

The expected result of how pleasant the user interface is to use is that most subjects will be neutral or agree. It would be a warning sign of poor wiki implementation if the text editor is a lot easier to use than the wiki, but as can be seen in Figure 11.19, the majority of subjects who used the wiki liked the user interface. If the goal is to compare the user interface of the wiki and text editor, it would make sense that the subjects used both and compare them to each other, but this was not done in the experiment.

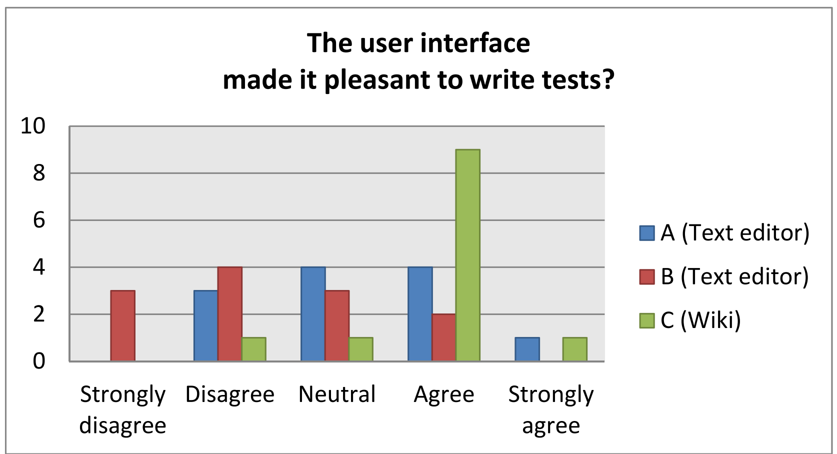


Figure 11.19: User interface satisfaction

### 11.1.16 Metrics 16 to 18 - Time used

Metrics 16 is the total time used, which is listed for each participant in Figure 11.20. Figure 11.21 shows little difference in how many needed more time than they got. Only four subjects said they did not have enough time to complete the whole experiment.

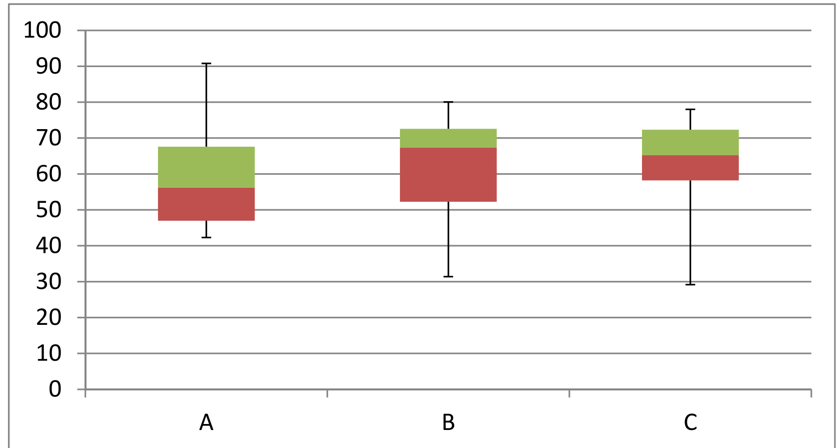


Figure 11.20: Time used at the assignment

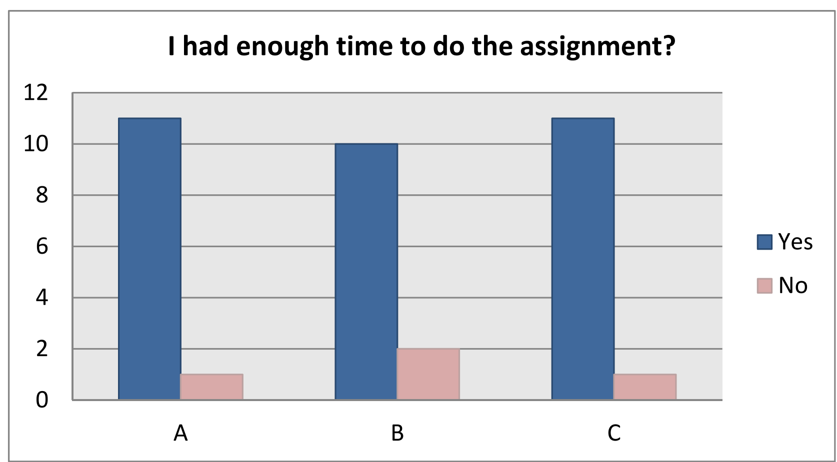


Figure 11.21: Enough time to do the assignment

Metric 17 is the time used per feature, which is depicted in Figure 11.22.

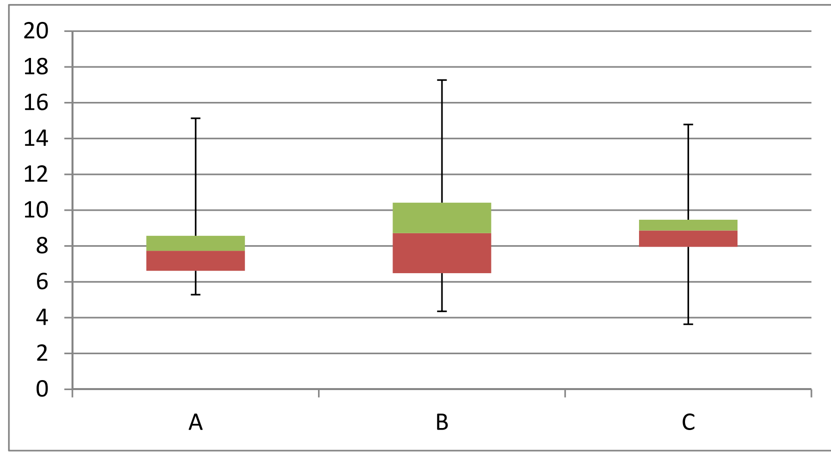


Figure 11.22: Time used per feature

Metric 18 in Figure 11.23 shows how many minutes is used per scenario for the different groups. Again, the values between the groups are quite similar, and the mean is within the expected difference of 20%.

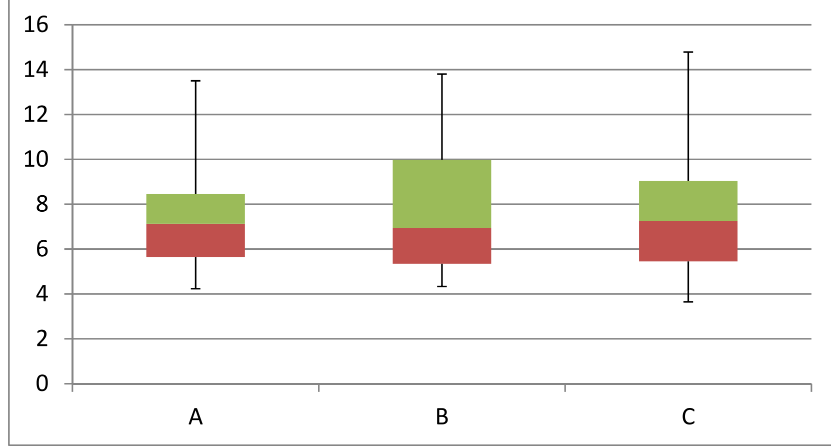


Figure 11.23: Time used per scenario

## 11.2 Hypotheses Testing

This section tests the hypotheses posed in Chapter 9.2. The data has been tested using ANOVA when comparing the mean of the three groups. t-test has been used when the wiki is compared to the text editor, and when the use of an ontology is compared to not using an ontology. The data of the statistical tests can be found in Appendix B. Throughout the hypotheses testing, the significance level used is 5%. This means that the p-values need to be 0.05 or smaller to be defined as statistical significant.

### 11.2.1 Question 1: Test case quality

The question of test case quality involves five hypotheses. $H1_0$ says that using WikiTest has no effect on test quality, while $H1_1$ says that using WikiTest leads to higher test quality. To get the complete sample space, we need a third hypothesis, $H1_2$, stating that the use of WikiTest leads to lower test quality. As explained in 9.2, we will look for a one-tailed p-value of 0.05 or less, and look at the means to make sure the result is the way we are investigating ($H1_1$, not $H1_2$).

The test case quality is found by combining M6-M10. The ANOVA in Table B.1 shows the average points per feature between the three groups. Group A and B has a mean of 9.1 each, while C has a mean of 9.6. The difference is not significant with a p-value of 0.0583. However, we really want to compare the difference between the use of WikiTest and a text editor, which is found by using a t-test. The results are found in Table B.2. With a mean of 9.1 for text editor and 9.6 for wiki, the difference is significant with a p-value of 0.00117 and the wiki has a higher mean. Thus, we **reject $H1_0$ and accept $H1_1$**, concluding that WikiTest leads to higher test quality.

A similar test is performed on the effect of an ontology. $H2_0$ says that using a domain ontology has no effect on test quality, while $H2_1$ states that the use of a domain ontology leads to higher test quality. Table B.3 shows that the mean is 9.1 for no ontology and 9.3 when using an ontology, which is not significant with a p-value of 0.154. The conclusion is that we **cannot reject $H2_0$**. Using a domain ontology has no effect on test quality.

The third hypothesis investigates the effect of experience on test quality. $H3_0$ says that experience does not lead to a difference in test quality, while $H3_1$ says that more experience leads to higher test quality. When designing the experiment, it was not know what type of experience the subjects would

have. If everyone had the same experience, it would not be possible to investigate this hypothesis. As it turned out, only one participant had heard of domain ontologies (metric 3), so this was not investigated. Wiki experience (metric 2) had a majority of participants who had read articles, but there were not enough participants who had more experience than this. The three experience metrics analyzed are programming experience (M1), automatic test tool experience (M4) and experience with Cucumber (M5). As this is a multiple analysis, the significance level becomes $(1-0.05)^3 = (1-0.14)$. To continue to use 5% as the significance level, each of the following experience will need a p-value less than 0.017, since $(1-0.017)^3 = (1-0.05)$.

The results for programming experience is given in Table B.4. Since most of the subjects did not have industrial experience, we decided to use two groups. The first group is those with only programming experience from school, while the second group is everyone who has more experience than that. The mean for the first group is 9.1, while the mean of the second group is 9.4. The p-value is 0.117, so we cannot reject $H3_0$ with regards to programmer experience.

Automatic test tool experience is displayed in Table B.5, where the subjects who has written an automatic test before has a mean of 9.30, while the subjects who has not written any automatic tests has a mean of 9.28. The difference is neither big nor significant, with a p-value of 0.447.

The last experience metric to investigate is Cucumber experience, which is analyzed in Table B.6. There are two groups, those who have heard about Cucumber before and those who have not. The means are 9.12 and 9.32, respectively. The group with less experience has a higher score so we cannot draw any conclusion, as $H3_1$ only state that *more* experience leads to higher test quality.

None of the experience metrics give any significant results, so **$H3_0$, saying that experience does not lead to a difference in test quality, is not rejected**.

$H4_0$ states that there is no difference in syntactic correctness when using WikiTest or a text editor, while $H4_1$ states that the syntactic correctness is higher using WikiTest than a text editor. It is first interesting to see the difference between the groups, which is displayed in Table B.7. The mean for group A, B, C is 22.4, 21.7 and 23.5, respectively. There is no significant difference between each group (p=0.262). The objective is to investigate the

difference between the text editor and the wiki. The text editor has a mean of 1.836, while the mean of the wiki is 1.963. This is significant with a p-value of 0.0215, as shown in Table B.8. **$H4_0$ is rejected in favor of $H4_1$**.

The fifth hypothesis is regarding the use of backgrounds. $H5_0$ express that using WikiTest or a text editor leads to the same amount of people writing Backgrounds, while $H5_1$ says that more people will write Backgrounds when using WikiTest compared to a text editor. This hypothesis is used to investigate if the wiki can make it easier to use special functionality of the Cucumber syntax, which could manifest itself through more participants writing backgrounds. The ANOVA in Table B.9 shows that 3 subjects in A, 5 in B and 9 in C wrote at least one background, which is significant with p=0.0432. Using a t-test in Table B.10 makes the difference clearer between the text editor and the wiki, with a p-value of 0.009. **Thus, $H5_0$ is rejected in favor of $H5_1$**.

### 11.2.2 Question 2: Understandability

The question of understandability has one hypothesis, which is related to which tool is better in order to understand the domain. $H6_0$ says "understanding the domain is equally easy using either WikiTest or Protégé", while $H6_1$ says "it is easier to understand the domain using WikiTest compared to Protégé". Table B.11 shows the t-test comparing WikiTest and Protégé in terms of understanding the domain. The mean for WikiTest is 3.42 and the mean for Protégé is 3.33. This is not significant with a p-value of 0.3906. **$H6_0$ is not rejected**.

### 11.2.3 Question 3: Usability

Usability is related to three hypotheses. The first is $H7_0$, which says there is no difference in how easy it is to get an overview of the tests when using WikiTest or a text editor. The alternative hypothesis $H7_1$ states that WikiTest makes it easier to get an overview of the tests compared to a text editor. The difference between the groups is given in the ANOVA - Table B.12. The t-test in Table B.13 shows that WikiTest is better with a mean of 3.83, compared to 3.08 of the text editor. This is significant with p=0.01765, so **$H7_0$ is rejected in favour of $H7_1$**.

$H8_0$ states that there is no difference between WikiTest and a text editor in how pleasant it is to write tests, while $H8_1$ states that WikiTest makes it more pleasant to write tests compared to a text editor. The comparison of

the groups is given in Table B.14, while the comparison of the wiki to text editor is given in Table B.15. The satisfication of the text editor is 2.79, while the wiki has a mean of 3.83. This is significant with p=0.00092. **We reject $H8_0$ in favor of $H8_1$**.

$H9_0$ says that there is difference in ease of use between WikiTest and Protégé. The alternative hypothesis, $H9_1$, propose that WikiTest is easier to use than Protégé. The ease of use is shown in Table B.16. Protégé has a mean of 2.75 and WikiTest has a mean of 3.92. This is significant at the 0.00285 level. **Thus, we reject $H9_0$ and use $H9_1$**.

### 11.2.4 Question 4: Efficiency

The fourth and last question is regarding efficiency. There are two hypothesis; one comparing the efficiency of WikiTest to a text editor, and another for the use of a domain ontology. Table B.17 shows that the efficiency per scenario between the three groups is close.

$H10_0$ states that there is no difference in test efficiency between a text editor and WikiTest, while $H10_1$ says there is a difference in test efficiency between a text editor and WikiTest. The results can be found in Table B.18 that displays the test efficiency. There we see that the means are almost similar and the two-tail p-value is 0.9448. The two-tailed valued is used as we are looking both for proof that the text editor is more efficient, and that the wiki is more efficient. **The null hypothesis $H10_0$, that says there is no difference in test efficiency between a text editor and WikiTest, is not rejected**.

$H11_0$ states that using a domain ontology has no effect on test efficiency, while $H11_1$ states that using a domain ontology leads to a difference in test efficiency. As can be seen in Table B.19, the means are almost the same. **With a two-tailed p-value of 0.881, we do not reject $H11_0$**. Using a domain ontology has no effect on test efficiency.

### 11.2.5 Exploring Further

During the execution and evaluation of an experiment, new ideas might come up based on patterns in the data. The weakness of this approach is the chance of a false positive. Analyzing large amount of data will eventually show a significant result, even if it does not exist. The following analyses explore parts of the data collected, which can be used as ideas for new hypotheses.

As the metrics for test case quality are divided in five parts, it is possible to analyze each part separately. The results of analyzing metric 6, feature coverage, is shown in Table B.20 and B.21. There is no significant difference between the three groups, but comparing the text editor and the wiki gives the average scores of 1.79 and 1.88, respectively. This is significant with p=0.02; the feature coverage is higher using the wiki than the text editor.

The scenario completeness metric shows no significant difference between the groups (Table B.22) and between the wiki and text editor (Table B.23). The ambiguity score, on the other hand, shows a difference. Table B.24 shows that the average for group A is 1.82, group B has 1.92 and group C has 1.98. There is a significant difference between the groups with p=0.0254. Looking at the difference in Table B.25, the significance level is even larger when comparing the text editor to the wiki, with a p-value of 0.00325.

The *verifiability* score is not significant between the groups, as shown in Table B.26, with p=0.13. When comparing the text editor to the wiki, a significant difference with p=0.0066 is found, as shown in Table B.27.

Another topic to further investigate is the effect of a domain ontology. When comparing Protégé and WikiTest, the statement "the ontology makes it easier to write tests" has an average score of 3.0 and 3.58, respectively. That means that more participants using WikiTest agreed that the ontology made it easier to create tests (Table B.28). The result is significant with a p-value of 0.0397.

A question was asked to find out if the subjects felt that more steam boiler help was needed. Table B.29 shows hardly any difference between the groups, with p=0.93.

The last topic to cover is the use of Cucumber. The difficulty of using the Cucumber syntax shows no significant difference between the groups, neither between the groups nor between text editor and wiki. There is no difference in how many subjects answer that they would rather write tests in a free format. The numbers can be found in Table B.30, B.31 and B.32. When asked if more help was needed concerning the Cucumber syntax, there is no difference between the groups (Table B.33) or the text editor and wiki (Table B.34).

Table 11.1 summarize the results for the topics which was further investigated. The analysis displays some more information about which metrics

made WikiTest score better. In test quality, it was everything except scenario completeness. Another thing to note is that the subjects using WikiTest felt the ontology made writing tests easier. This was perhaps because the auto-complete functionality in WikiTest made it more explicit how the domain ontology concepts was used, as some participants wrote about in their post-experiment comments.

| Measurement | Type | P-value |
|---|---|---|
| Feature coverage | Groups | 0.2568 |
| Feature coverage | Wiki/Text | 0.0271 |
| Scenario completeness | Groups | 0.6204 |
| Scenario completeness | Wiki/Text | 0.3201 |
| Ambiguity | Groups | 0.0254 |
| Ambiguity | Wiki/Text | 0.0033 |
| Verifiability | Groups | 0.1322 |
| Verifiability | Wiki/Text | 0.0066 |
| Ontology makes writing easier | Wiki/Protégé | 0.0398 |
| Steam boiler help | Groups | 0.9318 |
| Cucumber syntax | Groups | 0.8290 |
| Cucumber syntax | Wiki/Text | 0.3402 |
| Write tests in free text | Groups | 0.6860 |
| Cucumber help | Groups | 0.2889 |
| Cucumber help | Wiki/Text | 0.4211 |

Table 11.1: Summary of extra measurements

## 11.3 Summary of Hypotheses

38 subjects attended the experiment. The data from two subjects were excluded, as one did not take the whole experiment seriously and the other had computer problems. The data collected was found using a questionnaire before the experiment, and one questionnaire after the experiment. The data from these questionnaires were subjective answers, mostly using questions where the answer could be picked from a five point Likert scale. In addition, the tests that were written were handed in and evaluated by the author, grading each feature and scenario based on five criteria. The time used from the pre-experiment questionnaire was delivered to the time the post-experiment questionnaire was used as an indicator of the time used writing features.

The data collected was statistically tested using ANOVA when comparing the three groups, t-test when comparing the wiki to a text editor, and a t-test when comparing the use of a domain ontology. A total of eleven hypotheses were proposed. Five of them could not be rejected, as shown in Table 11.2. The key hypothesis were H1, H2 and H3. There were a difference in test quality when comparing the wiki to a text editor (H1), but no difference was found when comparing the use of a domain ontology (H2). Additionally, no difference in test quality was found when looking at subjects with different levels of experience (H3).

| ID | Hypothesis | P-value |
|---|---|---|
| H2 | Using a domain ontology has no effect on test quality | 0.154 |
| H3 | Experience does not lead to a difference in test quality | 0.117 |
| H6 | Understanding the domain is equally easy using either WikiTest or Protégé | 0.391 |
| H10 | There is no difference in test efficiency between a text editor and WikiTest | 0.945 |
| H11 | Using a domain ontology has no effect on test efficiency | 0.881 |

Table 11.2: $H_0$-hypotheses that could not be rejected

H10 and H11 looked at the difference in test efficiency. It was possible that a side effect of using WikiTest would be that the overall test specification efficiency would decrease. Since neither H10 or H11 could be rejected, no significant difference in test efficiency was found, and we conclude that the side effect did not occur.

The six hypotheses that could be rejected are shown in Table 11.3. Syntactic

correctness (H4) and the use of backgrounds (H5) are “specific” hypotheses, which means that they look at the strengths of the wiki. It was hypothesized that using the syntax correctly was easier when using forms in the wiki, and that the wiki makes it easier to use special constructs of the Cucumber functionality, such as backgrounds. Both these hypotheses gave significant results.

| ID | Hypothesis | P-value |
|---|---|---|
| H1 | Using WikiTest leads to higher test quality | 0.0012 |
| H4 | The syntactic correctness is higher using WikiTest | 0.0215 |
| H5 | More people will write Backgrounds when using WikiTest | 0.0090 |
| H7 | WikiTest makes it easier to get an overview of the tests | 0.0177 |
| H8 | WikiTest makes it more pleasant to write tests | 0.0009 |
| H9 | WikiTest is easier to use than Protégé | 0.0029 |

Table 11.3: Alternative hypotheses accepted

H7, H8 and H9 is related to the user interface of the wiki. One of the core ideas of acceptance testing is to involve the customer. High usability is a necessity in this regard. The hypotheses indicate that using WikiTest makes it easier to get an overview of the tests, it is more pleasant to write tests, and it is easier to use than Protégé.

# CHAPTER 12

# INTERPRETATION

In this chapter, the data calculation and statistical testing from Chapter 11 is discussed. The content of this chapter involves a subjective judgment of the results found using the statistical tests. The answers from the open questions in the post-experiment questionnaire will be used when discussing the data, and the interpretation will be done with threats to validity in mind.

## 12.1 Participant Feedback

To represent the comments given by the participants, an Ishikawa diagram will be displayed for each group. The Ishikawa diagram, also called a fishbone diagram, is used to represent cause and effect relationships for a topic. The Ishikawa diagram could be used to find the root cause of a defect or problem. Here, the diagram will be used to structure the comments from the post-experiment questionnaire. The three questions posed were:

1. What did you feel you could need help with, what was confusing?
2. What extra functionality would you want for writing Cucumber features and scenarios?
3. Additional comments (things not covered by the other questions)?

Only the third question, which asks about additional comments, is *neutral* in the sense that it does not ask for opinions of a certain category. The first and second comments asks about what the subject needed more help with, and what extra functionality was wanted to write Cucumber tests. The main angle of the questions is what can be improved. As an afterthought, it would have been useful to ask both for what worked well and what could need improvement. The feedback from the group using only a text editor is shown in Figure 12.1.

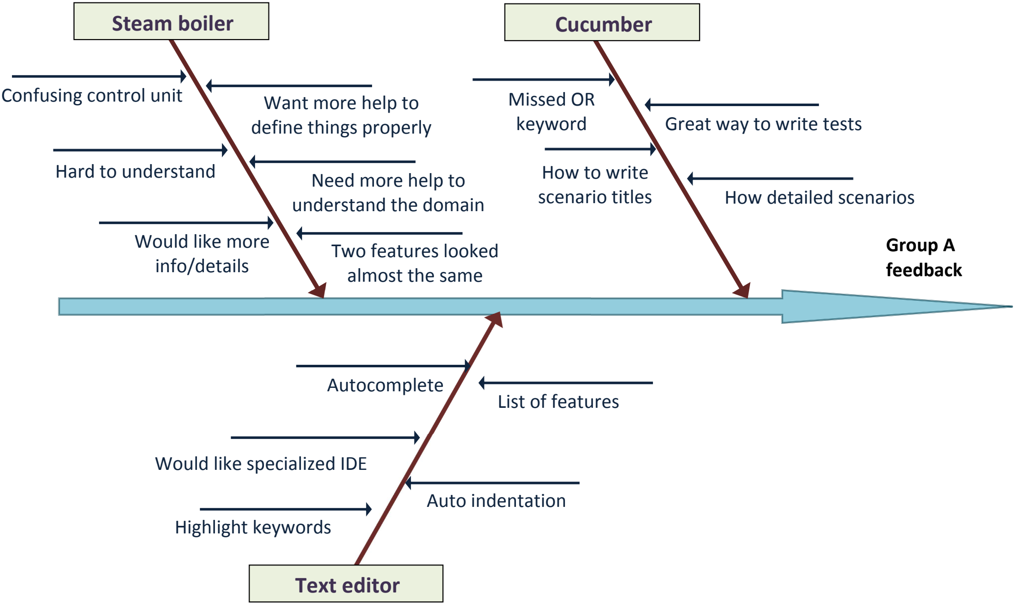


Figure 12.1: Ishikawa diagram for group A

The comments from group A can be divided into three categories: Cucumber, steam boiler and text editor. Several subjects from this group wanted more help about the steam boiler. One person wanted a keyword, “OR”, which do not exist in the Gherkin language used in Cucumber. The addition of an OR construct is an interesting thought, and would make it easier to create more complex tests. The feedback from the group using a text editor with a domain ontology is given in Figure 12.2. Although the comments relate, the main concern here is how Protégé is difficult to use. Some text editor comments are also in common with group A, including a need of syntax highlighting and auto-formatting of text.

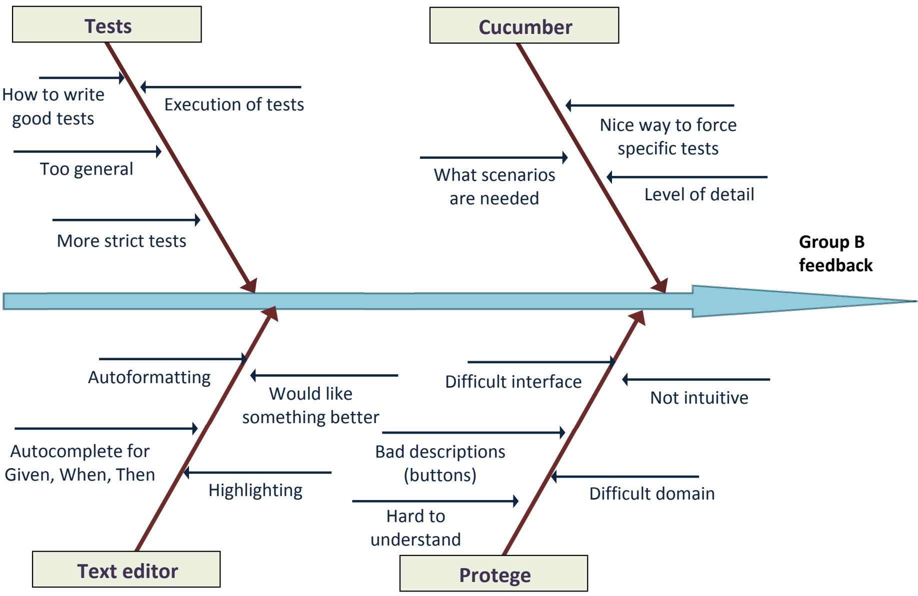


Figure 12.2: Ishikawa diagram for group B

The comments from the group using WikiTest is given in Figure 12.3. The main topic was the auto-complete functionality in the wiki. Some liked it, while others felt it was disturbing. It was suggested that the auto-complete need an option to turn it on and off, which seems like a good idea considering the conflicting feedback. One subject felt more intelligent auto-completion was needed, which was also considered in the requirement specification of the wiki tool.

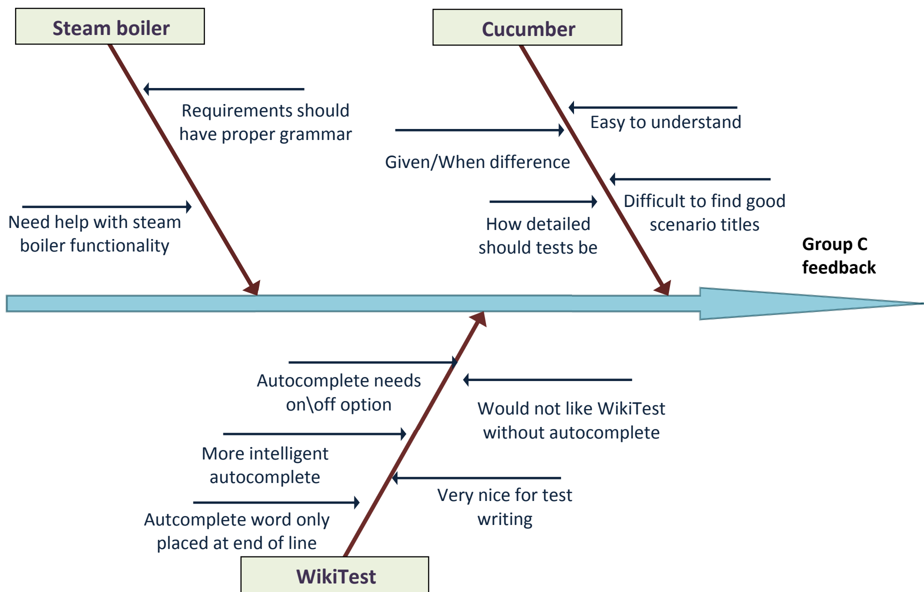


Figure 12.3: Ishikawa diagram for group C

For all the groups, at least one participant wanted more help with the Given/When keywords, which were hard to distinguish. Mixing these two keywords were the most common source of syntactic error when the tests were evaluated. Several subjects commented that it was difficult to find good titles for scenarios and features. This may be because the features often had a single scenario, with a title that was close to the feature title. Perhaps the difficulty of the steam boiler domain, as noted by some subjects, were a factor that made it difficult to find good scenario titles. It was also commented that it was difficult to find a good abstraction level of the Cucumber tests, i.e. how detailed the scenarios should be.

## 12.2 Validity Discussion

In Chapter 9, five threats were regarded as high priority threats. The first was the possibility of low statistical power. Hypothesis 2, the effect of a domain ontology on test quality, had mean score of 9,11 and 9,34, and a p-value of 0,154. It is possible that a significant difference could have been found if the number of subjects was increased. However, if we look more closely at Table B.1, group A and B have almost identical results, so the difference in ontology score comes solely from group C, which used the wiki. Overall, the possibility of low statistical power did not affect the experiment.

Instrumentation was a high threat. In order to run the experiment, 14 separate wikis was set up using separate database tables on the same server. If the server did not manage the load, the experiment would have to be aborted. As it turned out, it took a long time to load the first time the subjects entered, but other than that, the server handled the workload.

There were three high priority *construct validity* threats. The first was the mono-method bias, as the test case quality was graded based on subjective judgment. To reduce it, the features were graded by taking one from each group, before doing another round. This removed some risks, such as grading the first deliveries different from the last ones. Hypothesis guessing was certainly possible, and combined with experimenter expectancies, the results may have been influenced. It is worth to mention the Hawthorne effect [42], which says that the subjects of an experiment may behave differently when they know they are being part of an experiment. When judging the test case quality, it seemed that everyone except one subject had done a good effort, and the Hawthorne effect did not seem to create any artificial differences *between* the groups.

## 12.3 Conclusion

The outcome of the experiment was that six out of the eleven hypotheses could be rejected. The most important finding was the WikiTest lead to a higher test quality. It was easier to use the correct syntax with WikiTest, and the use of backgrounds increased. H7, H8 and H9 were related to usability, and WikiTest was significantly easier to use than Protégé. It was easier to get an overview of the tests, and the participants felt it was more pleasant to write tests using WikiTest.

It was not possible to show that the use of a domain ontology increased the test quality. It is possible that access to a domain ontology is only really helpful if a tool can use it to help the user, instead of the user himself having to navigate around in the ontology in search for the correct relationships which he wants to create tests for.

A large majority felt that Cucumber was easy to use, and disagreed that they would rather write tests in a free format. Considering only one participant had even heard about Cucumber before the experiment, this should be regarded as a confirmation that picking Cucumber as a specification and

execution tool was a good choice.

# Part IV

# Evaluation

# CHAPTER 13

# RESULTS

This thesis is made up of four parts. The first part presented the background theory of software requirements, software testing and possible technology platforms. The second part described the identification of functionality for a new tool and the development of the prototype tool called WikiTest. The third part was an evaluation of the usefulness of this tool in an experiment involving students. This fourth part will give the evaluation of the thesis, starting by answering the research questions posed in Chapter 2.

## 13.1 Approach Selected

Automation in software testing span a wide range of possibilities. Being able to automate repetitive tasks offers a chance of great time and cost savings. But, there is a risk. If the effort needed to create the automation framework is high, the time and cost savings may evaporate. Most model-based test approaches seen in the literature are constrained by the limitations and difficulties of creating a good test model.

The answer to RQ1 - selecting a test approach, is given in Chapter 4.4. The approach selected is one that utilizes a domain ontology to provide assistance for users writing tests. This approach was chosen to limit a problem often seen in automatic test generation projects, where the ideas grow too far away from what is possible to implement given the available resources of the project. The main reason to use an assisted test specification approach is the potential for further use, as a small improvement is more easily incorporated in the test process used by a company in the industry.

## 13.2 Tools and Methods

RQ2 is concerned with the exploration of tools and methods that already exist for the approach selected in the section above. As described in Chapter 4.5, Cucumber was preferred as the test execution tool. This is because Cucumber, with its use of step definitions, offers a relatively free style of test specification without sacrificing the ability to automate the tests.

Chapter 5 describes possible technology platforms for the creation of a user-assisting tool. There are methods in the literature and in the industry that utilize wikis for requirement management. Using a wiki seems like a good choice for test specification as well. An advantage is that the tests can be more closely connected to the requirements when a wiki is used to store the requirements, the domain information and the tests. Confluence and Jira are specialized wikis for software development, and their widespread adoption in the industry attest to the benefits of wikis in such settings.

Semantic MediaWiki was chosen as the wiki engine to use. A wide range of wiki engines exists, but MediaWiki has already proven to be a mature and well documented wiki server which is easy to install. In addition, MediaWiki has good support for semantic information, which is the most interesting functionality from the research perspective of this thesis.

## 13.3 Tool Requirements

Although wikis are already being used as a tool in software development, using a wiki with a domain ontology for test specification is something new. As part of this new approach, it is necessary to elicit possible requirements for such a tool. This functionality elicitation is the focus of RQ3, which is answered in Chapter 6. In addition to collecting ideas from articles, a focus group was used to explore what requirements a tool should have.

The main goals for the tool is to facilitate the use of an ontology in order to give intelligent suggestions for the user when he writes tests. The tool should provide information about the syntax of the tests so that the customer can write syntactically correct tests without knowing all the specifics of the execution tool that is used. Another use of the ontology is “ontology coverage”, which is a test coverage looking at which concepts and properties that have been used in tests.

## 13.4 Empirical Evaluation

The fourth research question, RQ4, looks at the applicability of WikiTest in an empirical setting. An experiment involving 38 students was conducted, where the students wrote features and scenarios for a steam boiler. The students were divided in the following three groups:

| **Group** | **Tool** | **Ontology** |
|---|---|---|
| A | Text editor | No |
| B | Text editor | Yes |
| C | Wiki | Yes |

Table 13.1: Groups in the experiment

The results of the experiment can be found in Chapter 11.3. The conclusions that can be drawn are:

- H1: Using WikiTest leads to higher test quality (p=0.00117)
- H2: Using a domain ontology has no effect on test quality (p=0.154)
- H3: Experience does not lead to a difference in test quality (p=0.117 for programming experience and p=0.447 for test tool experience)
- H4: The syntactic correctness is higher using WikiTest (p=0.0215)
- H5: More people will write Backgrounds when using WikiTest (p=0.009)
- H6: Understanding the domain is equally easy using either WikiTest or Protege (p=0.3906)
- H7: WikiTest makes it easier to get an overview of the tests (p=0.01765)
- H8: WikiTest makes it more pleasant to write tests (p=0.00092)
- H9: WikiTest is easier to use than Protege (p=0.00285)
- H10: There is no difference in test efficiency between a text editor and WikiTest (p=0.9448)
- H11: Using a domain ontology has no effect on test efficiency (p=0.881)

# CHAPTER 14

# DISCUSSION

The objective of this thesis was to look at the possibilities for using boilerplates and a domain ontology for semi-automatic selection of test strategy and generation of test cases. The assignment was related to the work done in the CESAR project. Exploration of test automation, with a focus on MBT tools, has been performed in other research projects, such as AGEDIS [43], D-MINT [44] and MOGENTES [45]. The decision to create a tool that can assist the user, instead of creating tests directly, was partly based on the conclusions of these three projects. As we did not find an efficient way to translate the domain ontology to an MBT model, choosing an MBT approach for acceptance test creation would still require significant manual work.

The advantage of the tool-assisted approach is that it provides a range of possibilities, as the ontology can be used in different ways to help the user. When the user writes a test, the ontology can be used to give suggestions for which domain concepts are valid in the given context. The ontology can be used to reason about test coverage ("ontology coverage"), and can be used to provide a common vocabulary for the customers and the testers. The main disadvantage of the approach is the lack of automation. In the prototype created, the semi-formal structure of the boilerplates is not utilized. The boilerplates are merely included as the requirement text in the feature pages. The WikiTest approach is tightly connected to the Cucumber way of specifying tests. We believe this is a necessary reduction of generalizability in order to be able to implement and automatically execute the tests at a later stage. Another test execution tool could also be used. The main difference would be the layout of the semantic forms that are used to create the tests.

Using a web-based wiki as the technology platform was in our experience a good choice. Most wiki engines provide a number of important functionalities that would otherwise have to be implemented from scratch. This includes a history of changes, previews, discussion, search, version control, access control, import and export. Feedback from the experiment showed that WikiTest was pleasant to use and gave a good overview of the tests. This was important, as usability is an important factor if customers are to use the tool. Choosing MediaWiki as the wiki engine was an obvious choice in the start of the project due to the amount of extensions that existed. In hindsight, using the enterprise-wiki Confluence would have been closer to what is actually used in the industry, but it would require an academic license. ZAgile Wikidsmart is a promising addition to Confluence, as it adds support for semantic information. If Confluence was used, a larger implementation effort would be needed from our side to reach the same level of functionality as Semantic Forms provides in SMW.

The requirements in Chapter 6 were written to provide a foundation for a tool-assisted approach. The focus group was used to discuss the potential requirements for such an approach, with a focus on how the domain ontology could be used. It would be useful to include developers, testers and customers from the industry to elicit a wider range of requirements. Due to the amount of existing functionalities provided by MediaWiki extensions, the implementation became a composition of tweaking extensions in PHP and using wiki syntax to create the wiki pages. This in turn lead to a a group of extensions solving multiple requirements, which made the link from requirements to implementation more difficult to track.

In the experiment, there were two important aspects which could have been better. First, to evaluate the experiment instrumentation on one or two people before the actual experiment would be a good way to improve the experiment instructions and documentation without much effort. The second is to use more than one person when judging the quality of the tests. Even though this risk was reduced by grading the features by five different metrics, the subjective judgment is a threat to the validity of the experiment.

The findings from the experiment indicate that the test quality increases when WikiTest is used, but not when only looking at the effect of the domain ontology. We believe the reason is that the ontology has to be integrated in the tool and not just be source of information on the “sideline”. It is interesting to point out that when asked about how easy the Cucumber style was to use, the group using WikiTest had a higher disparity than the other

groups, as shown in Table 11.13. The participant feedback showed that not everyone felt that the autocomplete functionality of the wiki was helpful. We believe more people would like the autocomplete if only relevant/valid concepts were included, but with a possibility to turn it on or off.

The tool provides a way to enforce syntactic correctness of the tests, as shown by hypothesis H4. The use of special constructs, as seen by H5, increases in the wiki. We find it likely that syntactic correctness would increase if another execution tool was used instead of Cucumber, as the wiki provides a way to validate the input from the user, such as saying that a text field is required. An initial fear was that the time it takes to write tests would increase considerably when using the wiki instead of a text editor. As shown in H10, no significant difference in test efficiency was found. A small difference in efficiency could anyways be acceptable if it leads to an overall increase in test quality.

Experience did not have an effect on test quality. It is likely that the differences in experience were not big enough among the participants, and that a group of experienced testers would have shown a difference. Another explanation may be that since the tests are only specified, and not implemented, the programming experience or test tool experience does not matter.

## 14.1 Further Work

Recommendations for further work will be divided into whether the continued work would be performed in an academic or industrial setting. In an academic setting, the major point of interest is how the most relevant words from the domain ontology can be provided to the user. This is the task of suggesting the most relevant words based on what is already written. A similar task is to find a way to determine when the states are specified “enough”, meaning that each step in the test are valid and contain enough information so that they actually make sense from a domain perspective.

In an industrial setting, a company should pursuit a more direct approach. The implementation could be tailored to whatever enterprise-wiki the company uses, with the goal of using the domain ontology as a natural tool in the development process. The wiki can make use of the ontology more accessible for everyone, and domain information can be stored here. The tests should be specified in the wiki in the syntax of the execution tool of that specific company or project.

Given more time on this thesis, the next goal would be to evaluate the tool in an industrial setting. Before this goal, the changes proposed by the participants of the experiment should be implemented. The top three most important functionality yet to be implemented are:

- More intelligent autocomplete
- Export all feature-pages to feature files
- Hierarchy of features. Provide a possibility to group related features.

# CHAPTER 15

# CONCLUSION

The purpose of this thesis was to explore the combination of boilerplates and a domain ontology for test creation. The approaches identified involved different degrees of automation. The approach selected for further study involved the creation of a tool that could guide or assist users when they specify tests. To do this, Semantic MediaWiki was used to create a tool that we named WikiTest. This tool is based on utilizing the wiki with its underlying domain ontology. Cucumber tests are created using Semantic Forms. The fields of the tests are filled in with domain specific suggestions using AJAX autocomplete. The domain concepts and relationships are easily accessible through the wiki, and provide a common vocabulary for the domain.

A set of possible requirements for a user-assisting tool was identified. One minimal installation with the newest Semantic Forms version was shown, as well as a larger installation using SMWHalo, Semantic Gardening and other extensions provided by the German company Ontoprise. Chapter 7 describes how the requirements were implemented, and what extensions were used. MediaWiki has the advantage of providing a multitude of extensions, which means the implementation details becomes a mixture of extensions, PHP code, wiki parser functions and regular wikitext. The goal was not to create a polished product, but to show how the tool-supported approach could be implemented using already existing features. As the versions of extensions and MediaWiki change, the WikiTest-specific code needs to be adapted as well.

WikiTest was evaluated in an experiment, where students were asked to create tests for a steam boiler domain. The experiences and impressions of the participants were collected both before and after the experiment was conducted. Each student delivered the Cucumber features he had created, which had been written using either a text editor (such as Microsoft Word) or WikiTest. The features were given grades based on their coverage, completeness, correctness, ambiguity and verifiability. These five metrics were combined to make up a total score for the test quality of the features.

The results of the experiment showed that the approach using the wiki gave a significant higher test quality than the text editor. The syntactic correctness was higher, and more participants used at least one Cucumber background. The participants felt it was easier to get an overview of the tests and it was more pleasant to write tests. Compared to the ontology editor Protégé, a significant amount felt WikiTest was easier to use.

The use of a domain ontology compared to not having an ontology did not show the same significant difference. It is likely that the domain ontology needs to be utilized in some concrete way in order for it to have an effect on test case quality. Just being able to browse the ontology does not seem to give the same results as when the ontology is directly connected to the test specification process. The integration of the ontology in a wiki such as Semantic MediaWiki offers many possibilities, and we believe this approach could be utilized in software development wikis such as Confluence. Further work would involve implementing more functionality and perform an empirical evaluation of WikiTest in an industrial setting.

The four research questions are summarized here.

- **Approach for further work (RQ1):** As described in Chapter 4.4, we decided against an MBT approach and instead created a tool that would provide user-assistance based on the ontology in the test specification process.
- **Tools and methods (RQ2):** Cucumber, offering a relatively free style of test specification without sacrificing the ability to automate the tests, was used as test execution tool (Chapter 4.5). Semantic MediaWiki was chosen as the wiki engine to use, as it has a large set of semantic extensions (Chapter 5).
- **Requirements for a new tool (RQ3):** The tool should give suggestions when the user writes tests, provide information about the test syntax and provide an "ontology coverage" (Chapter 6).
- **Empirical evaluation of the tool (RQ4):** WikiTest was found to give higher test case quality. It leads to more syntactic correct tests, gives a better overview of the tests and is more pleasant to use. The use of a domain ontology does not lead to higher test quality when used with the ontology tool Protege.

# REFERENCES


[1] W. H. Jessop, J. R. Kane, S. Roy, and J. M. Scanlon. ATLAS-An Automated Software Testing System. In *Proceedings of the 2nd international conference on Software engineering*, 1976.

[2] C. V. Ramamoorthy, Siu bun F. Ho, and W. T. Chen. On the Automated Generation of Program Test Data. *IEEE Transactions On Software Engineering*, 4:293–300, 1976.

[3] CESAR. Refined pilot application description and success evaluation criteria for Automation and Railway domain. Unpublished, August 2010.

[4] Elizabeth Hull, Ken Jackson, and Jeremy Dick. *Requirements Engineering*. Springer, 2005.

[5] Jeremy Dick. Requirements boilerplates. Accessed: 2010-11-15, http://freespace.virgin.net/gbjedi/books/re/boilerplates.htm.

[6] Tor Stålhane, Inah Omoronyia, and Frank Reichenbach. Ontology-guided requirements and safety analysis. *Proceedings of the 6th International Conference on Safety of Industrial Automated Systems*, 2010.

[7] Olav Undheim. Specialization project: Semi automatic test case generation. Norwegian University of Science and Technology, 2010.

[8] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Bjöorn Regnell, and Anders Wesslén. *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, 2000.

[9] Linda Westfall. Software requirements engineering: What, why, who, when and how, 2005.

[10] Barry Boehm and Victor R. Basili. Software Defect Reduction Top 10 List. *Computer*, v. 34:135–137, 2001.

[11] Alan M. Davis. *Software Requirements: Objects, Functions, and States, Second Edition.* Prentice Hall, 1993.

[12] W. T. Tsai, R. Paul, and L. Yu. Rapid Pattern-Oriented Scenario-Based Testing for Embedded Systems. *Software Evolution with UML and XML*, pages 222–262, 2005.

[13] Wei-Tek Tsai, Lian Yu, Feng Zhu, and Ray Paul. Rapid Embedded System Testing Using Verification Patterns. *IEEE Softw.*, 22:68–75, 2005.

[14] Sascha Konrad and Betty H.C. Cheng. Realtime specification patterns. In *ICSE 2005: Proc. 27th Int. Conf. Softw. Eng.*, 2005.

[15] Ernst Sikora, Bastian Tenbergen, and Klaus Pohl. Requirements engineering for embedded systems: An investigation of industry needs. *Requirements Engineering: Foundation for Software Quality*, pages 151–165, 2011.

[16] T. R. Gruber. Toward principles for the design of ontologies used for knowledge sharing. *International Journal of Human-Computer Studies*, 43:907–928, 1995.

[17] Inah Omoronyia. GNLQ. http://idi.ntnu.no/ inah1/cesar/, Accessed: 2010-11-12.

[18] Kshirasagar Naik and Priyadarshi Tripathy. *Software Testing and Quality Assurance: Theory and Practice.* WILEY, John Wiley & Sons, Inc., 2008.

[19] Structural Coverage Metrics. http://www.math.unipd.it/ tullio/is-1/dispense_2003/software_testing_metrics. *Information Processing Limited*, 1997.

[20] Matt Archer's Blog. http://mattarcherblog.wordpress.com/2008/12/08/the-testing-v-model-catch-22/, Accessed: 2010-10-11.

[21] Bart Broekman and Edwin Notenboom. *Testing Embedded Software.* Addison-Wesley, 2002.

[22] David Chelimsky, Dave Astels, Zach Dennis, Aslak Hellesøy, Bryan Helmkamp, and Dan North. *The RSpec Book Behaviour Driven Development with RSpec, Cucumber, and Friends.* The Pragmatic Programmers LLC., 2010.

[23] David de Florinier and Gojko Adzic. *The Secret Ninja Cucumber Scrolls: Strictly Confidential.* Neuri Limited, 2010.

[24] Jeff McAffer and Jean-Michel Lemieux. *Eclipse Rich Client Platform: Designing, Coding, and Packaging Java Applications.* Addison-Wesley Professional, 2005.

[25] Bo Leuf and Ward Cunningham. *The Wiki Way: Quick Collaboration on the Web.* Addison-Wesley Professiona, 2001.

[26] Jeff Orlof and Mizanur Rahman. *MediaWiki 1.1.* Packt Publishing, 2010.

[27] Thomas Meilender, Nicolas Jay, Jean Lieber, and Fabien Palomares. Semantic wiki engines: a state of the art. 2010.

[28] Yaron Koren. Wikimedia data summit. Sebastopol, CA, 2011.

[29] B. Decker, E. Ras, J. Rech, P. Jaubert, and M. Rieth. Wiki-based stakeholder participation in requirements engineering. *IEEE Software*, 24:28–35, 2007.

[30] Shailey Minocha and Peter G. Thomas. Collaborative learning in a wiki environment: Experiences from a software engineering course. *New Review of Hypermedia and Multimedia*, 13:187–209, 2007.

[31] Filipe Figueiredo Correia. Extending and integrating wikis to improve software documentation. In *Wikis for Software Engineering*, 2008.

[32] Tim Romberg. Wiquila – a wiki rich client that mixes well with other sources of software project information. In *Wikis4SE: wikis for software engineering*, 2008.

[33] Eric Knauss, Olesia Brill, Ingo Kitzmann, and Thomas Flohr. SmartWiki: Support for High-Quality Requirements Engineering in a Collaborative Setting. *Wikis for Software Engineering*, pages 25–35, 2009.

[34] Semantic MediaWiki. Ontology import. Accessed: 30.03.2011, http://semantic-mediawiki.org/wiki/Help:Ontology_import.

[35] Victor R. Basili and David M. Weiss. A methodology for collecting valid software engineering data. *IEEE Trans. Software Eng.*, 10:728–738, 1984.

[36] Rini van Solingen and Egon Berghout. *Goal/Question/Metric Method*. Mcgraw Hill Higher Education, 1999.

[37] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. The goal question metric approach. In *Encyclopedia of Software Engineering*. Wiley, 1994.

[38] Birger Madsen. *Statistics for Non-Statisticians*. Springer-Verlag Berlin Heidelberg, 2011.

[39] Andrew Rutherford. *Introducing Anova and Ancova : a GLM approach*. London : SAGE, 2000.

[40] Norman Blaikie. *Analysing Quantitative Data*. Sage Publications, 2003.

[41] John Tukey. *The Collected Works of John W. Tukey, Voll III: Philosophy and Principles of Data Analysis*. Wadsworth & Brooks/Cole, 1985.

[42] Ritch Macefield. Usability studies and the hawthorne effect. *Journal of usability studies*, 2:145–154, 2007.

[43] Alan Hartman. Agedis final project report, February 2004.

[44] D-MINT. Common Approach to Architecture-Driven Testing. *whitepaper*.

[45] MOGENTES Consortium. State of the art survey - part a: Model-based test case generation, 2008.

# APPENDIX A

# WIKI CODE

Listing A.1: Form:Feature

```
<noinclude>
This is the "Feature" form.
To create a page with this form, enter the page name below;
if a page with that name already exists, you will be sent to a form to
edit that page.


{{#forminput:form=Feature|size=50|button text=Create or edit a feature|
autocomplete on category=Feature}}

</noinclude><includeonly>
<div id="wikiPreview" style="display: none; padding-bottom: 25px;
margin-bottom: 25px; border-bottom: 1px solid #AAAAAA;"></div>
{{{for template|Feature}}}
{| class="formtable"
! Requirement: {{#info:Write the requirement for this feature.
The requirement can be written as a user story, boilerplate or as free text.}}
|-
| {{{field|Requirement|size=110|mandatory|input type=text with autocomplete|
values from property=Label|list|delimiter= }}}
|}
{{{end template}}}

{{{for template|Background|label=Background (shared steps)}}}
The use of a background is optional, and is only needed
if you have steps that are common for all scenarios.
{| class="formtable"
! Given:
| {{{field|Given|input type=textarea with autocomplete|
values from property=Given|list|delimiter= |rows=2}}}
|}
{{{end template}}}

{{{for template|Scenario|multiple|label=Scenarios (add as many as wanted)|
add button text=Add scenario}}}
```

```
{| class="formtable"
! Title:
| {{{field|Title|size=105|mandatory}}}
|-
! Given:
| {{{field|Given|input type=textarea with autocomplete|
values from property=Given|list|delimiter= |rows=2|mandatory}}}
|-
! When:
| {{{field|When|input type=textarea with autocomplete|
values from property=When|list|delimiter= |rows=2|mandatory}}}
|-
! Then:
| {{{field|Then|input type=textarea with autocomplete|
values from property=Then|list|delimiter= |rows=2|mandatory}}}
|}
{{{end template}}}


{{{standard input|save}}} {{{standard input|cancel}}}
</includeonly>
```

Listing A.2: Template:Feature

```
<noinclude>
This is the "Feature" template.
It should be called in the following format:
<pre>
{{Feature
|Requirement=
}}
</pre>
Edit the page to see the template text.
</noinclude><includeonly>
{|
| {{{Requirement|}}}
|}

[[Category:Feature]]
</includeonly>
```

Listing A.3: Template:Background

```
<noinclude>
This is the "Background" template.
It should be called in the following format:
<pre>
{{Background
|Given=
}}
</pre>
Edit the page to see the template text.
</noinclude><includeonly>
{{#if:{{{Given|}}} |
<table border="0">
<tr>
<td style="color:blue">'''Background:'''</td>
```

```
</tr>
<tr>
<td style="padding-left:15px">Given {{#arraymap:
  {{{Given|}}}|\n|x|[[Given::x]]|<br/>And }}</td>
</tr>
</table>
| }}
[[Category:Background]]
</includeonly>
```

Listing A.4: Template:Scenario

```
<noinclude>
This is the "Scenario" template.
It should be called in the following format:
<pre>
{{Scenario
|Title=
|Given=
|When=
|Then=
}}
</pre>
Edit the page to see the template text.
</noinclude><includeonly>
{|
|style="color:blue" |'''Scenario: {{{Title|}}}'''
|-
| style="padding-left:15px" |Given {{#arraymap:{{{Given|}}}|
  \n|x|[[Given::x]]|\n\n  And\n}}
|-
| style="padding-left:15px" |When {{#arraymap:{{{When|}}}|
  \n|x|[[When::x]]|\n\n  And\n}}
|-
| style="padding-left:15px" |Then {{#arraymap:{{{Then|}}}|
  \n|x|[[Then::x]]|\n\n  And\n}}
|}
[[Category:Scenario]]
</includeonly>
```

# APPENDIX B

## EXPERIMENT DATA

In the descriptive statistics that follows, the letter A, B or C is used to denote the different groups.

- A: The group using a text editor but no ontology
- B: Text editor and domain ontology (Protégé)
- C: WikiTest (which includes the domain ontology)

After removing the data of two subjects due to data validation, each group had exactly 12 subjects each. Out of the total 36 participants, three did not answer the pre-experiment questionnaire. However, everyone delivered the main assignment and the post-experiment questionnaire.

| Anova: Single Factor | | | | | | |
|---|---|---|---|---|---|---|
| *Groups* | *Count* | *Sum* | *Average* | *Variance* | | |
| A | 12 | 109.3393 | 9.1116 | 0.4222 | | |
| B | 12 | 109.1167 | 9.0931 | 0.4298 | | |
| C | 12 | 115.0286 | 9.5857 | 0.0529 | | |
| | | | | | | |
| *Variation Source* | *SS* | *df* | *MS* | *F* | *P-value* | *F crit* |
| Between Groups | 1.8713 | 2 | 0.9357 | 3.1020 | **0.0583** | 3.2849 |
| Within Groups | 9.9540 | 33 | 0.3016 | | | |
| Total | 11.8253 | 35 | | | | |

Table B.1: Average points per feature between the three groups

| t-Test: Two-Sample Assuming Unequal Variances | | |
|---|---|---|
| | *A+B* | *C* |
| Mean | 9.102331349 | 9.585714286 |
| Variance | 0.407566607 | 0.052908163 |
| Observations | 24 | 12 |
| Hypothesized Mean Difference | 0 | |
| df | 32 | |
| t Stat | -3.305036216 | |
| P(T<=t) one-tail | **0.001173875** | |
| t Critical one-tail | 1.693888748 | |
| P(T<=t) two-tail | 0.002347749 | |
| t Critical two-tail | 2.036933343 | |

Table B.2: Average points per feature between text editor and wiki

| t-Test: Two-Sample Assuming Unequal Variances | | |
|---|---|---|
| | *A* | *B+C* |
| Mean | 9.111607143 | 9.339384921 |
| Variance | 0.422201125 | 0.29417479 |
| Observations | 12 | 24 |
| Hypothesized Mean Difference | 0 | |
| df | 19 | |
| t Stat | -1.045768995 | |
| P(T<=t) one-tail | **0.154395245** | |
| t Critical one-tail | 1.729132812 | |
| P(T<=t) two-tail | 0.30879049 | |
| t Critical two-tail | 2.093024054 | |

Table B.3: Average points per feature between no ontology and ontology

| t-Test: Two-Sample Assuming Unequal Variances | | |
|---|---|---|
| Programming experience | | |
| | *School only* | *More than school* |
| Mean | 9.124107143 | 9.386678005 |
| Variance | 0.437623203 | 0.17654799 |
| Observations | 12 | 21 |
| Hypothesized Mean Difference | 0 | |
| df | 16 | |
| t Stat | -1.239484688 | |
| P(T<=t) one-tail | **0.116519711** | |
| t Critical one-tail | 1.745883676 | |
| P(T<=t) two-tail | 0.233039422 | |
| t Critical two-tail | 2.119905299 | |

Table B.4: Points grouped by experience

| t-Test: Two-Sample Assuming Unequal Variances | | |
|---|---|---|
| Experience: Written an automatic test before? | | |
| | *Yes* | *No* |
| Mean | 9.305494505 | 9.281904762 |
| Variance | 0.163398417 | 0.363480368 |
| Observations | 13 | 20 |
| Hypothesized Mean Difference | 0 | |
| df | 31 | |
| t Stat | 0.134539311 | |
| P(T<=t) one-tail | **0.446922926** | |
| t Critical one-tail | 1.695518783 | |
| P(T<=t) two-tail | 0.893845853 | |
| t Critical two-tail | 2.039513446 | |

Table B.5: Points grouped by experience

| t-Test: Two-Sample Assuming Unequal Variances | | |
|---|---|---|
| Experience with the test tool Cucumber | | |
| | *Heard about it* | *None* |
| Mean | 9.117857143 | 9.322151361 |
| Variance | 0.821683673 | 0.200277603 |
| Observations | 5 | 28 |
| Hypothesized Mean Difference | 0 | |
| df | 4 | |
| t Stat | -0.493329743 | |
| P(T<=t) one-tail | **0.323818757** | |
| t Critical one-tail | 2.131846786 | |
| P(T<=t) two-tail | 0.647637514 | |
| t Critical two-tail | 2.776445105 | |

Table B.6: Points grouped by experience

| Anova: Single Factor | | | | | | |
|---|---|---|---|---|---|---|
| *Groups* | *Count* | *Sum* | *Average* | *Variance* | | |
| A | 12 | 22.39286 | 1.866071 | 0.072791 | | |
| B | 12 | 21.66667 | 1.805556 | 0.083754 | | |
| C | 12 | 23.55 | 1.9625 | 0.004943 | | |
| | | | | | | |
| *Variation Source* | *SS* | *df* | *MS* | *F* | *P-value* | *F crit* |
| Between Groups | 0.150369 | 2 | 0.075184 | 1.396714 | **0.261653** | 3.284918 |
| Within Groups | 1.776373 | 33 | 0.053829 | | | |
| Total | 1.926742 | 35 | | | | |

Table B.7: Syntactic correctness between the three groups

| t-Test: Two-Sample Assuming Unequal Variances | | |
|---|---|---|
| | *A+B* | *C* |
| Mean | 1.835813 | 1.9625 |
| Variance | 0.075825 | 0.004943 |
| Observations | 24 | 12 |
| Hypothesized Mean Difference | 0 | |
| df | 28 | |
| t Stat | -2.11991 | |
| P(T<=t) one-tail | **0.021505** | |
| t Critical one-tail | 1.701131 | |
| P(T<=t) two-tail | 0.04301 | |
| t Critical two-tail | 2.048407 | |

Table B.8: Syntactic correctness between text editor and wiki

| Anova: Single Factor | | | | | | |
|---|---|---|---|---|---|---|
| *Groups* | *Count* | *Sum* | *Average* | *Variance* | | |
| A | 12 | 3 | 0.25 | 0.2045 | | |
| B | 12 | 5 | 0.4167 | 0.2652 | | |
| C | 12 | 9 | 0.75 | 0.2045 | | |
| | | | | | | |
| *Variation Source* | *SS* | *df* | *MS* | *F* | *P-value* | *F crit* |
| Between Groups | 1.5556 | 2 | 0.7778 | 3.4607 | **0.0432** | 3.2849 |
| Within Groups | 7.4167 | 33 | 0.2247 | | | |
| Total | 8.9722 | 35 | | | | |

Table B.9: Number of subjects who used at least one Cucumber background

| t-Test: Two-Sample Assuming Unequal Variances | | |
|---|---|---|
| | *Text editor* | *Wiki* |
| Mean | 0.333333333 | 0.75 |
| Variance | 0.231884058 | 0.204545455 |
| Observations | 24 | 12 |
| Hypothesized Mean Difference | 0 | |
| df | 23 | |
| t Stat | -2.549610534 | |
| P(T<=t) one-tail | **0.008957149** | |
| t Critical one-tail | 1.713871528 | |
| P(T<=t) two-tail | 0.017914297 | |
| t Critical two-tail | 2.06865761 | |

Table B.10: Number of subjects who used at least one Cucumber background

| t-Test: Two-Sample Assuming Unequal Variances | | |
|---|---|---|
| | *Protégé* | *Wiki* |
| Mean | 3.416666667 | 3.333333333 |
| Variance | 0.628787879 | 0.424242424 |
| Observations | 12 | 12 |
| Hypothesized Mean Difference | 0 | |
| df | 21 | |
| t Stat | 0.281312443 | |
| P(T<=t) one-tail | **0.390612006** | |
| t Critical one-tail | 1.720742903 | |
| P(T<=t) two-tail | 0.781224012 | |
| t Critical two-tail | 2.079613845 | |

Table B.11: Ontology makes the domain easier to understand

| Anova: Single Factor | | | | | | |
|---|---|---|---|---|---|---|
| *Groups* | *Count* | *Sum* | *Average* | *Variance* | | |
| A | 12 | 40 | 3.3333 | 1.6970 | | |
| B | 12 | 34 | 2.8333 | 1.0606 | | |
| C | 12 | 46 | 3.8333 | 0.6970 | | |
| | | | | | | |
| *Variation Source* | *SS* | *df* | *MS* | *F* | *P-value* | *F crit* |
| Between Groups | 6 | 2 | 3 | 2.6053 | **0.0890** | 3.2849 |
| Within Groups | 38 | 33 | 1.1515 | | | |
| Total | 44 | 35 | | | | |

Table B.12: Overview of tests between the three groups

| t-Test: Two-Sample Assuming Unequal Variances | | |
|---|---|---|
| | *Text editor* | *Wiki* |
| Mean | 3.083333333 | 3.833333333 |
| Variance | 1.384057971 | 0.696969697 |
| Observations | 24 | 12 |
| Hypothesized Mean Difference | 0 | |
| df | 30 | |
| t Stat | -2.204453638 | |
| P(T<=t) one-tail | **0.017651524** | |
| t Critical one-tail | 1.697260887 | |
| P(T<=t) two-tail | 0.035303048 | |
| t Critical two-tail | 2.042272456 | |

Table B.13: Overview of tests between text editor and wiki

| Anova: Single Factor | | | | | | |
|---|---|---|---|---|---|---|
| *Groups* | *Count* | *Sum* | *Average* | *Variance* | | |
| A | 12 | 39 | 3.2500 | 0.9318 | | |
| B | 12 | 28 | 2.3333 | 1.1515 | | |
| C | 12 | 46 | 3.8333 | 0.5152 | | |
| *Variation Source* | *SS* | *df* | *MS* | *F* | *P-value* | *F crit* |
| Between Groups | 13.7222 | 2 | 6.8611 | 7.9213 | **0.0015** | 3.2849 |
| Within Groups | 28.5833 | 33 | 0.8662 | | | |
| Total | 42.3056 | 35 | | | | |

Table B.14: User interface satisfaction between the three groups

| t-Test: Two-Sample Assuming Unequal Variances | | |
|---|---|---|
| | *Text editor* | *Wiki* |
| Mean | 2.791666667 | 3.833333333 |
| Variance | 1.21557971 | 0.515151515 |
| Observations | 24 | 12 |
| Hypothesized Mean Difference | 0 | |
| df | 31 | |
| t Stat | -3.405186077 | |
| P(T<=t) one-tail | **0.000922808** | |
| t Critical one-tail | 1.695518783 | |
| P(T<=t) two-tail | 0.001845616 | |
| t Critical two-tail | 2.039513446 | |

Table B.15: User interface satisfaction between text editor and wiki

| t-Test: Two-Sample Assuming Unequal Variances | | |
|---|---|---|
| | *Protégé* | *Wiki* |
| Mean | 2.75 | 3.916666667 |
| Variance | 0.931818182 | 0.810606061 |
| Observations | 12 | 12 |
| Hypothesized Mean Difference | 0 | |
| df | 22 | |
| t Stat | -3.061684674 | |
| P(T<=t) one-tail | **0.002856957** | |
| t Critical one-tail | 1.717144374 | |
| P(T<=t) two-tail | 0.005713914 | |
| t Critical two-tail | 2.073873068 | |

Table B.16: Ease of use

| Anova: Single Factor | | | | | | |
|---|---|---|---|---|---|---|
| *Groups* | *Count* | *Sum* | *Average* | *Variance* | | |
| A | 12 | 0.0637 | 0.0053 | 4.28E-06 | | |
| B | 10 | 0.0543 | 0.0054 | 5.55E-06 | | |
| C | 11 | 0.0596 | 0.0054 | 4.68E-06 | | |
| | | | | | | |
| *Variation Source* | *SS* | *df* | *MS* | *F* | *P-value* | *F crit* |
| Between Groups | 1.04E-07 | 2 | 5.18E-08 | 0.01081 | **0.9893** | 3.316 |
| Within Groups | 0.000144 | 30 | 4.79E-06 | | | |
| Total | 0.000144 | 32 | | | | |

Table B.17: Efficiency per scenario between the three groups

| t-Test: Two-Sample Assuming Unequal Variances | | |
|---|---|---|
| | *A+B* | *C* |
| Mean | 0.005361988 | 0.005417825 |
| Variance | 4.62234E-06 | 4.6757E-06 |
| Observations | 22 | 11 |
| Hypothesized Mean Difference | 0 | |
| df | 20 | |
| t Stat | -0.070061776 | |
| P(T<=t) one-tail | 0.472420152 | |
| t Critical one-tail | 1.724718243 | |
| P(T<=t) two-tail | **0.944840304** | |
| t Critical two-tail | 2.085963447 | |

Table B.18: Efficiency per scenario between text editor and wiki

| t-Test: Two-Sample Assuming Unequal Variances | | |
|---|---|---|
| | *A* | *B+C* |
| Mean | 0.005307 | 0.005423 |
| Variance | 4.28E-06 | 4.83E-06 |
| Observations | 12 | 21 |
| Hypothesized Mean Difference | 0 | |
| df | 24 | |
| t Stat | -0.15165 | |
| P(T<=t) one-tail | 0.440367 | |
| t Critical one-tail | 1.710882 | |
| P(T<=t) two-tail | **0.880733** | |
| t Critical two-tail | 2.063899 | |

Table B.19: Efficiency per scenario between no ontology and ontology

| Anova: Single Factor | | | | | | |
|---|---|---|---|---|---|---|
| *Groups* | *Count* | *Sum* | *Average* | *Variance* | | |
| A | 12 | 21.2143 | 1.7679 | 0.0503 | | |
| B | 12 | 21.6333 | 1.8028 | 0.0269 | | |
| C | 12 | 22.5821 | 1.8818 | 0.0095 | | |
| *Variation Source* | *SS* | *df* | *MS* | *F* | *P-value* | *F crit* |
| Between Groups | 0.0819 | 2 | 0.0409 | 1.4169 | **0.2568** | 3.2849 |
| Within Groups | 0.9532 | 33 | 0.0289 | | | |
| Total | 1.0351 | 35 | | | | |

Table B.20: Feature completion score between the three groups

| t-Test: Two-Sample Assuming Unequal Variances | | |
|---|---|---|
| | *Text editor* | *Wiki* |
| Mean | 1.78531746 | 1.881845238 |
| Variance | 0.037208009 | 0.009523326 |
| Observations | 24 | 12 |
| Hypothesized Mean Difference | 0 | |
| df | 34 | |
| t Stat | -1.993786591 | |
| P(T<=t) one-tail | **0.027123264** | |
| t Critical one-tail | 1.690924255 | |
| P(T<=t) two-tail | 0.054246529 | |
| t Critical two-tail | 2.032244509 | |

Table B.21: Feature completion score between text editor and wiki

| Anova: Single Factor | | | | | | |
|---|---|---|---|---|---|---|
| *Groups* | *Count* | *Sum* | *Average* | *Variance* | | |
| A | 12 | 20.0603 | 1.6717 | 0.0577 | | |
| B | 12 | 18.9359 | 1.5780 | 0.0727 | | |
| C | 12 | 18.9237 | 1.5770 | 0.0896 | | |
| | | | | | | |
| *Variation Source* | *SS* | *df* | *MS* | *F* | *P-value* | *F crit* |
| Between Groups | 0.0710 | 2 | 0.0355 | 0.4844 | **0.6204** | 3.2849 |
| Within Groups | 2.4189 | 33 | 0.0733 | | | |
| Total | 2.4900 | 35 | | | | |

Table B.22: Scenario completeness score between the three groups

| t-Test: Two-Sample Assuming Unequal Variances | | |
|---|---|---|
| | *Text editor* | *Wiki* |
| Mean | 1.624840669 | 1.576978114 |
| Variance | 0.064630622 | 0.08955678 |
| Observations | 24 | 12 |
| Hypothesized Mean Difference | 0 | |
| df | 19 | |
| t Stat | 0.474935203 | |
| P(T<=t) one-tail | **0.320123407** | |
| t Critical one-tail | 1.729132812 | |
| P(T<=t) two-tail | 0.640246813 | |
| t Critical two-tail | 2.093024054 | |

Table B.23: Scenario completeness score between text editor and wiki

| Anova: Single Factor | | | | | | |
|---|---|---|---|---|---|---|
| *Groups* | *Count* | *Sum* | *Average* | *Variance* | | |
| A | 12 | 21.8631 | 1.8219 | 0.0306 | | |
| B | 12 | 23.0500 | 1.9208 | 0.0224 | | |
| C | 12 | 23.7500 | 1.9792 | 0.0024 | | |
| *Variation Source* | *SS* | *df* | *MS* | *F* | *P-value* | *F crit* |
| Between Groups | 0.1516 | 2 | 0.0758 | 4.1139 | **0.0254** | 3.2849 |
| Within Groups | 0.6082 | 33 | 0.0184 | | | |
| Total | 0.7599 | 35 | | | | |

Table B.24: Ambiguity score between the three groups

| t-Test: Two-Sample Assuming Unequal Variances | | |
|---|---|---|
| | *Text editor* | *Wiki* |
| Mean | 1.871378968 | 1.979166667 |
| Variance | 0.027863857 | 0.002367424 |
| Observations | 24 | 12 |
| Hypothesized Mean Difference | 0 | |
| df | 30 | |
| t Stat | -2.924655007 | |
| P(T<=t) one-tail | **0.003254272** | |
| t Critical one-tail | 1.697260887 | |
| P(T<=t) two-tail | 0.006508543 | |
| t Critical two-tail | 2.042272456 | |

Table B.25: Ambiguity score between text editor and wiki

| Anova: Single Factor | | | | | | |
|---|---|---|---|---|---|---|
| *Groups* | *Count* | *Sum* | *Average* | *Variance* | | |
| A | 12 | 21.5476 | 1.7956 | 0.0212 | | |
| B | 12 | 21.3500 | 1.7792 | 0.0545 | | |
| C | 12 | 22.9071 | 1.9089 | 0.0078 | | |
| | | | | | | |
| *Variation Source* | *SS* | *df* | *MS* | *F* | *P-value* | *F crit* |
| Between Groups | 0.1198 | 2 | 0.0599 | 2.1524 | **0.1322** | 3.2849 |
| Within Groups | 0.9182 | 33 | 0.0278 | | | |
| Total | 1.0380 | 35 | | | | |

Table B.26: Verifiability score between the three groups

| t-Test: Two-Sample Assuming Unequal Variances | | |
|---|---|---|
| | *Text editor* | *Wiki* |
| Mean | 1.787400794 | 1.908928571 |
| Variance | 0.036258863 | 0.007807282 |
| Observations | 24 | 12 |
| Hypothesized Mean Difference | 0 | |
| df | 34 | |
| t Stat | -2.614018807 | |
| P(T<=t) one-tail | **0.006619037** | |
| t Critical one-tail | 1.690924255 | |
| P(T<=t) two-tail | 0.013238074 | |
| t Critical two-tail | 2.032244509 | |

Table B.27: Verifiability score between text editor and wiki

| t-Test: Two-Sample Assuming Unequal Variances | | |
|---|---|---|
| | *Protégé* | *Wiki* |
| Mean | 3 | 3.583333333 |
| Variance | 0.909090909 | 0.265151515 |
| Observations | 12 | 12 |
| Hypothesized Mean Difference | 0 | |
| df | 17 | |
| t Stat | -1.864783997 | |
| P(T<=t) one-tail | **0.039785258** | |
| t Critical one-tail | 1.739606726 | |
| P(T<=t) two-tail | 0.079570517 | |
| t Critical two-tail | 2.109815578 | |

Table B.28: Ontology makes it easier to write test

| Anova: Single Factor | | | | | | |
|---|---|---|---|---|---|---|
| *Groups* | *Count* | *Sum* | *Average* | *Variance* | | |
| A | 12 | 37 | 3.0833 | 1.1742 | | |
| B | 12 | 39 | 3.2500 | 1.4773 | | |
| C | 12 | 38 | 3.1667 | 0.8788 | | |
| *Variation Source* | *SS* | *df* | *MS* | *F* | *P-value* | *F crit* |
| Between Groups | 0.1667 | 2 | 0.0833 | 0.0708 | **0.9318** | 3.2849 |
| Within Groups | 38.8333 | 33 | 1.1768 | | | |
| Total | 39 | 35 | | | | |

Table B.29: Steam boiler help

| Anova: Single Factor | | | | | | |
|---|---|---|---|---|---|---|
| *Groups* | *Count* | *Sum* | *Average* | *Variance* | | |
| A | 12 | 45 | 3.7500 | 0.2045 | | |
| B | 12 | 46 | 3.8333 | 0.1515 | | |
| C | 12 | 44 | 3.6667 | 0.9697 | | |
| *Variation Source* | *SS* | *df* | *MS* | *F* | *P-value* | *F crit* |
| Between Groups | 0.1667 | 2 | 0.0833 | 0.1886 | **0.8290** | 3.2849 |
| Within Groups | 14.5833 | 33 | 0.4419 | | | |
| Total | 14.7500 | 35 | | | | |

Table B.30: Difficulty of using the Cucumber syntax

| t-Test: Two-Sample Assuming Unequal Variances | | |
|---|---|---|
| | *Text editor* | *Wiki* |
| Mean | 3.791666667 | 3.666666667 |
| Variance | 0.172101449 | 0.96969697 |
| Observations | 24 | 12 |
| Hypothesized Mean Difference | 0 | |
| df | 13 | |
| t Stat | 0.421425262 | |
| P(T<=t) one-tail | **0.340164637** | |
| t Critical one-tail | 1.770933396 | |
| P(T<=t) two-tail | 0.680329274 | |
| t Critical two-tail | 2.160368656 | |

Table B.31: Difficulty of using the Cucumber syntax

| Anova: Single Factor | | | | | | |
|---|---|---|---|---|---|---|
| *Groups* | *Count* | *Sum* | *Average* | *Variance* | | |
| A | 12 | 26 | 2.1667 | 0.3333 | | |
| B | 12 | 27 | 2.2500 | 0.5682 | | |
| C | 12 | 29 | 2.4167 | 0.6288 | | |
| | | | | | | |
| *Variation Source* | *SS* | *df* | *MS* | *F* | *P-value* | *F crit* |
| Between Groups | 0.3889 | 2 | 0.1944 | 0.3812 | **0.6860** | 3.2849 |
| Within Groups | 16.8333 | 33 | 0.5101 | | | |
| Total | 17.2222 | 35 | | | | |

Table B.32: Rather write tests in free format

| Anova: Single Factor | | | | | | |
|---|---|---|---|---|---|---|
| *Groups* | *Count* | *Sum* | *Average* | *Variance* | | |
| A | 12 | 39 | 3.2500 | 0.7500 | | |
| B | 12 | 31 | 2.5833 | 0.8106 | | |
| C | 12 | 34 | 2.8333 | 1.6061 | | |
| | | | | | | |
| *Variation Source* | *SS* | *df* | *MS* | *F* | *P-value* | *F crit* |
| Between Groups | 2.7222 | 2 | 1.3611 | 1.2895 | **0.2889** | 3.2849 |
| Within Groups | 34.8333 | 33 | 1.0556 | | | |
| Total | 37.5556 | 35 | | | | |

Table B.33: Cucumber help

| t-Test: Two-Sample Assuming Unequal Variances | | |
|---|---|---|
| | *Text editor* | *Wiki* |
| Mean | 2.916666667 | 2.833333333 |
| Variance | 0.862318841 | 1.606060606 |
| Observations | 24 | 12 |
| Hypothesized Mean Difference | 0 | |
| df | 17 | |
| t Stat | 0.202250875 | |
| P(T<=t) one-tail | **0.421060615** | |
| t Critical one-tail | 1.739606726 | |
| P(T<=t) two-tail | 0.842121229 | |
| t Critical two-tail | 2.109815578 | |

Table B.34: Cucumber help