NTNU

Norwegian University of
Science and Technology

Robustness in Early Phase Software Development

Håkon Haga
Øyvind Skjervold

Master of Science in Computer Science
Submission date: June 2008
Supervisor: Tor Stålhane, IDI

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

Problem Description

Robustness is becoming increasingly important as we become more dependent of software systems. Robust software performs according to its intention for all type of system input and environment changes. Realization of robustness has traditionally been done during the development phase, where it is more expensive to perform changes than in the earlier phases.

This thesis studies a way to move the focus of robustness to the requirement and design phase. During our work with the thesis we will implement and test several ideas from "Robustness in Software Development", a project done during the autumn of 2007. The result of this project was a first, high level requirement specification for the tool that will aid software developers in the realization of robustness in the early stages of the development. The requirement specification will be further elaborated on before advancing to the design and implementation phases. Experiments will be performed in two phases, in order to get feedback to our prototype. First an experiment with students with main focus on usability and error detection in the program will be performed. All output will then be evaluated and the prototype updated correspondingly. Then a test with IT professionals will be performed, with focus on the value of the tool for the industry. This results will be used to updated the prototype, and conclude on the usefulness of the prototype.

The goal for the project is to develop a tool prototype to support robustness realization in early software development phases.

Assignment given: 15. January 2008
Supervisor: Tor Stålhane, IDI

ABSTRACT

Ensuring robustness in software is as important as ever, with the increasing significance of information technology in our lives. Users of any IT system expect and require a high level of uptime. The earlier a threat to robustness is discovered in the development of a system, the cheaper it is to handle. Allowing robustness to come into focus at an early stage of development has been the objective of this master thesis.

This thesis is a continuation of the work that we did in the autumn 2007 (Skjervold and Haga 2007), where we created a requirement specification for a tool that can aid system developers in realizing robustness during their design. This requirement specification was based upon interviews with software development companies in Trondheim, one which we performed usability testing with in this thesis.

We have developed a tool based upon the requirements from our previous work, along with some additional requirements in the early phases of this thesis. After developing a first version of the tool, a usability test was performed on 11 students. The feedback we got was evaluated and the implementation was updated correspondingly. A second test was performed, focusing on both the usability and the value of the tool, with four system developers from a software development company in Trondheim. The responses to the usability were divided, but mostly positive, and helpful. Some of the changes suggested were implemented, and the rest was inserted as further work. For the value of the tool, there were strong opinions amongst the four developers, as expected, followed by a constructive discussion. The consensus was that the tool had good potential, but the professional developers felt it needed some improvements and changes. Most of these suggested improvements were too time consuming to address in this thesis, and are therefore inserted as further work.

This report consists of four parts. Part I describes the state of the art and requirement specification. In Part II the experiments that was run are described and evaluated, and finally the implementation of the tool is shown in Part III. In Part IV the bibliography and appendices are shown.

PREFACE

This master thesis was written as part of our *MSc* at Department of Computer and Information Science, at the Norwegian University of Science and Technolohy (NTNU), spring 2008. It extends the work done in the project *Robustness in Software Development (Skjervold and Haga 2007),* by the same authors in autumn 2007.

We would like to thank Professor Dr. Tor Stålhane at IDI, NTNU for his inputs to this report and his good spirits during our meetings. Thanks also to the students participating in our student test, and to the company, which wish to remain anomynous, that participated in business test.

Trondheim, June 7, 2008

_____ _____

Øyvind Skjervold Håkon Haga

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

PART I
INTRODUCTION

1 INTRODUCTION

This master thesis is the follow-up work for the project *Robustness in Software Development* by Haga and Skjervold, autumn 2007 (Skjervold and Haga 2007). The work done in this thesis includes reviewing the requirements in (Skjervold and Haga 2007), and designing and implementing the tool. The tool was also tested: The graphical user interface (GUI) was tested on students and a functional prototype was tested on system developers. The results from these tests were used as design input to the final product.

The main goal for this project was to support development of robust software by providing a prototype of the tool. The tool is developed to support the industry when dealing with robustness in software development. The tool will help developers uncover errors related to input to the system at an earlier stage of development and will save both time and money for the developers.

1.1 STATE OF THE ART

It is necessary to have an understanding of which methods that have been used earlier in a field before developing or using new ones. In this chapter some state of the art solutions used to achieve robustness are presented. Some of the features in the solutions are general, and some are created especially for robustness.

1.1.1 AGENT-BASED SOFTWARE REDUNDANCY

This method is based on the hypothesis that robustness may be increased through redundancy. The redundancy is achieved by using agents. Reinforcement learning is used to build up trust between the agents. Redundancy applies to both software and hardware, the problem with hardware redundancy is that any amount of redundant hardware can fail because of the same faulty software. One way to build fault tolerant software systems is N-version programming (NVP). A major problem with this method is the relation between minimizing probability of getting same results in different versions and maximizing the version development independency (Turlapati and Huhns 2005).

Cooperation between agents is one way of ensuring software redundancy. Multiagents may learn by trials, errors and cooperation by sharing instantaneously information. A voting technique is used to have the agents reinforced. This is important in multiagent systems because one agent may learn from other agents' performances (Huhns, Holderfield et al. 2003).

1.1.2 JACOBSON'S ANALYSIS METHOD

This method was developed by Ivar Jacobson (Rosenberg and Scott 1999). The method is an intermediate level of design between use cases and software design. The method identifies a set of objects that participates in the analyzed use case. The objects are classified into three stereotypes (Zhou and Stålhane 2004):

1. Boundary objects, which the actors use when communicating with the system
2. Entity objects, which are usually objects from the domain model
3. Control objects, which "connects" the boundary objects and entity objects

The objects are shown in Figure 1. There are rules for interaction between these objects which are listed below and also represented graphically in Figure 2.

1. Actors can only talk to boundary objects
2. Boundary objects can only talk to Control objects and Actors
3. Entity objects can only talk to Control objects
4. Control objects can talk to boundary objects, other Control objects, but not to Actors

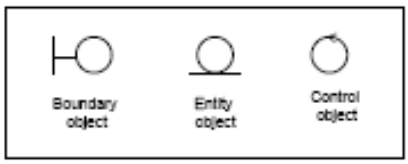

Figure 1: Representation of the classified objects in Jacobson's analysis model

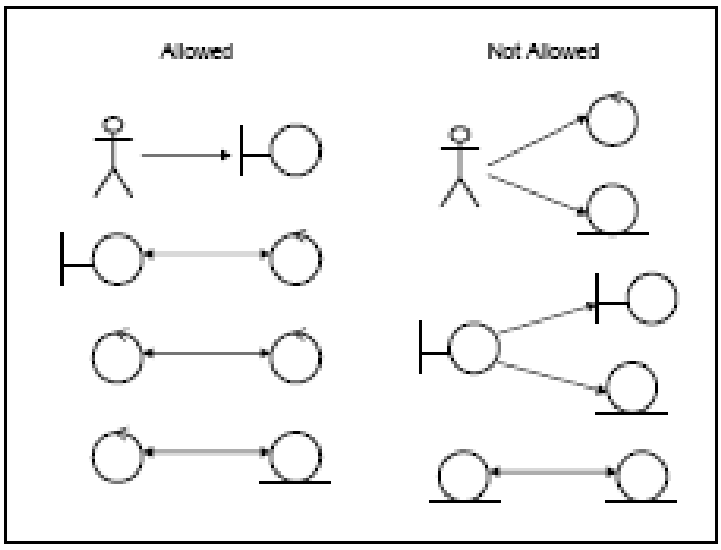

Figure 2: Rules of interactions between objects in Jacobson's analysis model

This method has similarities to our system, e.g. the focus on input validation.

2 REQUIREMENTS SPECIFICATION

The requirements are presented as a requirements list and use cases. In the requirements list the requirements are presented as **id** *(priority)* – <description>, where id is a unique identification, the priority is High, Medium or Low and the description is a short explanation of the requirement.

2.1 ORIGINAL FUNCTIONAL REQUIREMENTS

Quite a few changes have been made to the original functional requirements, and the new functional requirements are listed in *section 2.2 - Final functional requirements.* Below the original requirements are shown.

OF1 (H) - The system shall accept textual use cases as input. The user can upload the textual use case to the system and the system will interpret it. A method for identifying, comparing or categorizing the use cases' Input items is needed, to realize requirement OF3. Input items in the textual use cases are the input data to the future system. If they cannot be detected automatically, a manual solution must be used.

OF2 (H) - The Input items from the textual use cases can be related to one or more Failure modes. The relationship between textual use cases and Failure modes can be many-to-many.

OF3 (H) - The system shall suggest Failure modes for the Input items from the textual use case. The user will be presented with a list of relevant Failure modes the system has identified as relevant for the use case, with the option to choose the ones the user finds appropriate for his use case.

OF4 (H) - New Failure modes can be entered into the system by the user, and connected to relevant Input items.

OF5 (H) - The Failure modes stored in the system can be connected to one or more Barriers. The relationship between Failure modes and Barriers can be many-to-many.

OF6 (H) - The user will be presented with a list of Barriers the system has found for the Failure modes chosen in OF3, with the option to choose the ones he find relevant for his use case.

OF7 (H) - New Barriers can be entered into the system by the user, and connected to relevant Failure modes.

OF8 (H) - The Barriers stored in the system can be connected to one or more Actions. The relationship between Barriers and Actions can be many-to-many.

OF9 (H) - The user will be presented with a list of Actions the system has found for the Barriers chosen in OF6, with the option to accept the ones he find relevant for the present use case.

OF10 (H) - New Actions can be entered into the system by the user, and connected to relevant Barriers.

OF11 (H) - The Actions stored in the system can be connected to one or more Tests. The relationship between Actions and Tests can be many-to-many.

OF12 (H) - The user will be presented with a list of Tests the system has found for the Actions chosen in OF9, with the option to accept those he find relevant for his use case.

OF13 (H) - New Tests can be entered into the system by the user, and connected to relevant Actions.

2.2 FINAL FUNCTIONAL REQUIREMENTS

In this section the updated and final functional requirements are listed. These are based on the original requirements, and were changed to fit our updated knowledge and perception of the tool. There are fewer requirements than the original list, but they are more accurate and correct.

F1 *(H)* – The system shall present Input items to the user so that the user can choose the Input items relevant to his use cases.

F2 *(H)* – The system shall present Failure modes, Barriers, Actions and Tests for the chosen Input items.

F3 *(H)* – New Input items can be entered by the user and the user shall be to add Failure modes, Barriers, Actions and Tests for the new Input items.

F4 *(H)* – Each Input item, Failure mode, Barrier, Action and Test shall have a name and a description that shall be editable for the user.

F5 *(H)* – The user shall be able to add Failure modes, Barriers, Actions and Tests to existing Input items.

F6 *(M)* – The user shall be able to delete Input items, Failure modes, Barriers, Actions and Tests from the data storage.

F7 *(M)* – The relationship between Input items and Failure modes should be one-to many.

F8 *(M)* – The relationship between Failure modes and Barriers should be one-to-many.

F9 *(M)* – The relationship between Barriers and Actions should be one-to-many.

F10 *(M)* – The relationship between Actions and Tests should be one-to-many.

F11 *(H)* – After the user has selected the relevant items and confirmed his choices, the system shall present them in a well arranged manner.

The most important change from the original to the final requirements was that the system would no longer support uploading of textual use cases or automatic discovering of Input items from these. If the application should identify Input items from the textual use cases, complex algorithms and code would be needed. This functionality was considered to be unimportant for this thesis, and hence this requirement (OF1) was removed and the user now has to register Input items manually. Another change was that the relationship between the items are no longer many-to-many, but one-to-many. The intention of many-to-many was that for instance a Barrier could be used by many Failure modes, and that the database could keep track of the relations between the parents and children. These requirements were discarded due to the amount of work required to fulfill them.

2.3 NON-FUNCTIONAL REQUIREMENTS

The non-functional requirements state how the system should be experienced by the user. There were no important changes compared to (Skjervold and Haga 2007), but they are listed here to complete the requirements specification.

NF1 *(H)* - The effort needed to employ the system in a company should be low. The empirical study from (Skjervold and Haga 2007) concludes that the industry require a low effort for employing the tool, but they are willing to learn new methods if the possible payoff is good. It should therefore be possible to check whether the system can be useful for the company in less than one working-day.

NF2 *(M)* - The system shall focus on usability, and the graphical user interface should therefore be intuitive to most users. 90% of the users should learn how to use the main functions in the system in less than one hour (with guidance).

NF3 *(H)* - The advantages of the system should be clear to the user, or at least to the company's management. This means that the system's main functionalities and their benefits should easy to present to the companies.

2.4 USE CASES

This section contains the use cases that describe all the functional requirements stated earlier in this chapter except requirements F6 through F9, which we did not find suitable for use case representation.

Figure 3 shows the use case representing requirement F1 and F2. The tool presents the Input items in the data storage, and the user chooses the ones relevant for his system. When the user chooses an Input item, the tool suggests Failure modes, Barriers, Actions and Tests that corresponds to the chosen Input item.

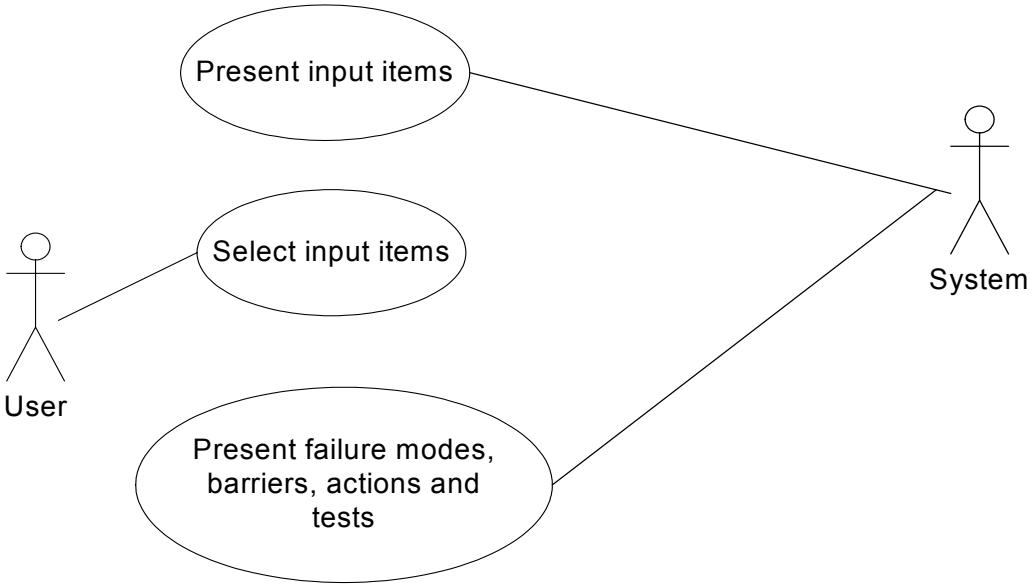

Figure 3: Use case representing requirement F1 and F2

If the user needs additional Input items to the ones suggested by the tool, the user must add them; this is shown in Figure 4. After adding the Input item the Failure modes, Barriers, Actions and Tests can be added by the user. When an item is added the data storage is updated.

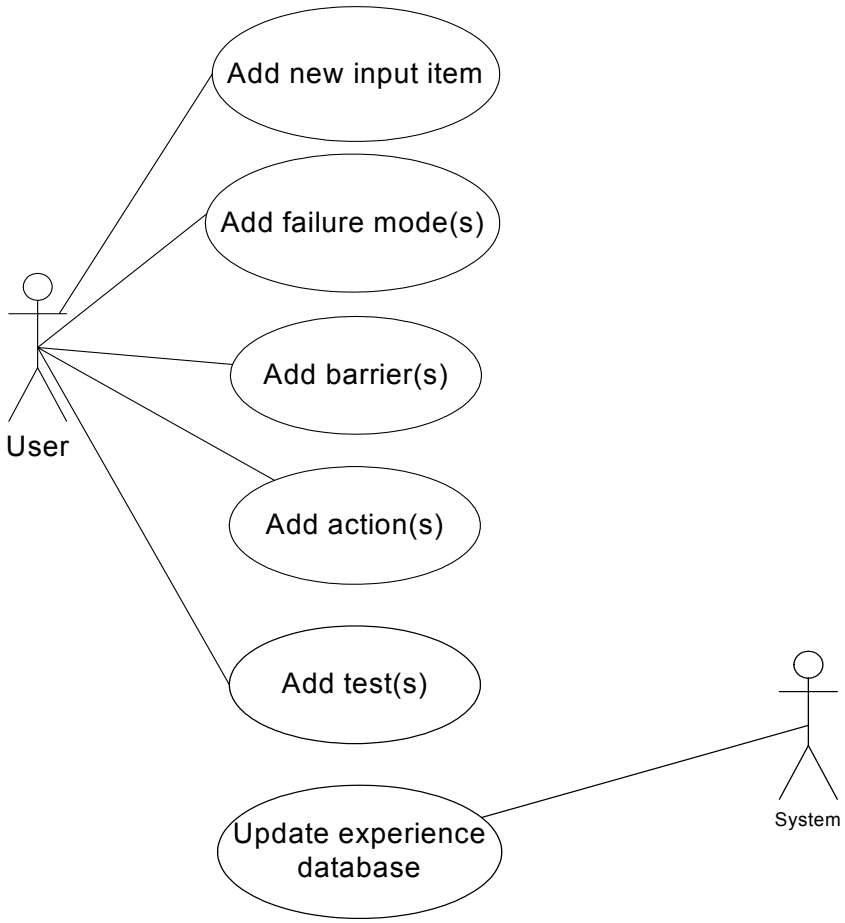

Figure 4: Use case representing requirement F3

Each item has a description and a name. The name and description are presented by the tool for the chosen item when it is highlighted. The user can edit both name and description for items and when the user click the save-button the changes are stored in the data storage. The use case shown in Figure 5 describes requirement F4.

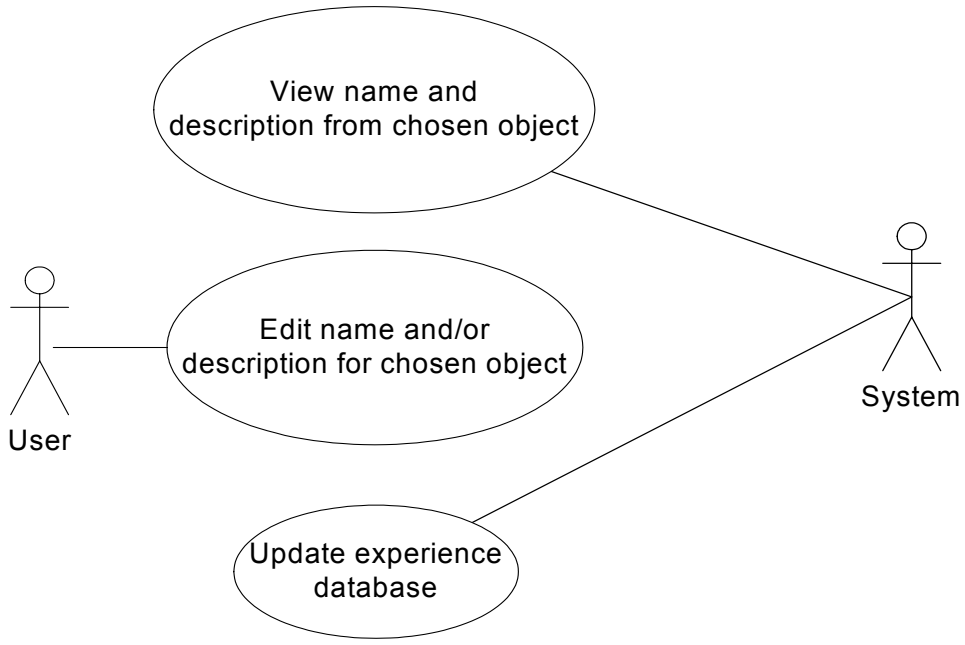

Figure 5: Use case representing requirement F4

The user can add child items to a chosen Input item. This is shown in Figure 6. The tool presents the Input items from the data storage; the user then chooses which Input item to add new child items to. After adding an item the data storage is updated. This is according to requirement F5.

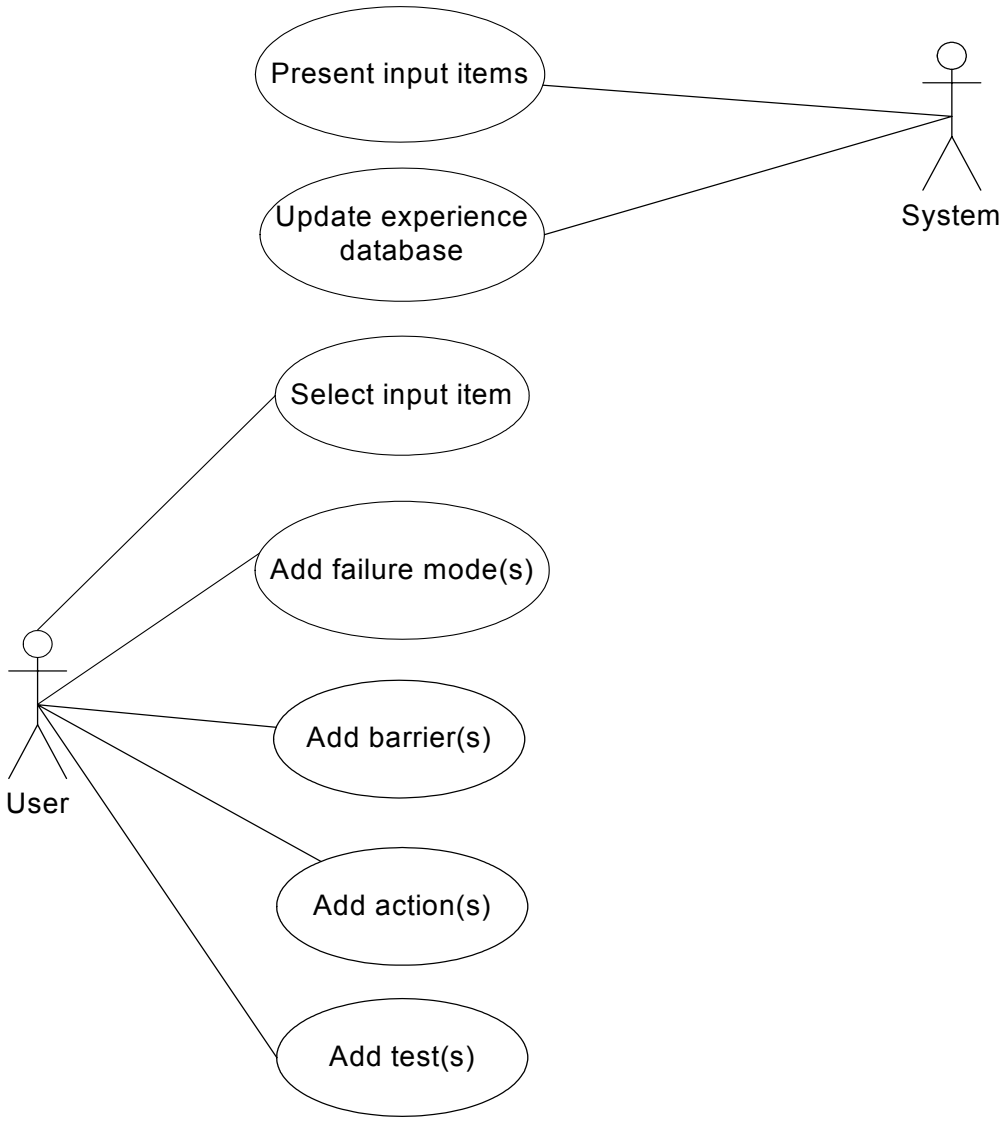

Figure 6: Use case representing requirement F5

The user can delete Input items and child items. This is shown in Figure 7. The system presents the available Input items and corresponding child items from the data storage. The user may choose to delete any item. When deleting an item, all child items are also deleted. After deleting an item the data storage is updated. This is according to requirement F6.

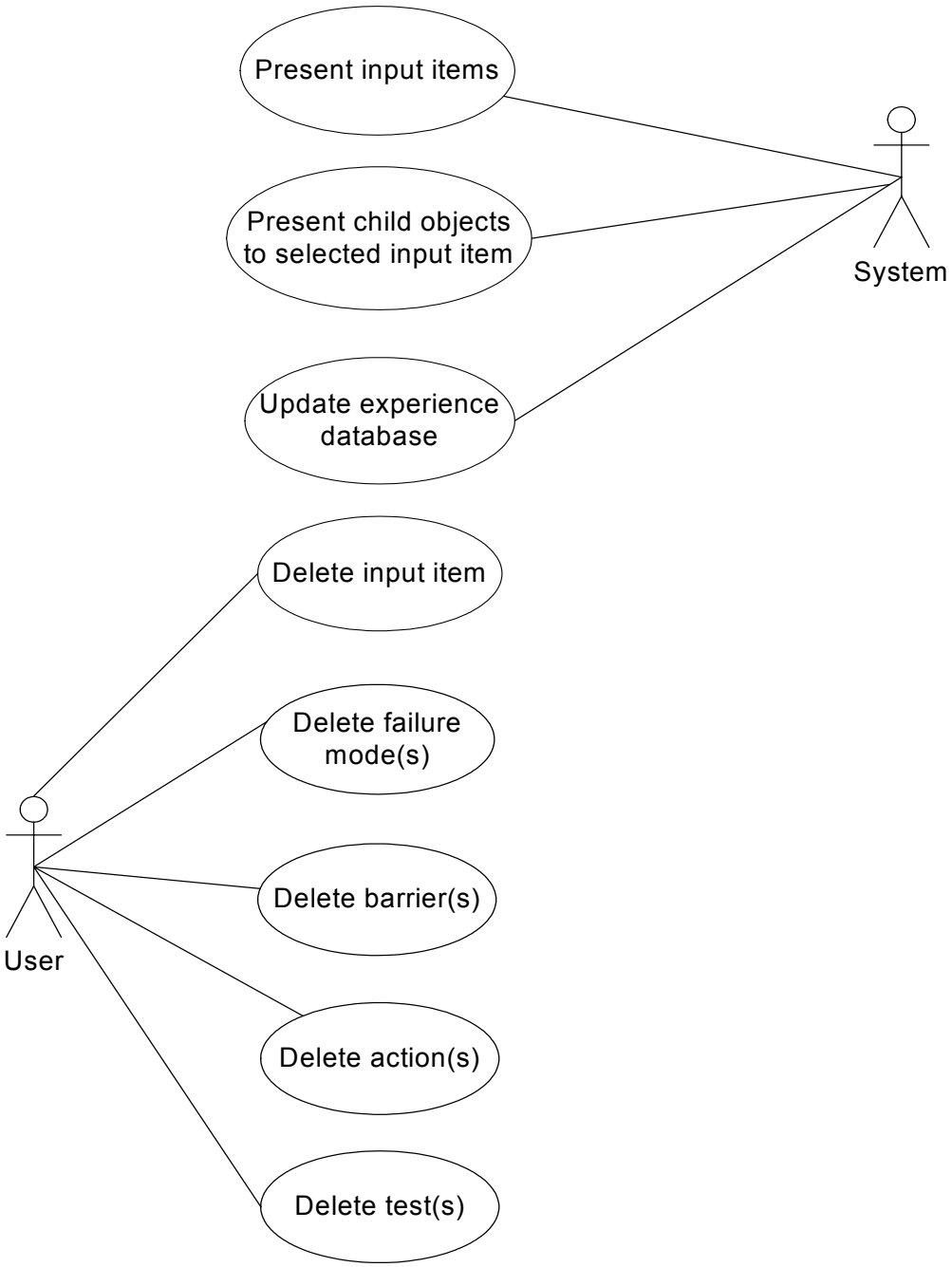

Figure 7: Use case representing requirement F6

When the user has chosen the relevant Input items and made necessary changes to the child items, it shall be possible to confirm these choices. When they are confirmed, the system shall present the results to the user. These should be presented in a way that makes it is easy for the user to get an overview of the result. Figure 8 shows the use case representing requirement F11.

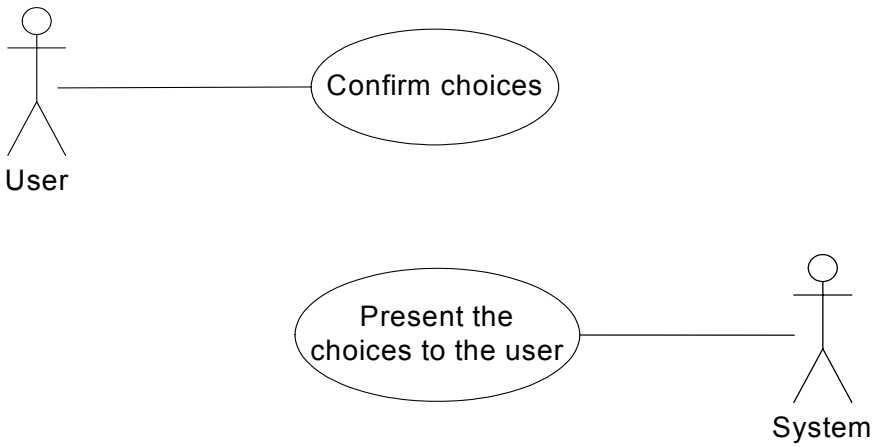

Figure 8: Use case representing requirement F11

3 RESEARCH METHODS

In this chapter the methods used in this thesis are presented. The descriptions of the methods are taken from (Wohlin, Runeson et al. 2000).

3.1 THE EXPERIMENTS

The best way to see if a tool is useful is to let others persons try it out. In order to do this in an orderly fashion we used two experiments – one with students and one with IT professionals. Off-line experiment was preferred since it has a higher level of control under normal conditions. An off-line experiment is performed in a controlled environment where the conditions are simulated to correspond to the real world, as opposed to an on-line experiment that is performed in the field under normal conditions. Since both the students and professionals were available in Trondheim, the off-line experiment was also the most practical.

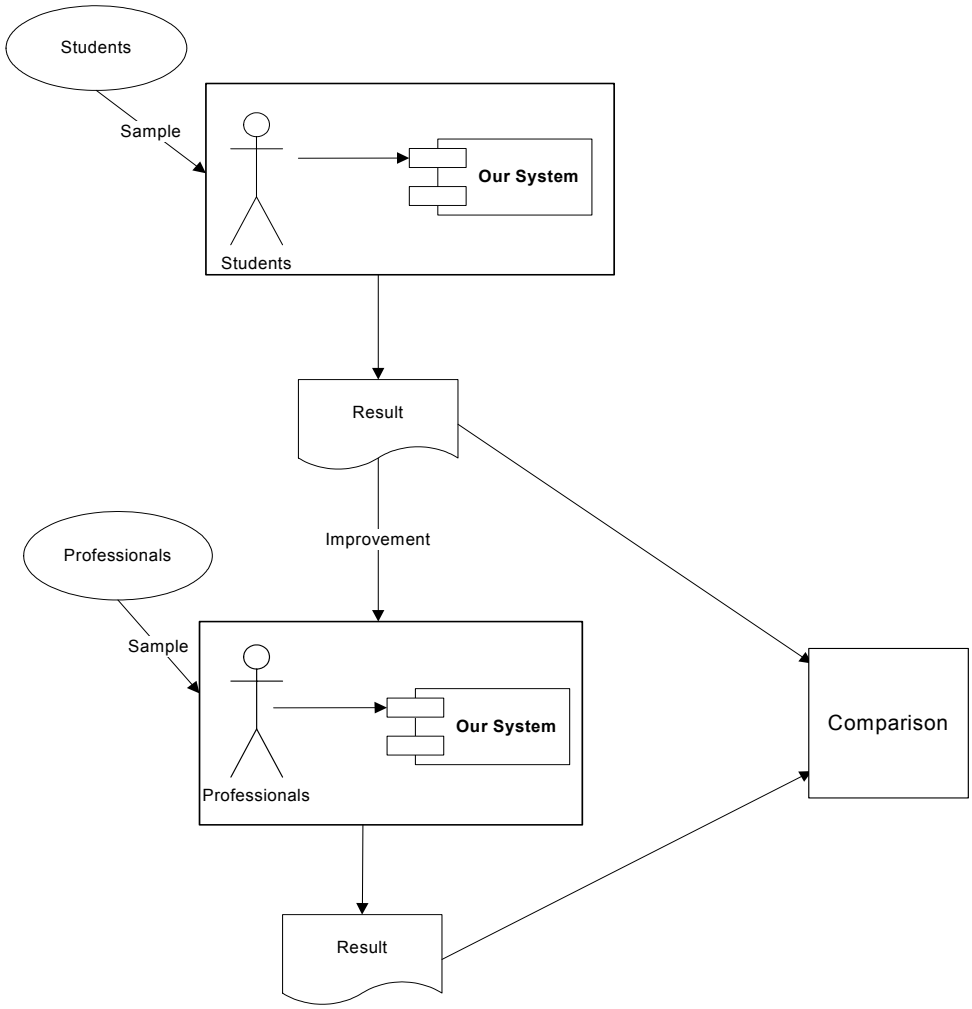

Figure 9: Experiment planning with dependent and independent variables

In experiments a set of variables are defined and then sampled over. In our case we had students and professionals as two independent variables and their results from trying out the tool as dependent variables. Figure 9 shows how we ran the first experiment with the students and used the results to improve the system. We then ran a second experiment with professionals using the improved tool. The results from both experiments were used to (1) compare the students and the professionals and (2) check whether both groups found the program easy to use.

3.1.1 EXPERIMENT CHARACTERISTICS

Experiments have several characteristics depending on which aspects they are used to investigate. Some of these aspects are listed below.

- Confirm theories, i.e. to test existing theories
- Confirm conventional wisdom, i.e. to test people's conceptions
- **Explore relationships, i.e. to test that certain relationships holds**
- Evaluate the accuracy of models, i.e. to test that the accuracy of certain models is as expected
- Validate measures, i.e. to ensure that a measure actually measure what it is supposed to

The third aspect, marked bold, was the one that fit our experiment. We investigated the usability and functionality for our system. First we tested how the students reacted to our graphical user interface. Thus, the tests assess people's conceptions to our user interface. Non-functional requirement 2 states that 90% of the users should understand the most significant parts of our system within one hour usage with guidance. Thus, the experiment also helped us to test the usability of the system.

3.1.2 EXPERIMENT PROCESS

The experiment process has the following steps:

- Definition
- Planning
- Operation
- Analysis and interpretation
- Presentation and package

The definitions include defining problem and goals. The main goal of our project was to support the development of robust software. The main goal of the experiments, however, was to see how user-friendly the system is. A software tool like ours needs a high level of usability, and the most important problem to be explored in the experiments was the quality of the user interface. In addition, the system must provide the company's development

procedure with added value. The goal of the experiments is written below based on the goal syntax suggested in (Wohlin, Runeson et al. 2000):

Analyze the system
for the purpose of improving the system
with respect to usability and functionality
from the developers' point of view
in the context of the developers M. Sc

In the planning phase the design of the experiment is determined, and the threats to the validity of the experiment evaluated. Our experiment design is shown in Figure 9. We also made an experiment document that we handed out to the participants when the testing session started. In this document there is information of the experiment, the model, examples, tasks and a feedback form. The filled-in feedback form is shown as the result in Figure 9. This feedback contains both quantitative and qualitative information. The business experiment document is shown in Appendix A, and the student experiment document is shown in Appendix E.

The most important threat to validity for the comparison part of this experiment was that the students and the professionals were testing different tools, since the code was modified between the two experiments. Threats are discussed in *section 3.1.3 - Threats to validity*. The other aspects in the experiment process are not discussed since there was no focus on these before the execution of the experiment.

3.1.3 THREATS TO VALIDITY

A general checklist in (Wohlin, Runeson et al. 2000) was used as a basis for identifying threats to our experiment. The threats are categorized as follow: threats to conclusion validity, threats to construction validity, threats to internal validity and threats to external validity. Only the threats that are applicable to our experiment are described. In *section 9.3.2 – Experimental threats* the threats that applied to our experiments are discussed.

3.1.3.1 THREATS TO CONCLUSION VALIDITY

These threats are concerned with the issues that affect the ability to draw the correct conclusion from the experiment results. The threats that could affect our experiment were:

- **Low statistical power.** If the power of a statistical test is low, the risk of drawing the wrong conclusion is high. The power of a statistical test is the ability the test has to reveal a true pattern in the data. Our sample sizes were lower than recommended for these tests. This was a serious threat since a low sample size reduced the probability that the sample represents the whole population.

- **Reliability of measures.** Measurement in our case depended on several factors e.g. question formulation, instrumentation and layout. In our case there were only subjective measures that could be less reliable that the objective one. As mentioned earlier, we also compared the results where the latter system was modified before the last testing session. This did, however, only threaten the comparison part of the experiment, not the usability assessment part.

3.1.3.2 THREATS TO INTERNAL VALIDITY

These threats are influences that can affect the independent variable regarding causality, without the researcher's knowledge. The threats that could affect our experiment were:

- **Instrumentation.** Poor formulations in the feedback form, example or walkthrough could affect the answers from the participants. Avoiding leading questions in the feedback form was crucial. Some of the questions could also be connected, and this could lead the participants to answer in the same way on several questions, whether quantitative or qualitative. To avoid this, the questions were tested, corrected and approved before starting the experiment.
- **Selection.** This is the effect of variation in human performance. Since the participants in the first experiment volunteered for this experiment they were generally more positive than other in the population. We were therefore aware that the feedback from the first test could be more positive than from the second test.

3.1.3.3 THREATS TO CONSTRUCT VALIDITY

Generalizing results from the experiment is the concern of the construct validity. The threats that could affect our experiment were:

- **Hypothesis guessing.** This threat regarded participants that were curious of what the purpose of the experiment was. This could lead them to answer what they thought were the "correct" answers rather than their opinion.
- **Evaluating apprehension.** Some people do not like to be evaluated. This was a threat that could affect our experiment, especially the first testing phase. The students could be afraid of looking stupid in front of class mates. This risk was probably not applicable to the same degree for the professionals as for the students. To avoid this threat we gave the students the opportunity to choose which PC they wanted to use. This means that they could keep a distance from other students if they wanted to. Hence, the students in the sample were probably not affected by this threat.

3.1.3.4 THREATS TO EXTERNAL VALIDITY

The external validity regards the generalization of the experiment's results to an industrial practice. According to (Wohlin, Runeson et al. 2000) there are three types of interactions with the treatment: people, place and time:

- **Interaction of setting and treatment.** This threat regards not having the material that is representative of the industrial practice. This threat also includes using fictional problems in the experiment. In the first session the usability was the only thing that was evaluated. In the second session we used a fictional use case that the professionals used when performing the experiment. We made this use case in a way so that each of the participants could recognize the problem, and extract the Input items from it. The use case is shown in *Appendix A.1.1 – Textual use case.*
- **Interaction of history and treatment.** One should avoid arranging the experiment on a special day or time that may affect the results. E.g. avoid arranging the experiment right after a robustness failure since this could affect the result. The company scheduled the date for the professional experiment so this was, as far as we know, arranged without any influence from other events.

3.1.3.5 THREATS TO OUR EXPERIMENT

The statistical power was one of the threats that we addressed. Even if the results were statistical significant it was important not to draw too strong conclusions based on this. As discussed earlier, our dependent variable was the feedback from the participants. Since the first result was used to improve the system before the second session we had to be aware of the difference in the system that the participants evaluated. This would affect the results in the second testing session. The results from both sessions were compared but the statistical tests lose some of its significance when the results are not based on the same system.

3.2 INTERVIEW

As part of the experiment, we performed interviews with the professionals. The interviews were used to collect qualitative information. We made an interview guide, see *Appendix C*, which we used in the interviews. An interview guide allows you to have control over the interview situation. If the interviewee answers the questions accurately and the answer naturally leads to new questions it might not be necessary to use an interview guide. Otherwise, the guide supports the interviewer with new questions so the interview flows smoothly (Ringdal 2007). (Wohlin, Runeson et al. 2000) indicate the following advantages of using interviews instead of other data collection methods like for instance surveys:

- High response rates
- An interviewer will generally decrease the number of "blank" answers because he can answer questions listed in the questionnaire
- The interviewer can observe and ask follow-up questions. The interviewer may also add questions that were not thought of before the interview

According to (Ringdal 2007), interviews should be used when the extent of nearness is high and the extent of standardization is low – see the details in Table 1. The extent of nearness represents the geographical distance to the participant. Our nearness was high as the interview was performed in the offices of the company. The other dimension is the extent of standardization. Lower standardization gives the interviewer more flexibility. Low standardization fit our interview well since we wanted feedback of both positive and negative aspects from the interviewee, and the questions in the interview were not answered to the same extent by each participant. This means that the questions asked to each interviewee were not strictly the same – the interview was performed as a conversation. Since we used a group conversation it was natural to keep the conversation going, but also be sure to include every participant so that everyone had the opportunity to state their opinions.

Table 1: Data collection using interview or surveys

Extent of nearness	Method	Extent of standardization	
		Low	**High**
Low	Mail, e-mail	-	Survey
Medium	Phone	Conversation interview	Modern survey interview
High	Personal visit	Conversation interview	Classic survey interview

The drawback with interviews is that they take a large amount of time to perform and analyze compared to surveys. How useful it is to perform interviews depend on the sample size. For big samples one often just wants to see a summary of the answers, while for a small sample it can be useful to have an in-depth feedback from each participant. In our case we got enough information from the experiment with the students. In the second session the sample size was low, and we thus decided to arrange an interview instead of a survey. This was because of the small sample size and because we could ask additional questions to the developers depending on their opinions. Also, given the fact that our interviewees in the second session were experienced system developers, we wanted a more thorough and detailed feedback which an interview gave us.

PART II
EXPERIMENTS

4 EXPERIMENT EXECUTION

In this chapter we discuss the execution of the experiment. We have pointed out what modifications we did compared to the research methods. The object of the experiment was to find answers to our research questions.

- **RQ1: Is the system easy to use?**
  We will test the system's usability – first on students and then on IT professionals.
- **RQ2: Does the system add value to the developing process?**
  We will assess the added value by interviewing the IT professionals.

4.1 STUDENTS TESTING THE USER INTERFACE

To quickly test the usability of the GUI we arranged a test session, and invited students to participate. The students were second grade Master of science students taking the course Software Engineering. This experiment was important in the development because the students could easily identify errors or lack of functionality in the tool, and their opinion regarding the GUI was important feedback for us. The students were rewarded with a wage corresponding to NTNU's policy for such activities. An invitation to participate was published on the course web site and also advertised in class. The students tested the program, and answered multiple choice questions and gave opinions in free text. The suggestions that we found reasonable was analyzed in-depth, and the GUI was improved before moving on to the next testing phase. We did not find it necessary to perform interviews with the students. However, after the experience with the group discussion with the professionals we saw that a group discussion could have been appropriate for the students as well. On the other hand, the student sample size was larger and to have a well arranged discussion group we would have had to prepare this before the testing session.

Each student was given a PC with our tool installed. They were also given the experiment document shown in *Appendix E*. Before they started reading the document we informed the students what was going to happen so it would be easier for them to understand the information in the document. When the students had read the experiment document they started the program and began testing the system's usability. They were allowed to ask questions, and we answered them if they were not able to continue unless they got more info. We did not, however, want to help them too much, but rather observe their handling of the problems. After completing the tasks they answered the questions in the feedback form.

When the students had finished the experiment we analyzed the results, corrected the errors that were discovered in the system and discussed if changes to the GUI were necessary. The results are shown and analyzed in *chapter 5 - Experiment evaluation.* The testing session was performed according to plan. All the students finished the experiment and delivered the feedback form within the scheduled time. The session lasted about one hour.

4.2 PROFESSIONALS TESTING THE FUNCTIONALITY

The functionality and usability was tested on professionals after evaluating the student test since the tool could be improved between the tests. The purpose of this test was to get feedback from the system developers on their opinion of both utility value and the usability of the tool. Some of the questions from the student test were used to be able to compare the results. Performing interviews in addition to the questionnaire was appropriate. An interview guide was prepared, and the plan was to interview the participants separately, but it ended up as a group interview.

Since an off-line experiment was chosen, software developers located in Trondheim were used. This test was performed in the company's offices in Trondheim, and four of the employees participated in the testing session. An information letter about the experiment was sent to the participants in advance to prepare them for the experiment. The test started with a short intro from us, and the participants were given the user manual shown in section 8.5, and the experiment document shown in Appendix A. They started by reading the handouts and installing our program. They also read the example in Appendix A.1 before moving on to the experiment tasks. We observed them while they tested the program, and answered their questions as accurate as possible. The only exception was if the question dealt with some of the functions in the program that we wanted them to test. When such questions came up we made notes that later could be used to identify modifications to the system. When all the participants had finished the testing, we performed an "all together" interview session. We originally planned to interview them one by one, but since the participants were eager to discuss the program together, the group interview was a better solution. Since the participants had commented everything regarding the user interface in the feedback form, the discussion quickly turned to the utility value of the program. From this session we got a lot of interesting ideas from the developers, and the results are analyzed in the *chapter 5 - Experiment evaluation.*

5 EXPERIMENT EVALUATION

In this chapter the results from the questionnaires are presented. Each question in the questionnaire has been given a score. This was done by calculating a mean score, using the numbers from 1 (least positive alternative) to 4 (most positive alternative). When analyzing results from experiments like this it is important to remember that people see things in different ways. Some variation would therefore occur in the results even though we expect most of the answers to be similar. NF2 stated that 90% of the users should be able to learn how to use the most important functions within an hour of use. In this chapter we check if the experiments imply that this requirement is met.

5.1 USER INTERFACE TEST PERFORMED ON STUDENTS

This section summarizes the results from the user interface test. The experiment gave both quantitative and qualitative answers. The quantitative answers are shown in Figure 10 through Figure 14 and the qualitative answers are shown in *Appendix F*. Our findings are discussed in this section and the consequences of the findings are stated.

5.1.1 QUANTITATIVE RESULTS

The quantitative results showed us that most of the students were satisfied with the program's usability. Each of the students scored each question on a scale from 1 to 4 and mean score for each question was calculated as shown below.

$$mean\ score = \frac{(a * 1) + (b * 2) + (c * 3) + (d * 4)}{n}$$

Here *a* is the number of participants answering the least positive alternative, and so on until *d* that is the number of participants answering the most positive alternative. The letter *n* stands for the total number of participants.

The first question asked whether it was intuitive or not to select input items from the leftmost tree in the program. Figure 10 shows that most of the students did not find this intuitive. This meant that we had to look into the qualitative results and find out what confused the students. The consequences are stated in *section 5.1.2.1 - The Root item* and *section 5.1.2.2 - Button for moving Input items between the trees*. Mean score for question 1 = 3.27.

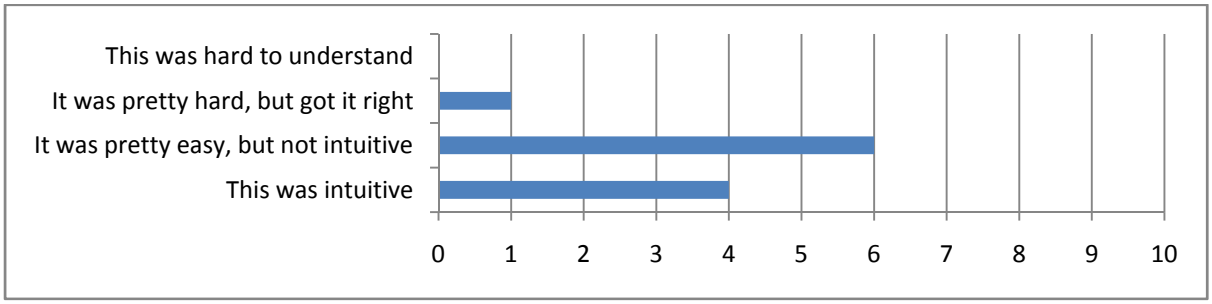

Figure 10: Question 1, ease of selecting Input items

Figure 11 shows that the students did not have problems understanding how to add Input items. The two students that did not find this intuitive claimed that the *root* item was confusing, and that there was some lack of redundancy. This is some of the same results that came up for the previous question, and the consequences for this question were therefore similar to the previous question. Mean score for question 2 = 3.82

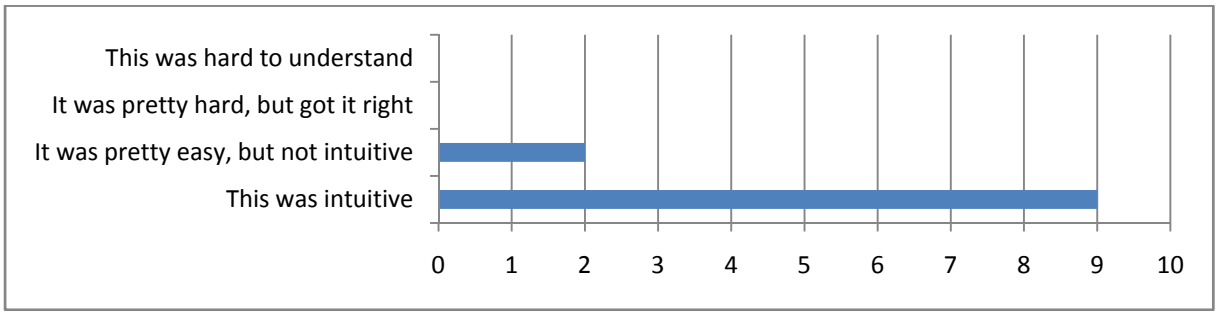

Figure 11: Question 2, ease of adding Input items

Figure 12 shows the results from the third question, and like the previous question, nine students found the task intuitive. In this case one student found the task a bit hard to perform. This student disagreed with the other students regarding the colors of the items' icon. S/he found the colors on the icons confusing, but since none of the other students shared this opinion we decided not to change anything regarding the colors on the icons. It was therefore nothing to report in the *section 5.1.2 - Consequences* from this question. Mean score for question 3= 3.73.

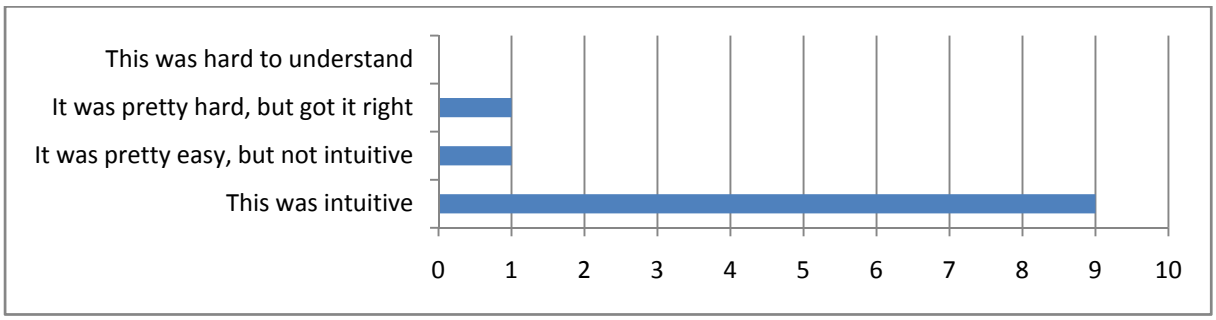

Figure 12: Question 3, ease of selecting child items under selected Input items

Most of the students found it intuitive to add Failure modes, Barriers, Actions and Tests as shown in Figure 13. They found this question quite similar to question 2. The students that did not find this operation intuitive did not claim that something was hard to understand regarding the usability, but that they had a hard time understanding the model, see *Appendix E.1.6. – Data structure*. Even though the concept was explained to them, and they had some documentation we did not expect the students to understand the model in detail. We therefore do not see the need to change anything because of these results. Mean score for question 4 = 3.64.

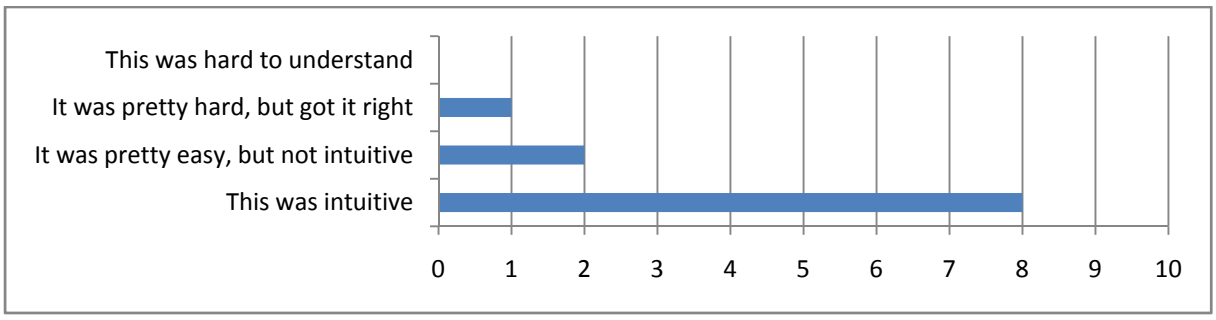

Figure 13: Question 4, ease of adding child items under Input items

Most of the students did not notice the redundant possibilities of adding child items since they performed the tasks intuitively. One student said he missed some options and claimed that it would be easier to move Input items from one tree to the other if there was a button under the trees with this functionality. We chose to follow the advice, and implemented this functionality. This is described in *section 5.1.2 - Consequences*. Tree students noticed that there were some redundant options, but none of them found it confusing. Mean score for question 5 = 3.36.

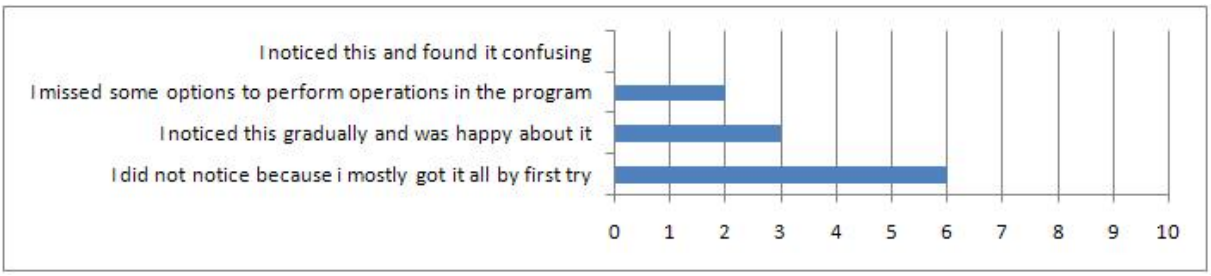

Figure 14: Question 5, noticing the redundant options

When analyzing each of the bar charts in Figure 10 through Figure 14 we see that none of the students have used the most negative alternative on the multiple choice questions. The second most negative alternative implies that the student found it hard but got it right after some time. We thus satisfy the requirement that 90% of the users should learn the program within an hour. This also support our first research question (RQ1) which stated that the program should be easy to use.

The chart shown in Figure 15 summarizes the results from the first testing session.

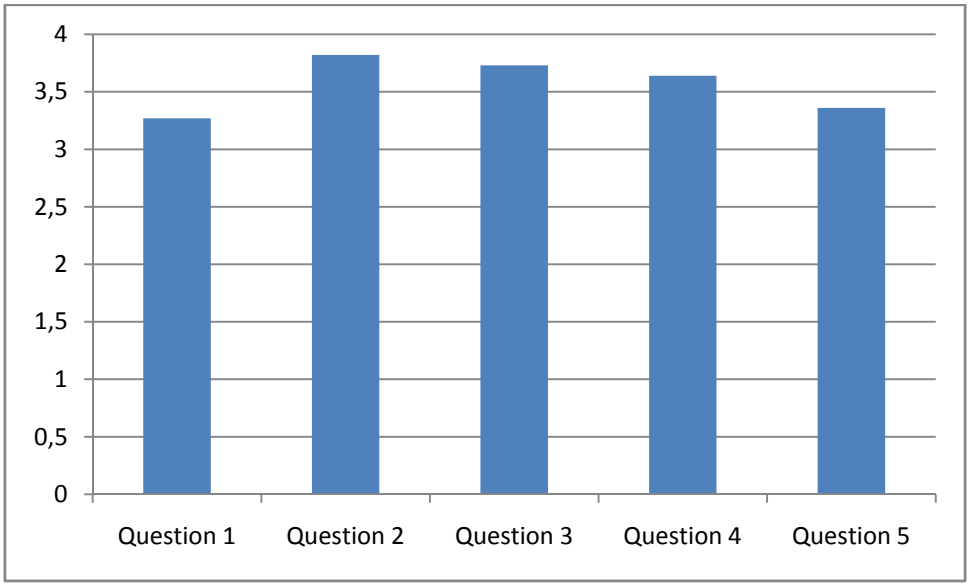

Figure 15: Mean score for each question asked to the students

5.1.2 CONSEQUENCES

Results that lead to modifications of our tool after testing the user interface are described below.

5.1.2.1 THE ROOT ITEM

The *Root item* was originally intended to be a helpful item when adding new Input items. See Figure 16 and Figure 17 for details. The parent item should be highlighted when adding child items, and this was the reason for having the *Root item*. Since Input items were at the top of the tree, it seemed a good idea to have a *Root item* as a parent for Input items. Some of the students found the root item confusing. We decided to remove it, and the Input items now have no parent. When adding Input items a button that is available independently of what is highlighted in the program is used. This button, named *Add Input item*, was added as a replacement and is shown in top of the menu in Figure 18 and Figure 19.

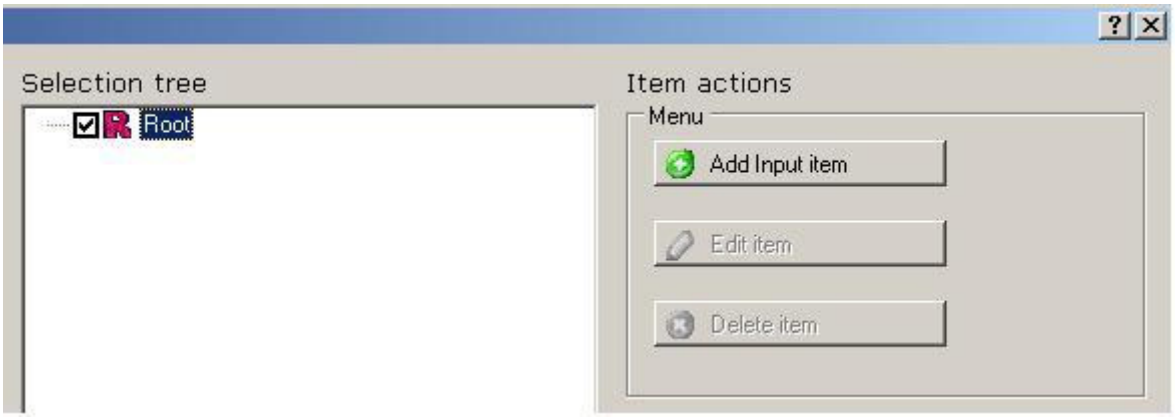

Figure 16: Before GUI test: One can add Input item when highlighting root

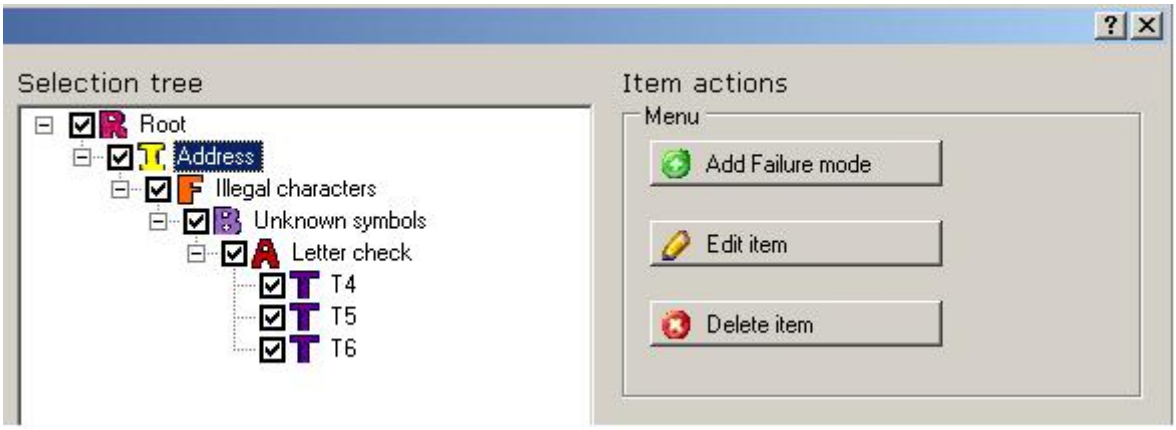

Figure 17: Before GUI test: Cannot add Input item when other than root are highlighted

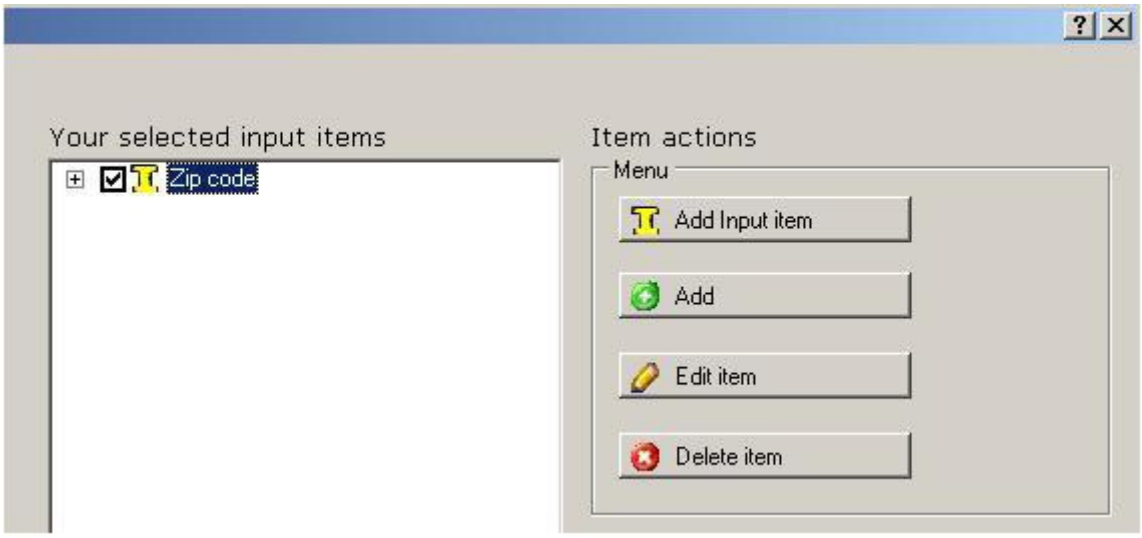

Figure 18: After GUI test: Button added. Can add Input item independent of highlighting

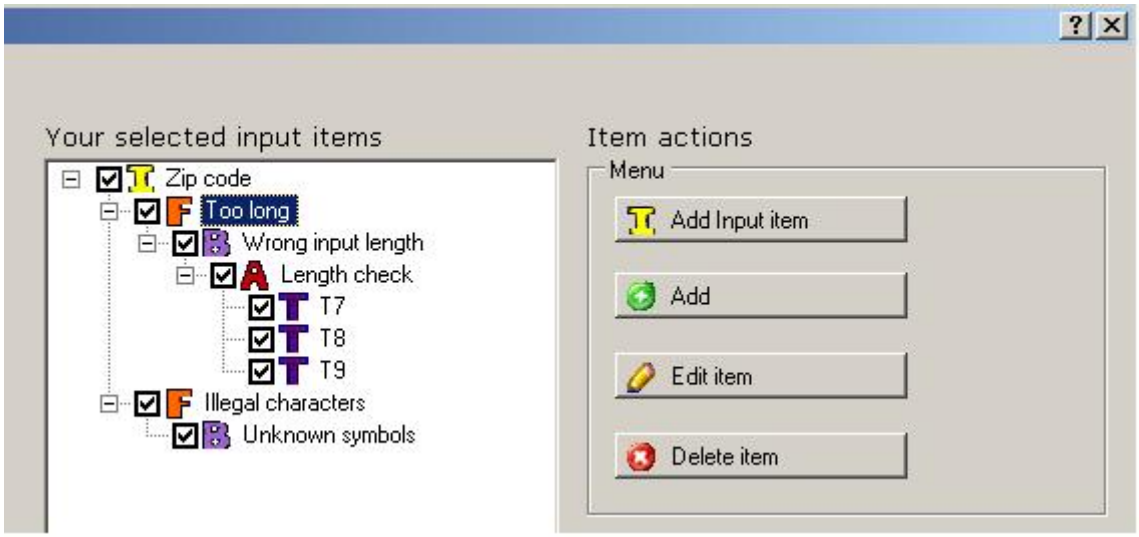

Figure 19: Input items can be added at all times

5.1.2.2 BUTTON FOR MOVING INPUT ITEMS BETWEEN THE TREES

Most of the actions in the application can be performed in several ways. For instance, for moving an Input item from the leftmost tree to the rightmost tree, the user can check the checkbox, double-click or use the right click menu. Some students wanted other alternatives, and as a consequence of this we inserted buttons under the trees to move the Input items between the trees.

5.1.2.3 HEADINGS FOR THE TWO TREES

Several students found it hard to see why both trees were needed. Most of the students understood that the rightmost tree held the selected Input items, but some of them still did not see the difference between the trees. The headings *input item list* (leftmost) and *selection tree* (rightmost) did not help the students to see why the two trees were required. The heading of the leftmost tree was changed to *Input item database*, which should make the difference between the trees clearer. In our opinion, this did not solve the problem completely, but it was the best solution we could come up with without doing major changes to the tool.

5.1.2.4 WINDOW MENU

One of the students requested the window menu bar that is a standard in Windows. We decided not to insert a menu bar since some students claimed that more redundancy could lead to confusion, and the menu bar would have little or no effect on the usability.

5.1.2.5 TREE STRUCTURE IN THE SELECTION TREE

When choosing several Input items, some confusion arose. A student pointed this out and said that s/he found it difficult to separate the Input items in the tree, and suggested to split them by a solid line. Another student mentioned that minimizing the Input items that is not currently used made it easy to maintain a good overview in the selection tree. Regarding presentation of the results from the tool, it was decided to separate each Input item by a solid line. The Input items in the selection tree in the tool are not separated at any time. This was considered, but the user control which was used to show the Input items (TreeView) had no functionality of adding lines and the change was not considered important enough to spend time implementing it.

5.2 FUNCTIONALITY TEST PERFORMED ON PROFESSIONALS

In order to get feedback from the industry, we ran the same test with professional system developers as we ran with the students. Employees in a company from Trondheim, Norway participated on this test. We focused on both functionality and usability in this test. The quantitative answers are shown in Figure 21 through Figure 27 and the qualitative answers are shown in *Appendix B* and discussed in *section 5.2.1 - Qualitative results.* We calculated a mean score based on the quantitative results for the questions that are similar for the tests run with students and with professionals.

5.2.1 QUALITATIVE RESULTS

The feedback was gathered and evaluated, and we made a priority list for changes since it was not enough time to implement them all. The priorities were based on importance of the change, and time needed to fix the problem. The importance of the suggestions should count more than the time to fix it. This is why we have chosen the set (5 (High), 3 (Medium), 1 (Low)) for importance and the set (3 (Low amount), 2 (Medium amount), 1 (High amount)) for the time needed. The priority list is shown in Table 2 and the suggestions are discussed below.

5.2.1.1 BUG IN RIGHT CLICK MENU

When we were in the middle of the professionals' testing session, one of the participants discovered a bug in the program regarding right clicking. When adding an item using the right click menu, nothing seemed to happen. The item was added in the model but not shown in the GUI. When another item was added by using the *add button*, both items became visible in the GUI. This was a bug that we knew how to fix, and we concluded that the bug was important to get rid of so we prioritized it High (5). This bug was fixed immediately after the testing session. It was estimated to take a low amount of time (3).

5.2.1.2 THE ">>" ICON

The ">>" icon was inserted between the two trees after the student test. This was not done because the students missed something, but rather because we found it appropriate to add this icon since it would, in our opinion, make it easier to understand that the input items should be moved from the left to the right tree. A screenshot from the program including the icon is shown in Figure 20. During testing, one of the participants tried to click on this icon because s/he found it intuitive and thought this would move the highlighted Input item to the rightmost tree. We observed this and the participant also did mention it in the feedback form. We decided that if the icon should exist we had to add the intuitive functionality to it. We prioritized this High (5) and found the icon useful as another redundant option to move Input items. We found this pretty easy to change and thought this

could be done in a low amount of time (3). We also added a “<<” sign to move the Input items from right to left.

Table 2: Priority of suggestions from the professionals

Suggestion	Status	Importance	Time	Score (T*I)
Bug in right click menu should be fixed	Fixed	5	3	15
The “>>” icon should have functionality	Fixed	5	3	15
Change background color	Fixed	3	2	6
Hard to see difference between DB and project	Further work	3	1	3
Undo function	Further work	1	2	2
Copy function	Further work	1	2	2
Support field dependencies	Further work	1	1	1
Design level focus	Already ok	1	3	3

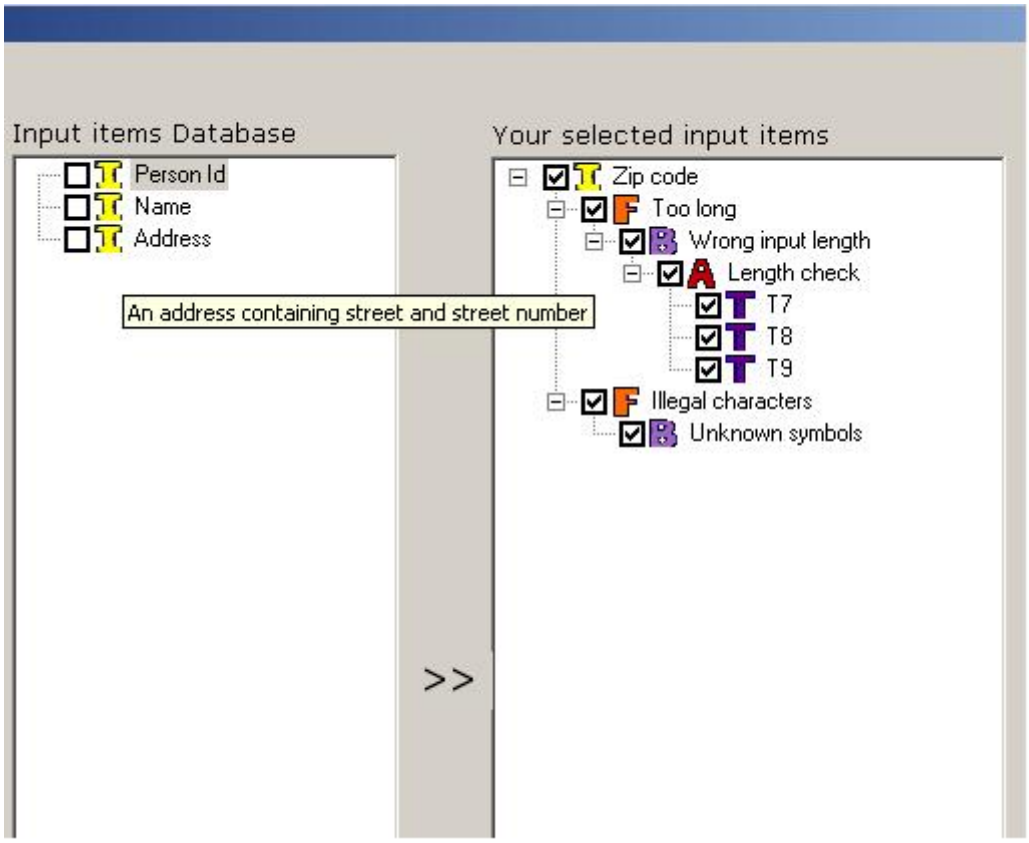

Figure 20: The ”>>” sign implies moving of Input items to the rightmost tree

5.2.1.3 CHANGE BACKGROUND COLOR

Some of the participants found the program’s background color inappropriate. It also seemed as the color changed from computer to computer. Someone thought standard grey was boring and others had a hard time reading what was written because of too similar colors in background and text. We found it important that the background color should be comfortable, and the user should actually not notice it at all. This is why we prioritized this change as medium (3). Changing the color required little time but it was important to find a color that was appropriate for the program. We therefore thought this problem would take medium amount of time (2).

5.2.1.4 DIFFERENCE BETWEEN DATABASE AND PROJECT

In the interview session the system developers said that they wanted the possibility to store each project separately, meaning that all changes done to the items in the right tree could be stored, and later loaded for continuing the work. As the tool currently worked, the Input items were moved between the trees. With this change, the Input items would be copied which would allow specialized changes to the Input items in the right tree. We considered that this would be necessary in some cases, and that it therefore would be an important feature in the program. We did, however, not consider this something that had to be done

to finalize the tool so we have given it medium (3) importance. This was also something we considered would take a large amount of time (1).

5.2.1.5 UNDO FUNCTION

One of the participants in the test wanted an undo function, which we also had thought of before the professional test. This was something we found less important (1) at this stage as we did not consider it important to a prototype. The only action where the undo function would be important was the deletion of an item, and there were warnings in the tool that deletion could not be undone, which somewhat compensated for this. We estimated that adding an undo function would require a medium amount of time (2).

5.2.1.6 COPY FUNCTION

When adding new items that were almost similar to existing items, the participants thought it would be practical to have a feature that allows copying existing items. We agreed that this could save time for the user. In our opinion this was not something that had to be implemented to finalize the program. It was also considered to take a medium amount of time (2) and that the importance is low (1).

5.2.1.7 SUPPORT FIELD DEPENDENCIES

Some of the professional developers mentioned that there could be dependencies between input fields. An example of such fields was day of birth and place of birth. A product that someone wants to buy may have an age limit (e.g. alcohol), and this limit may be different from country to country. We calculated that supporting such dependencies would take a large amount of time (1). This was an aspect that we did not think of as a robustness aspect before the professional test, and in our opinion this is something that was outside our scope. We chose to prioritize this as low (1) since it would change the focus late in the project. We were, however, aware that this was important and it should therefore be considered for further work.

5.2.1.8 DESIGN LEVEL FOCUS

Research question 2 seeks to determine what functionality the company wanted from a tool like ours. After testing the program they had two answers to this question. One was that they wanted a more automatic tool that could be able to generate code and unit tests as for example JUnit[1] does. The other answer was to have the program focus on the design level of

[1] http://www.junit.org/

the software development process. This means that the program should not generate code that could be useful in the implementation phase. When the discussion ended, the employees decided that focusing on the design level would be the best solution. Since the program already had this focus it was no need to change this, but it was worth noticing for further work that a solution with automatic code generation could be useful for the industry. Since there will be some problems expanding the program to a code generation tool it will probably be better to make a new tool and use our system as a background study. Code generation would require several changes to the prototype and would therefore be hard to implement.

5.2.2 QUANTITATIVE RESULTS

In this section the quantitative results for each question from the questionnaire and the information that can be extracted from these results are presented. The quantitative results showed that most of the professionals were happy with the tool, but disagreed among themselves on whether the tool could be helpful in software development in the company. Question 1 through question 4 and question 6 are the same questions as we used in the student test. Question 5 was based on the observation that some students did not see the difference between un-checking and deleting items. In question 7 we asked if the developers thought the tool could be useful in software development in their company.

Question 1 asked if it was intuitive to select Input items from the leftmost tree. Figure 21 shows that the employees found this easy, and some of them even found it intuitive. Even though we had few test participants, this was a positive result and removing the root item before this test made it easier for the user. Mean score for question 1 = 3.50.

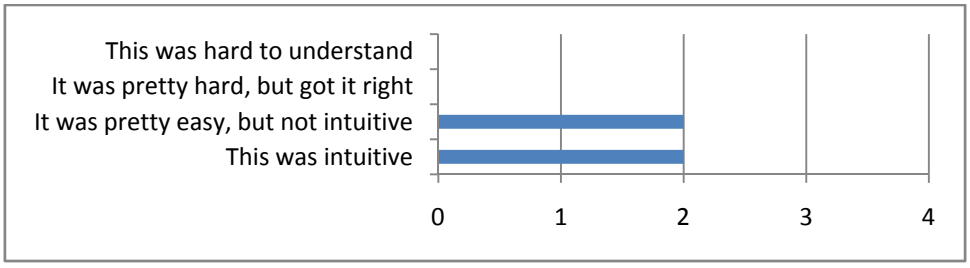

Figure 21: Question 1, ease of selecting Input items

The second question was if it was easy to add new Input items. Figure 22 shows that the employees found this easy, and 50% also found it intuitive. Even though we added a button that was available independently of which item was highlighted, 50% of the participants did not find this intuitive. Mean score for question 2 = 3.50.

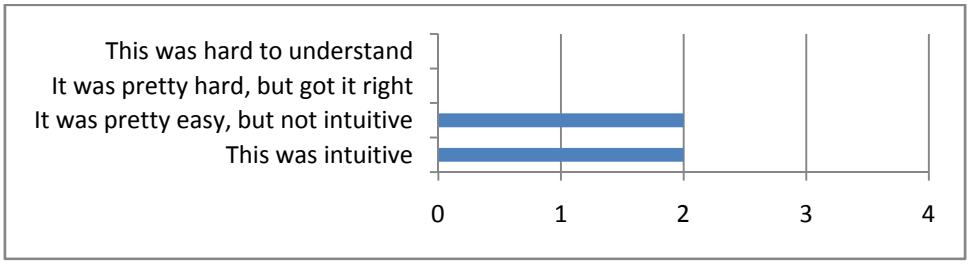

Figure 22: Question 2, ease of adding Input items

Figure 23 shows that 75% of the participants found it intuitive to select the child items under the Input items. This was the same result as we had for the students. The results were also similar for question 4 for the professionals and the students. Figure 24 shows the employees' results for question 4. Mean score for question 3 = 3.75. Mean score for Question 4 = 3.25

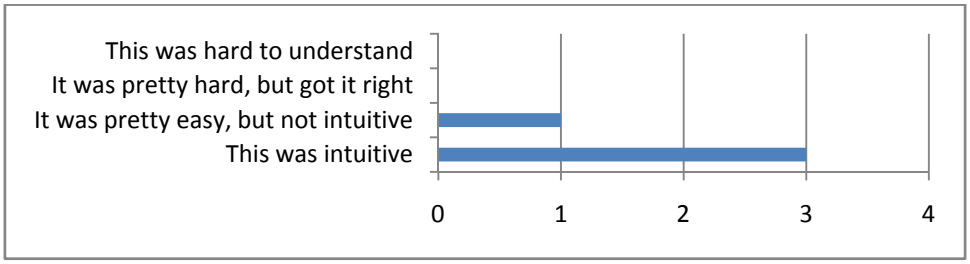

Figure 23: Question 3, ease of selecting child items under selected Input items

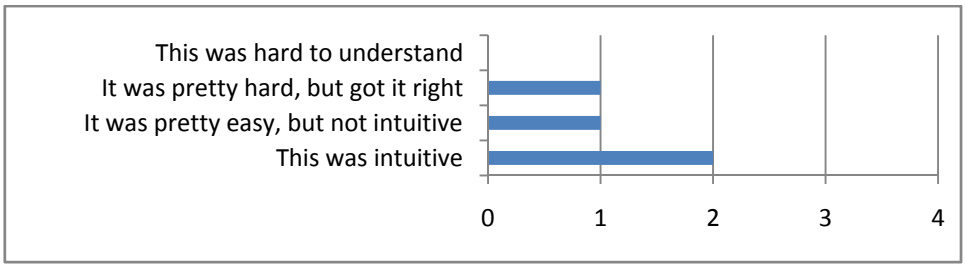

Figure 24: Question 4, ease of adding child items under Input items

Question 5 concerned the difference between un-checking and deleting. This was a problem for the students and we were worried that this difference was not explained well enough. Figure 25 shows that 50% of the employees found it intuitive. We also noticed that one participant found this hard to understand. It was hard to apply changes based on this since the observation was not supported by any qualitative feedback. The participant did not present this opinion during the interview session, so when considering the other opinions we did not find any changes that would make this more understandable. This question was not asked to the students so the mean score will not be evaluated further. Mean score for question 5 = 3.00

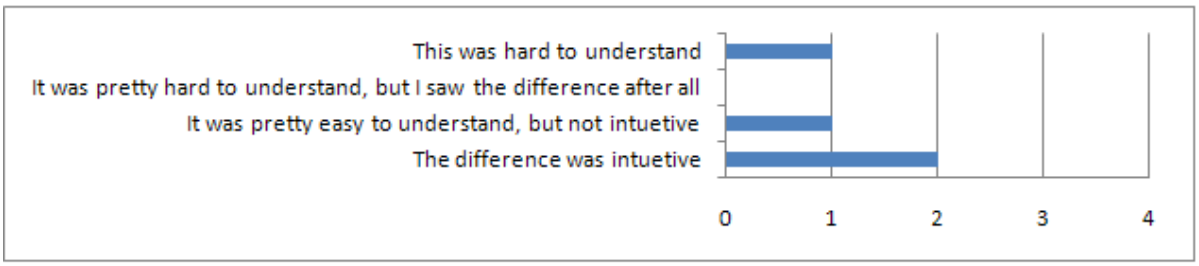

Figure 25: Question 5, difference between deleting and un-checking

Question 6 was the same question as the last question in the student experiment. This question was answered by only three of the four participants. The last participant did not understand the question, and we are not sure why the person did not ask for help. The result from this question is shown in Figure 26. The three that answered the question all had different opinions so we cannot conclude anything from this result. Mean score for question 6 = 2.67. The maximum score for this question would have been 3 if the last participant had the most positive alternative when answering this question.

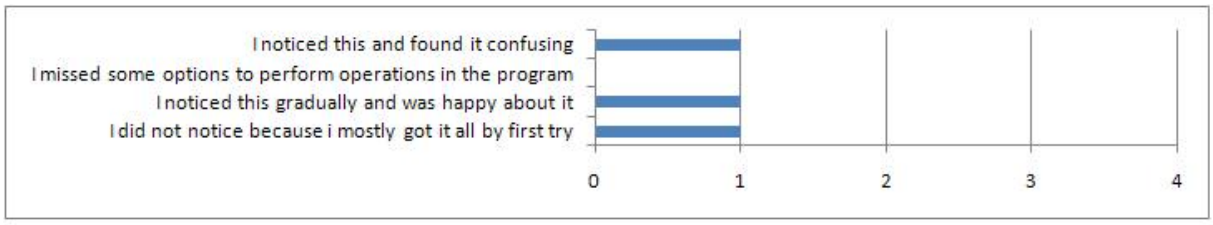

Figure 26: Question 6, noticing the redundant options

The last question was if they thought the tool could be a useful support for their software development process. One of the participants did not think that this tool could be useful. One thought that the tool needed some modifications to be useful for the company. The last two saw more potential in the tool and thought that it could be interesting to use. The results are shown in Figure 27. Mean score for question 7 = 2.25. This score was not used in the statistics since this question was not asked to the students. When considering Research question 2 shown in *chapter 4 - Experiment execution*, 50% of the developers said that the program probably could add value to the development process. This was considered a good result since the participants were critical to the tool in advance of the experiment.

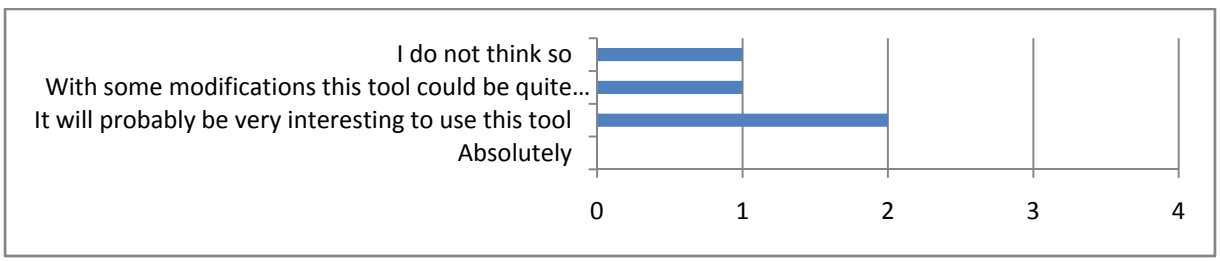

Figure 27: Question 7, whether the tool would be a useful to the company

Question 7 did not regard learning to use the program. Two answers from the developers implied that they did not understand the function in the program. In total there were 23 answers from the professionals. This means that 91% of the answers imply that the function was understood. This result satisfied requirement NF2. On the other hand we had four participants and two of them answered that some of the functions was hard to understand. This means that only 50% of the professionals learned to use all the main functions within an hour use. This would thus not satisfy the requirement of 90% of the users should learn the main functions within one hour of use. Research question 1 shown in *chapter 4 - Experiment execution* asks if the program was easy to use. Only 9 % of the answers were of the least positive character, which means that 91% of the answers supported RQ1, which was a great result. The chart shown in Figure 28 summarizes the results from the second test session.

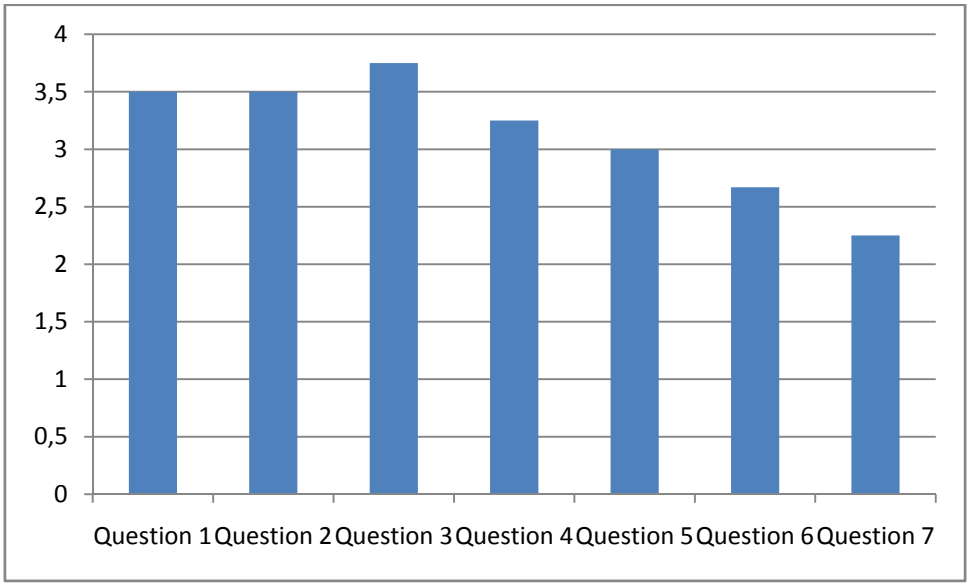

Figure 28: Mean score for each question asked to the professionals

5.3 COMPARING RESULTS FROM THE TWO EXPERIMENTS

In this section the results from the student test and the professional test are compared by using statistical tests. This was based on the mean score of each question, and only on the questions that were given to both groups.

5.3.1 SIGN TEST

A sign test is a simple statistical test that compares two objects regarding questions and sets sign to indicate which object that has the highest answer score. Our sign test checked which of the two test groups that was most satisfied with our tool. We chose to use a '+' sign if the students were most positive and a '-' if the professionals were most positive. The results are shown in Table 3.

Table 3: Sign test for students and professionals

Question	Students mean score	Professionals mean score	+/-
1	3.27	**3.50**	**-**
2	**3.82**	3.50	**+**
3	3.73	**3.75**	**-**
4	**3.64**	3.25	**+**
5	**3.36**	2.67 *	**+**

*Even with a maximum score of 3 it was the students that were most positive. This means that the sign test would have outputted "+" even if the last professional had answered the most positive alternative.

In Table 3 it is an equal number of "+" and "-", which indicate that both groups were equally satisfied. The sign test is, however, rather coarse and we therefore also performed a paired t-test. The mean score show that the students were more satisfied than the professionals in three out of five questions. In question 3 it was almost a tie score and the professionals were more positive than the students on the question that had the most influence on our choice of improvements, namely question 1. This means that the improvements made after the student test seemed to increase the usability of the tool.

5.3.2 PAIRED T-TEST

The paired t-test is a variant of the student's t-test. We used a paired t-test since we had two different response groups to the same objects - questions. In our case these two treatments were groups of people with different experience. Our questions, i.e. the objects in the paired t-test, differed in complexity.

The first hypothesis h_0 was that the difference between the two groups was zero. The test was performed by first calculating a mean score for both the students and the professionals. This was done by summing up all the scores and dividing by the number of scores from *section 5.1.1 - Quantitative results* and *section 5.2.2 - Quantitative results* for the students and the professionals respectively. In addition we estimated the variance for each group. This was calculated using the formula shown below.

$$S_d = \sqrt{\frac{\sum_{i=1}^{n}(d_i - \bar{d})^2}{n-1}}$$

Since we were estimating the variance, the number of degrees of freedom was the divisor in the formula. We use the number of observations in the equation for calculating t_0. This equation is shown below.

$$t_0 = \frac{\bar{d}}{S_d/(\sqrt{n})}$$

Our degree of freedom was 4 and we chose a significant level of 95%. The critical t values were retrieved by looking in the t-distribution table. For one-tailed our α-value was 0.05. This meant that the P(T<=t) had to be lower than 0.05 for the results to be statistically significant. On the other hand it was an 11% probability that this was a random result and 89% probability that this result was because the two groups were different, thus the difference was not statistically significant and hence h_0 cannot be rejected. This result would have remained the same independently of the last participant's choice on the professional's question 6. Our difference mean was calculated to 0.23 and a 95% confidence interval from this difference was [-0.2174, 0.6774]. All the results are shown in Table 4.

Table 4: Results from the paired t-test

	Students	Professionals	Difference
Mean	3.564	3.334	0.23
Variance	0.05673	0.16903	0.12985
Observations	5	5	
Hypothesized mean difference	0		
Degrees of freedom	4		
t_0	1.427223		
P(T<=t) one tail	0.11335		
t Critical one-tail	2.132		
P(T<=t) two tail	0,2267		
t Critical two-tail	2,776		

5.3.3 DISCUSSION OF SIMILARITIES

When comparing the results from the tests summed up in *section 5.1 - User interface test performed on students* and *section 5.2 - Functionality test performed on professionals* some similarities was revealed. The sign test showed that when the students' score were the largest they were in average 0.43 above the professionals' and in average 0.13 below when not. This implied that the students overall found the program more intuitive than the professionals did. There may be several reasons for this. One possible reason was that students answered the intuitive option because they did not want to admit that they did not understand the tool. Another possible reason was that professionals interpreted "intuitive" differently than the students did. In our opinion there was no reason to believe that any of the students lied when answering. On the other hand we suspected that the professionals

were more critical than the students. The mean scores for both students and professionals are shown in Figure 29. Question 5 and question 7 were not asked to the students.

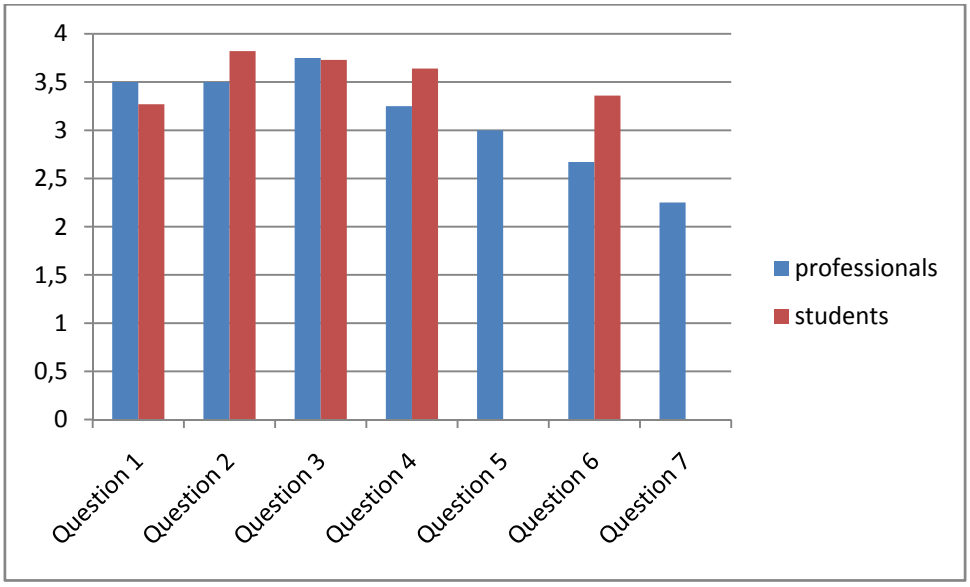

Figure 29: Mean score for students and professionals

It was important to remember that some modifications were made to the program between the tests. This especially applied to the question 1 where the professionals found the program more intuitive than the students. The improvements might be the reason why question 1 was the only where the professionals had the significant highest score. We were aware of the fact that it was preferable to use the same system when applying statistical methods, but this inconvenience was disregarded since improving the tool was more important. The t-test showed that it was 11% probability that the two groups were different. This was not statistically significant but it was still a high probability. The results of the experiments indicate that the students found the program more intuitive than the professionals. Since there were fewer participants in the business experiment, each professional influenced the mean score more than each student did. One of the participants from the company did not understand one question and the percentage might have been affected of this. This participant was the most critical of all the participants. Based on the results and our own impressions we believe that there were differences between the students and the professionals even though there was no statistical significance to support this claim.

PART III
IMPLEMENTATION

6 ARCHITECTURE

The architecture is the first part of the design phase, and is equivalent to high level *design* (Braude 2001). It is followed by *detailed design,* which is covered in the chapter 7 - Detailed design. (Bass, Clements et al. 2003) defines the software architecture for a system as *the structure or structures of the system, which comprise elements, the externally visible properties of those elements, and the relationship among them*.

6.1 STAKEHOLDERS

The developers for this project were the authors of this report. We were responsible for all development, testing and maintenance of the project. All future developers also fall into this category. Our teaching supervisor was Tor Stålhane, professor at department of Computer and Information Science, Norwegian University of Science and Technology in Trondheim. As part of the evaluation of the tool developed we ran two tests, and the persons participating in these tests are also, in some sense, stakeholders for the architecture.

6.2 VIEWS

According to (Bass, Clements et al. 2003), an architectural view is *a representation of a coherent set of architectural elements, as written by and read by system stakeholders*. Having several views can help represent the architecture in different ways for different stakeholders. We selected however only one view, the Logical view, a part of the 4+1 view model of software architecture by (Kruchten 1995). The Logical view primarily supports the functional requirements, and was implemented by the use of class diagrams. A class diagram shows a set of classes, with their relations; inheritance, usage, association, and so forth. The logical view for our architecture is shown in *section 6.4 - Our architecture*.

6.3 MODEL-VIEW-CONTROLLERS

Model-View-Controllers (MVC) is a well-known architectural pattern. A*n architectural pattern is a description of element and relation types together with a set of constraints on how they may be used (Bass, Clements et al. 2003).* In MVC the *Model* represents the knowledge in the system, the *View* is the visual representation of the model and the *Controller* is the connection between the *Model* and the *View*. It provides the user with output by using the views, and allows the user to give input through menus and other controls (Reenskaug 1979). A schematic of MVC is shown in Figure 30.

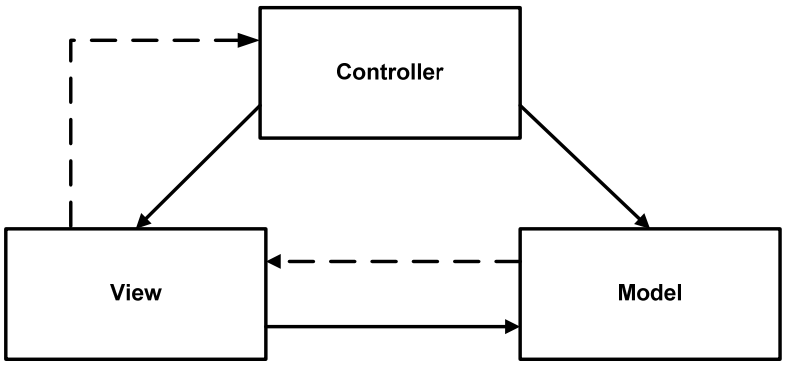

Figure 30: Model-View-Controllers[2]

6.4 OUR ARCHITECTURE

MVC is a pattern that was well suited for the project. No thorough process of selecting the pattern was performed, as we were united in the choice of pattern from the beginning. The project is divided into three packages; Model, View and Controller, these are described below. The architecture is shown in Figure 31.

The *Model* package represents the data, which are the Input items with their child items. We also implemented a class with methods to load the data into suitable classes and store changes to the model. It contains one class with tools to load and save the data, and classes to represent the items.

The *Controllers* package handles the logic between the View and the Model. It is a single class, with static methods for saving, loading and editing the Model.

The *View* package contains one class, namely a window form that presents the content of the Model using appropriate user controls.

[2] http://en.wikipedia.org/wiki/Model-view-controller

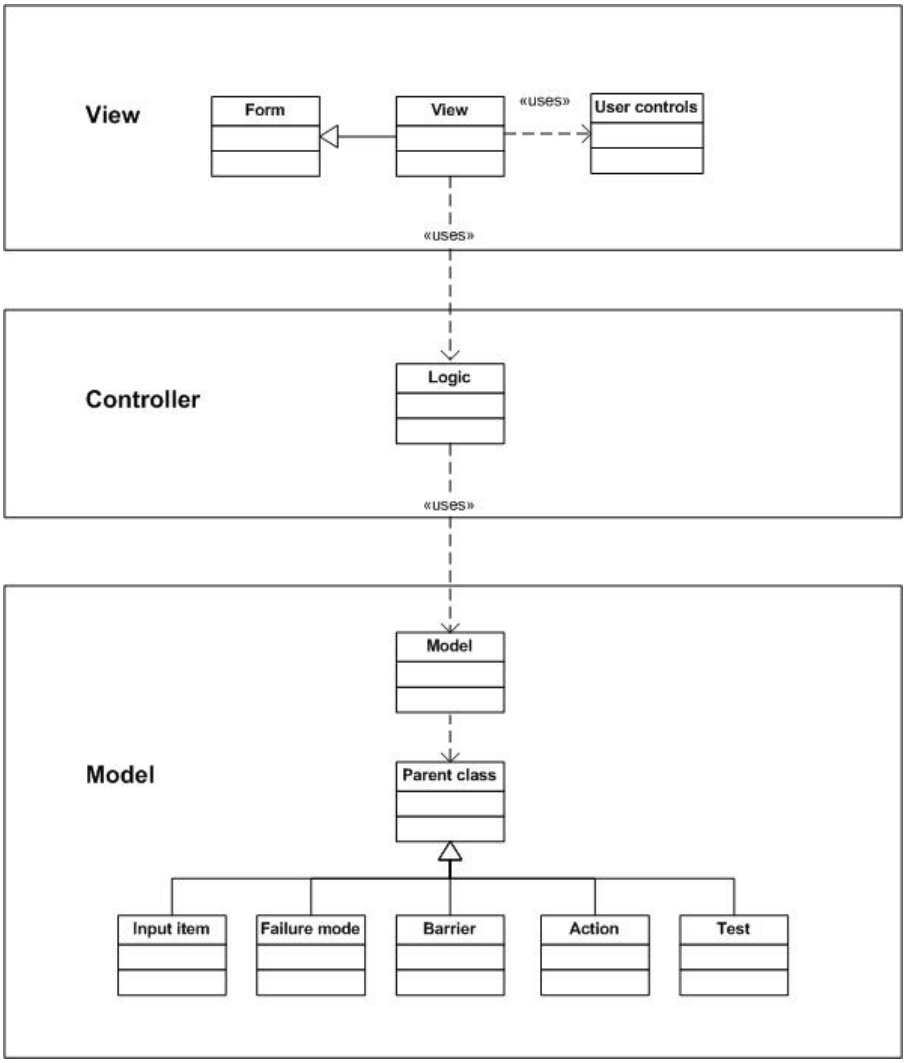

Figure 31: The architecture of our project

7 DETAILED DESIGN

This chapter covers the detailed design. It succeeds the architecture, and the goal is to prepare the project for the implementation by providing an implementation blueprint for the programmer.

7.1 CHOICE OF TECHNOLOGY

Since some design decisions depended on the technology used, the choice of technology was done as the first part of the design. Since we were making a prototype, we based our choice mainly on the GUI programming abilities of the technology, as this would be the most important part of the project. The two options are shown below, along with the decision.

7.1.1 JAVA

Java was the only programming language which we had learned in-depth. Still, we were novices in GUI programming in Java. Java does not offer a GUI editor, though add-ons to for instance Eclipse[3] can be used. To run a Java program, the Java Runtime Environment must be installed on the computer in use, which was considered a limitation. CVS[4] or SVN[5] were our options for version control with Java.

7.1.2 C#

Based upon Microsoft's .NET platform, C# is developed using the programming tool Visual Studio. Through *MSDN Academic Alliance*[6] we could download and use *Visual Studio 2008 Professional Edition* for free. The syntax in C# is similar to the syntax of Java, which meant that it would be easy adaptable for us. Skjervold, who was in charge of the development, had been working with C# previously while Haga was new to it. Working with GUI in Visual Studio is easy and intuitive, based on drag and drop functionality. It is also easy to create an EXE-file to use during testing. *Visual SourceSafe 6.0d*, also available through *MSDN Academic Alliance,* would be used for version control.

---

[3] http://www.eclipse.org/

[4] http://www.nongnu.org/cvs/

[5] http://subversion.tigris.org/

[6] http://msdn60.e-academy.com/NO_700027

7.1.3 OUR CHOICE

We chose to use C# for this project. The superiority in GUI programming was the most important factor. We did not perform a thorough analysis of many technologies as the most important factor was that we had to be satisfied with the use of the technology. This means that the programming languages that we were not comfortable with were not taken into consideration. Thus, our own opinion of the most suitable technology was heavily weighted and after some initial testing of both Java and C# the choice was made. In retrospect we have no regrets as C# proved to be well suited for our project.

7.2 GRAPHICAL USER INTERFACE

The first part of the design was the graphical user interface (GUI). The usability of our program had the highest priority, and hence the layout of the GUI was our first design decision. The GUI was also important for the remaining design process.

7.2.1 DESIGN

The first design of the GUI was done by sketching on paper. This took about two days and included studying the relevant user controls in C#. The user controls and their functionalities were necessary to understand how to make good GUI decisions during the sketching. The sketching resulted in Figure 32.

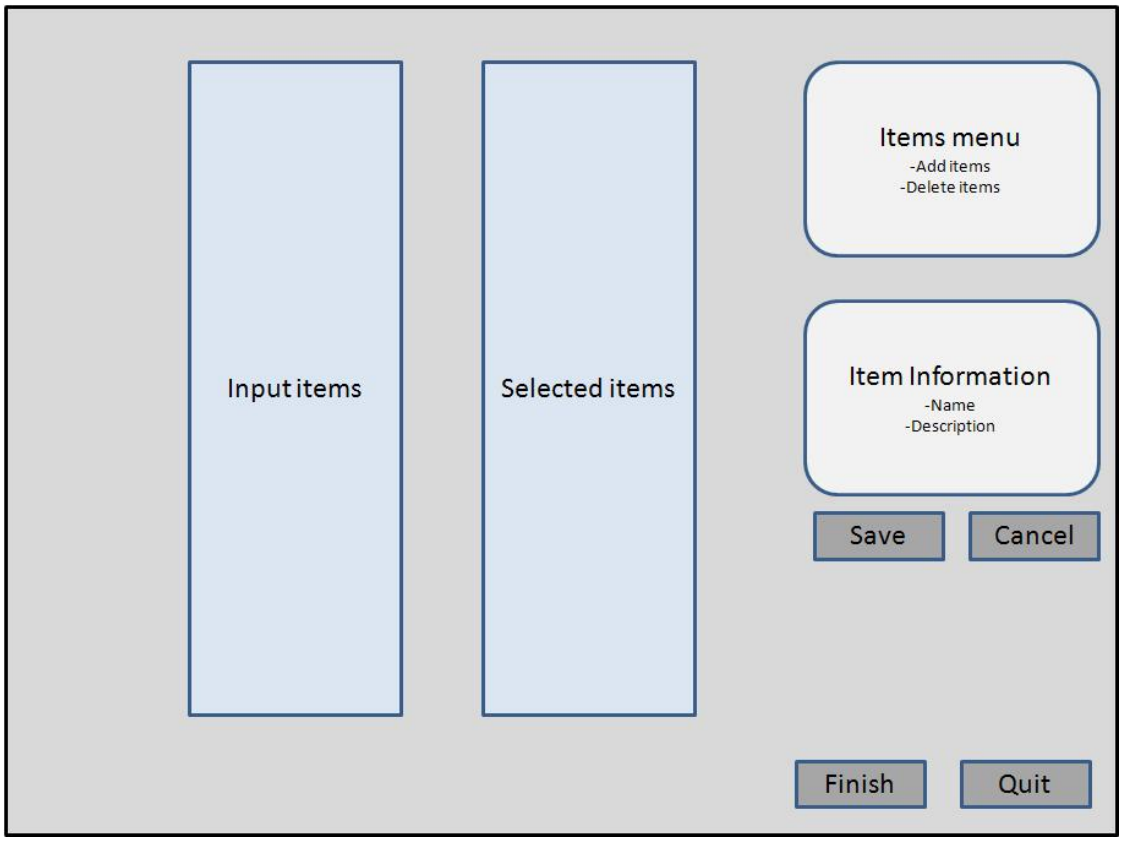

Figure 32: The early sketch of the GUI

7.2.2 USER CONTROLS

The two boxes *Input items* and *Selected items* in Figure 32 symbolize hierarchical trees, like the folder view in Microsoft Windows. Hierarchical trees are common user controls[7] for programming platforms, and were a good solution for presenting our data model as it is hierarchical. In C# these controls are named *TreeViews*[8]. The reason for using two trees was that *Input items* should contain all the Input items in the database, while *Selected items* should only contain the Input items that the user find relevant for his program. In addition, *Selected items* show all underlying items to the Input items. The *items menu* provides operations on the chosen item in one of the trees, and is a set of buttons or other appropriate user controls. *Add items* and *Delete items* are the two most intuitive actions, but other functionality can be added as the need arises. *Item information* presents the information of the highlighted item in the trees, using text boxes or similar user controls. *Save, Cancel, Finish* and *Quit* are all buttons with self-explanatory functionalities.

7.3 CLASSES

The next step was to create a detailed class diagram. Only the most important parts of the classes are included in the diagrams in Figure 33 through Figure 35, as it would overload the diagrams if every event, method and parameter was included.

7.3.1 MODEL

Developing a good data model to represent the items used in the tool was important. Our model was named **NodeModel**. The items used in the *TreeView* control are of the class *TreeNode*, so NodeModel was set to inherit from TreeNode. To store the model in between sessions a few options was considered: Database, XML and Serialization. A database was considered an exaggeration for this project, and given the hierarchical model it would be difficult to implement and parse the database. Storing the information in an XML file was a good and suitable option, and as good as Serialization, which was the solution that was used. Serialization is a process of saving an object onto a file or a similar medium[9], and was chosen since we were familiar with the method. C# has good and simple mechanics for this, and we had used it before. We thus chose to use this technology. In retrospect we learned that Serialization lead to some difficulties, for instance that a serialized file could not be reused if the structure of the code was changed. This meant that a new file had to be saved each time

[7] A User control allows the user to communicate with the system, for instance a Text box

[8] http://msdn2.microsoft.com/en-us/library/system.windows.controls.treeview.aspx

[9] http://en.wikipedia.org/wiki/Serialization

some important parts of the code were changed, which lead to additional work. Hence, XML would probably have been a slightly better and more code independent choice and should be considered for further development.

To allow NodeModel to be serialized it was set to implement the interface *ISerializable*. Also, NodeModel was set to be abstract. An abstract class cannot be initialized, but can be inherited from. The classes that inherit from an abstract class must implement all abstract methods in the parent class. Hence, they provide its children with a recipe for the methods they must implement. This was suitable for NodeModel, since there are not any instances of the class itself, only of its children.

For each item in our model, a class was created that inherited from NodeModel. These were **Input item, Failure mode, Barrier, Action** and **Test**. These classes implement more specific versions of the *AddChild* and *AddEmptyChild* methods which they override from the parent class NodeModel. The **Model** class is the link to the Controller package. It implements two methods: The *LoadModel* method retrieves (de-serializes) the data from a file onto the data model, and *SaveModel* stores (serializes) the data model objects to a file.

The detailed design for the Model package is shown in Figure 33.

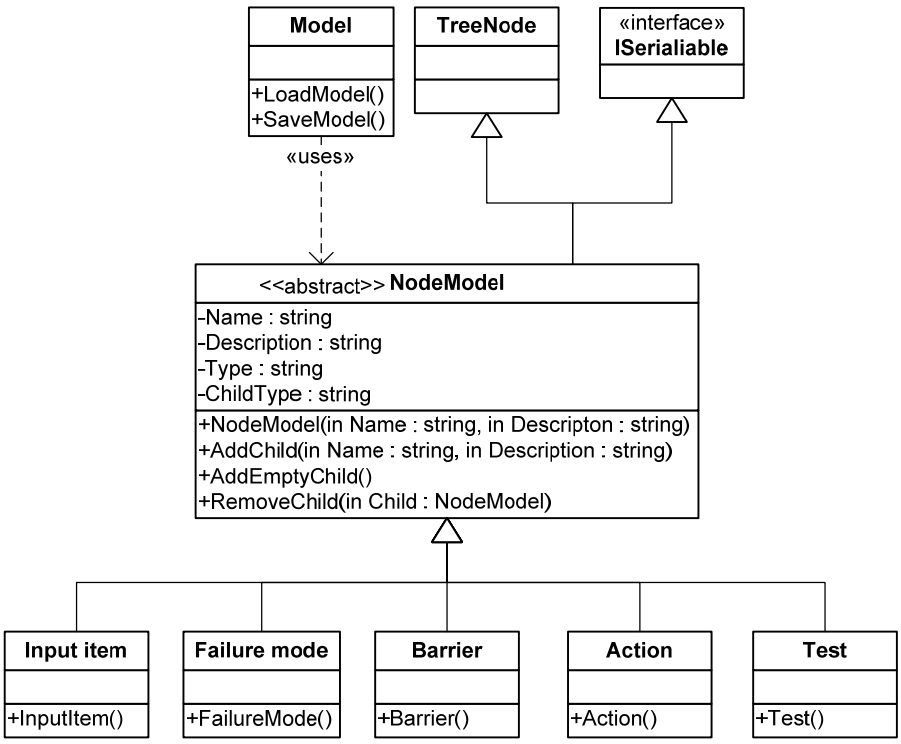

Figure 33: The detailed design of the Model package

7.3.2 CONTROLLER

The Controller package contains only one, static class **Logic**, shown in Figure 34**.** *ImageList* is a common list of images that can be used in the user controls. *TreeViewLeft* and *TreeViewRight* are the two TreeViews in the View. Having them in the Logic class makes it easier to perform tasks on them. The View package handles the presentation of the trees to the user. The first time the trees are accessed they are loaded from the serialized file.

AddChildItem adds a child item to the input parameter *Parent*. It uses the method *Model.NodeModel.AddEmptyChild.* Adding an Input item differs from adding other items, as this is the top item in the TreeView. There is a special method for this, namely *AddInputItem*. *DeleteItem* deletes the input parameter *Node*, and *PrintResults* shows the results of the user session. The results are presented as a graphical representation of the items in *TreeViewRight*. SaveToModel save the changes made to the nodes, by using *Model.SaveModel.*

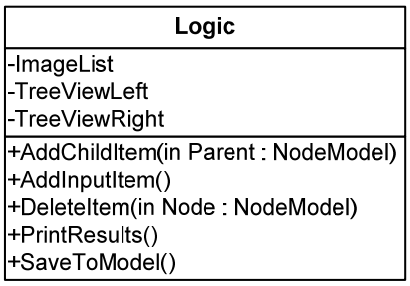

Figure 34: The detailed design of the Controller package

7.3.3 VIEW

The View package has only the class **MainForm**, shown in Figure 35. It inherits from the C# form class *System.Windows.Form*, and is used to present the information in the Model.

CheckForUnsavedItems trigger when a new item is highlighted. It checks if the previous item was changed without being saved. If so, the user is prompted with a question whether he wants to save or cancel these changes, and the chosen action is made. *DeleteItem* ask the user if he is sure about his decision to delete the highlighted item, and remind him that this is irreversible. The Logic class then deletes the item if the user confirmed the deletion.

View contains many events thrown by the user controls, which again call upon the Logic class to perform the requested operations. Since there are many events, only the most important are mentioned here. Most of the events are simple as they only do one call to the Logic class. For instance *AddInput*Item only use the corresponding Logic method *Logic.AddInputItem* to add an Input item to a tree. Every button has an event for clicking the button. The textboxes has *TextChanged* events to determine when the name or description for an item is changed, along with a *KeyUp* event to determine when the keyboard button

enter is pushed. For the TreeViews there are several events. They handle both GUI specific actions, as moving items between the two trees (as mentioned before, this does not affect the Logic and Model), and Logic actions, as adding or removing items.

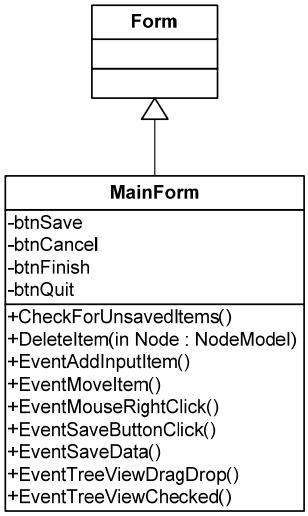

Figure 35: The detailed design of the View package

8 IMPLEMENTATION

This chapter describes the implementation of the tool that was designed in chapter 7 - Detailed design, along with testing of the requirements. A user manual for the tool is also added as part of chapter 8.

8.1 CLASSES

This section shows the final classes of the implementation, with comments on implementation choices made. The details of the classes that were explained in the design are not elaborated here.

8.1.1 MODEL

The classes for the Model package are shown in Figure 36, and are based on the design in Figure 33. The fields are left out, since they are not important for understanding the implementation, and are so many that it would overload the diagram. The class *Model* was renamed to *Tools* to avoid confusion between the class and the package. In *Tools*, the methods *LoadTree* and *SaveTree* are equivalent with *LoadModel* and *SaveModel* in the design. *LoadTree* creates an empty TreeView, fills it with the items from the serialized file, and returns it. *SaveTree* saves the content of both TreeViews to a file; hence it does not differentiate on which TreeViews the items are in. For the *NodeModel* class, there are some additional methods in the implementation compared to the design. *ShowChildren* is used to show or hide the children of an Input item, since they should not be shown in the left tree and should be shown in the right tree. This method uses the field *_children*, which is discussed in *section 8.4 - Discussion.*

UpdateNode uses ShowChildren to update an item when a new child has been added. *CheckChildren* sets the status of all children of an item to checked or un-checked, based on the Boolean input parameter. This method is used for instance when an item is un-checked; all the children should then also be un-checked.

GetObjectData is a method inherited from *ISerializable* interface, and defines the fields that are included in the serialized file. There is also a constructor for the serialization, where the fields are de-serialized.

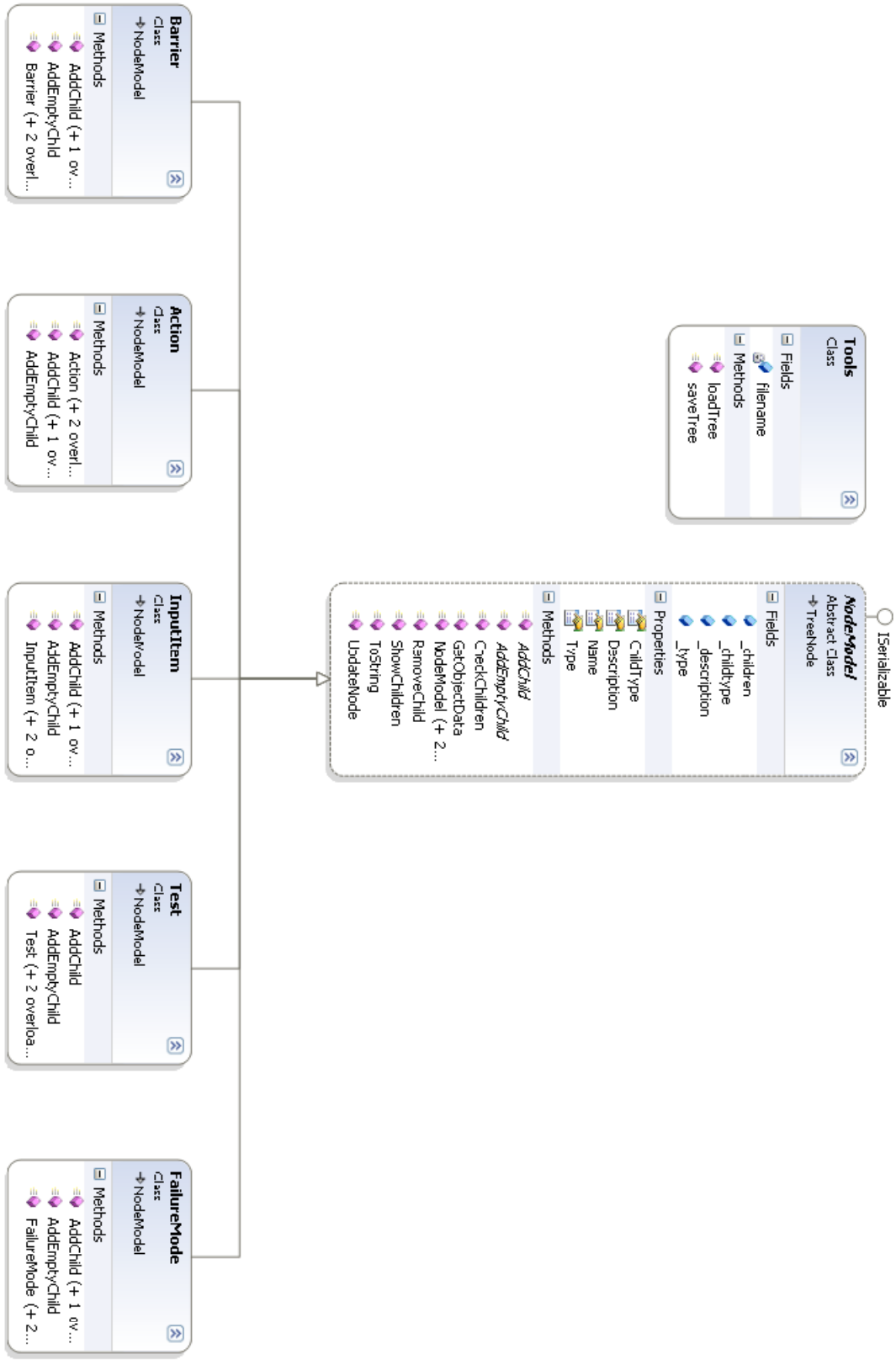

Figure 36: The class diagram for the Model package

8.1.2 VIEW

The class diagram for the View package is shown in Figure 37. The difference between the designed class diagram in Figure 35 and the final class diagram is noticeable, as only the most important events and methods were included in the design. This was because many of the events and methods needed for the implementation were difficult to indentify during the design.

The *EventBtnCancelChange_Click* event and other similar button events are triggered when the buttons are clicked, and most of them have simple calls to the Logic. Some of the user controls use the same event, for instance the event Event*DeleteItem* is used by the delete button in the upper right menu, the right click menu and when pushing the delete button on the keyboard. All the *EventTree* events are related to the TreeViews, and these events have intuitive behavior according to their names. For instance *EventTreeViewLeft_DoubleClick* is fired when an item in the leftmost tree is double-clicked. The *SelectedNode* property is the item that is highlighted in the program in either of the TreeViews, as only one node in the program can be highlighted at any time. The *Set* methods are used to configure the buttons and text boxes depending on the highlighted item in the TreeViews, while *SaveData* tells the Logic to save changes made to the Model.

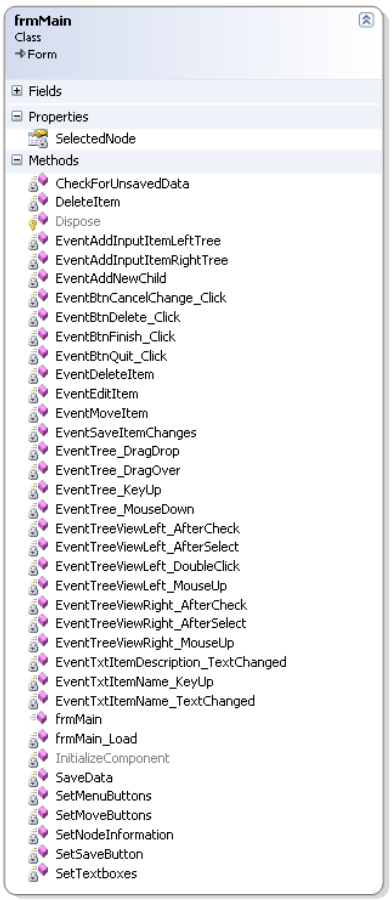

Figure 37: The class diagram for the View package

8.1.3 CONTROLLERS

The class diagram for the Controllers package is shown in Figure 38. The implementation of the *Logic* class was close to the design; the only difference was the implementation of two *AddInputItem* methods, one for each tree. This was done to ease the use of the methods for the View. The XML methods are used by the method *PrintResults*, the same goes for the fields *xDoc*, *htmlLoc* and *xmlLoc*.

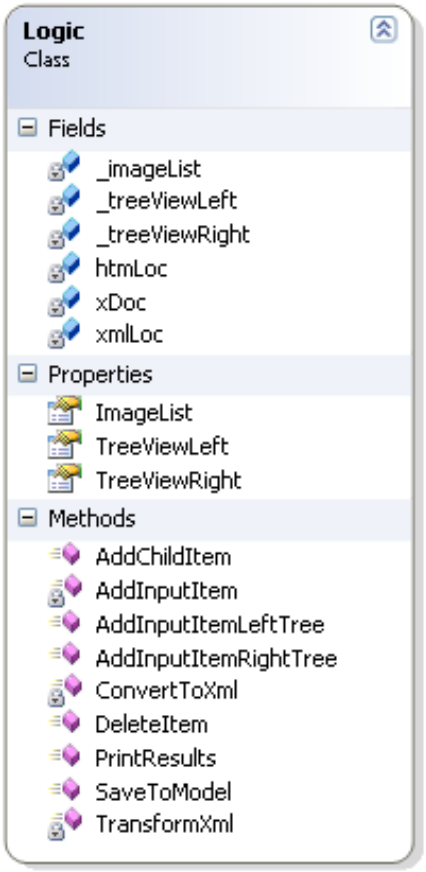

Figure 38: The class diagram for the Controllers package

8.2 GRAPHICAL USER INTERFACE

The final GUI is shown in Figure 39. On the left hand side there is a walkthrough for the tool. For a regular user it is unnecessary in the long run, and overlapping with the user manual in *section 8.4 - User manual.* Our experiences were that the walkthrough was not used much by the testers as they started using the program immediately in a way they found intuitive. It was kept nevertheless, as it is good to have for a user that is stuck.

The leftmost tree *Input items database* contains every Input items in the database. The rightmost tree *Selected items* contains the Input items which the user has selected as relevant for his design. The Input items can be moved between the trees in several ways:

- Checking and un-checking the checkbox on the Input item
- Double-click the Input item
- Right click and select *Move item to Selection tree*
- Use the *Move* buttons beneath the trees
- Click the arrows "*>>*" and "*<<*" between the trees
- Drag and drop the Input items

Right click move and double-click only works when the Input items are in the leftmost tree. Since they are more often moved from left to right than from right to left, there are more options for the left to right scenario. The user only moves an Input item from right to left when he regrets choosing it for his requirements. The checkboxes on the items are used to select the items that the user finds relevant for his design. When an Input item in the rightmost tree is un-checked it is moved back to the leftmost tree. The child items of an Input item can be un-checked if the user finds the Input item relevant, but not the child item.

When an item is highlighted the tool provides several options. The information of the item is shown in the group box *Item information*; the description of the item is also shown as tool tip[10] when the mouse cursor is hovered above the item. Changes to the item can be made in *Item information*, and saved either by using the *Save* button or by clicking the keyboard button Enter when you are done. The *Cancel* button removes all unsaved changes made to an item. If changes are made, and the user attempts to select another item without saving, he is prompted on whether he wants to save or discard the changes made.

The group box *Menu* in the upper right corner is customized to the highlighted item. The upper button *Add Input item* is always enabled; it adds a new Input item to the leftmost

[10] A small box that appears with information regarding the item being that is being hovered over, see example in Figure 43.

tree. The middle button changes according to the highlighted item. When an Input item in the left tree is highlighted, the button is disabled since you need to move the Input items to the right tree to add child items to it. When an item in the right tree is highlighted, the button is called *Add <child item>*, for instance when a Failure mode is highlighted the button is called *Add Barrier*. The last button in the menu; *Delete item,* is enabled whenever an item in any of the trees is highlighted. Deleting an item can also be done using the keyboard button *Delete*.

The lower group box *Model* shows the data model with the relationships between the items in the data model. This group box has no other functionality than to show the data model to the user. The *Quit* button shuts down the program, while the *Finish* button saves the items in the right tree to an html file and launches it. We experienced some problems with the launching of the htm file; launching a program can be prohibited by some systems. Therefore, an information box pops up describing that the html has been created and that it should be launched in a browser window. The user can then launch it if the system does not do so.

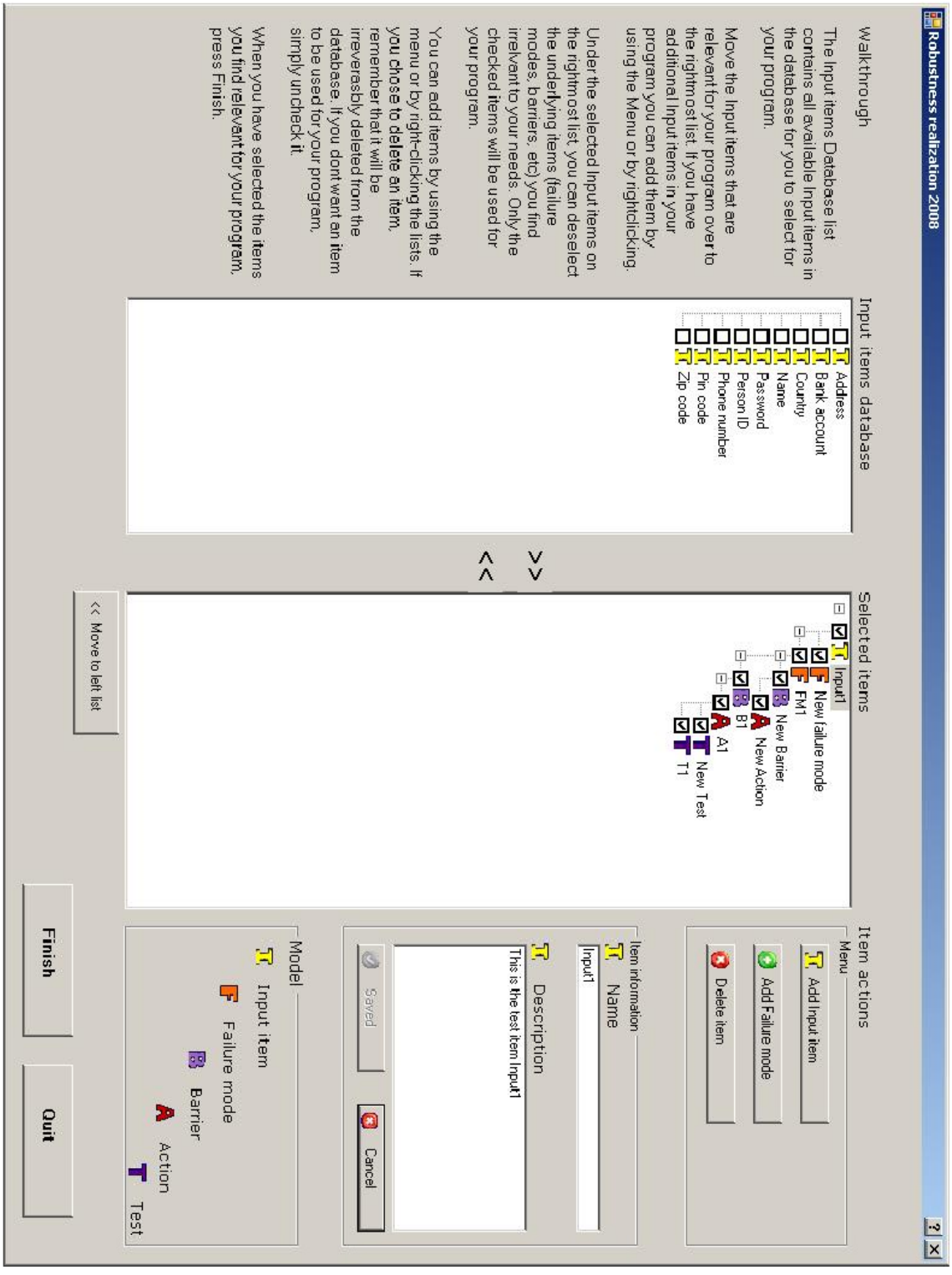

Figure 39: The final GUI

8.3 TESTING

The testing of the tool was performed by the authors. The purpose of this activity was to confirm the realization of the requirements in *chapter 2 - Requirements specification*, and to check for errors in the implementation. The tests are functional simple tests, not following a standard as for instance IEEE-829 (IEEE 1998). This standard was considered, but found overkill for this implementation. All errors found in the tests were corrected immediately. The tests of the requirements are shown in Table 5 through Table 9.

Requirements F7 through F10 were not tested, as these requirements could be confirmed through observation of our model in Figure 36, and thus did not need functional tests. These requirements stated that the model should implement a one-to-many relationship for the items, meaning that each item can have:

1. Only one parent
2. Multiple children.

These requirements were realized through the inheritance of the TreeNode class, which has these requirements implemented.

Table 5: Test 1

Test ID	Test 1	
Test name	Item presentation	
Requirements affected	F1, F2, F4	
Test description	**Expected results**	**Results**
Start the program in an empty folder. Check that the Input items are presented in the left tree after selecting OK to the database question.	The Input items are shown in the left tree.	OK
Check the Input item *Name* in the left tree.	*Name* is moved to the right tree, Failure modes and other items are shown beneath it	OK
Select *Name* and some of the child items.	The name and description of the items should be displayed	OK

Table 6: Test 2

Test ID	Test 2	
Test name	Add Input items	
Requirements affected	F3	
Test description	**Expected results**	**Results**
Add a new Input item to the leftmost tree by right clicking and selecting *Add Input item* from the menu.	An Input item is added to the left tree.	OK
Add a new Input item to the rightmost tree by right clicking and selecting *Add Input item* from the menu.	An Input item is added to the right tree.	OK
Add a new Input item to the leftmost tree by using the menu button *Add Input item* in the upper right corner.	An Input item is added to the right tree.	OK

Table 7: Test 3

Test ID	Test 3	
Test name	Add child items	
Requirements affected	F5	
Test description	**Expected results**	**Results**
Add a new Input item to the rightmost tree, name it *Input1*. Right click Input1 and chose *Add Failure mode* from the menu.	A Failure mode called *New failure mode* is added to Input1	OK
Highlight Input1 in the right tree. Click the button *Add Failure mode* in the upper right menu. Name it *FM1*.	A Failure mode called *New failure mode* is added to Input1. The name changes to FM1 when you save.	OK
Right click FM1 and chose *Add Barrier* from the menu.	A Barrier called *New Barrier* is added to FM1	OK
Mark FM1 in the right tree. Click the button *Add Barrier* in the upper right menu. Name it *B1*.	A Barrier called *New Barrier* is added to FM1. The name changes to B1 when you save.	OK
Right click B1 and chose *Add Action* from the menu.	An Action called *New Action* is added to B1	OK
Mark B1 in the right tree. Click the button *Add Action* in the upper right menu. Name it *A1*.	An Action called *New Action* is added to B1. The name changes to A1 when you save.	OK
Right click A1 and chose *Add Test* from the menu.	A Test called *New Test* is added to A1	OK
Mark A1 in the right tree. Click the button *Add Test* in the upper right menu. Name it *T1*.	A Test called *New Test* is added to A1. The name changes to T1 when you save.	OK

Table 8: Test 4

Test ID	Test 4	
Test name	Delete item	
Requirements affected	F6	
Test description	**Expected results**	**Results**
Use Input1 from Test 3 in Table 7, or redo the tasks there to create it. Select Test T1 under Input1. Click Delete.	A window pops up asking if you are sure of your choice to delete the item	OK
Click *Cancel*	The window disappears, and the item remains in the tree	OK
Use the upper right corner button *Delete item* on T1. Click *OK* on the window when it appears	The window disappears and T1 is removed.	OK
Quit the program, and start it from the same location.	T1 is not present in the trees.	OK
Comments		
This test tests some of the program functionality not mentioned in the requirements, but elements that should be tested nevertheless.		

Table 9: Test 5

Test ID	Test 5	
Test name	Results	
Requirements affected	F11	
Test description	**Expected results**	**Results**
Use Input1 from Test 3 in Table 7, or redo the tasks there to create it. Make sure that Input1 is in the right tree. Click the button *Finish*.	Your browser pops up showing the left tree. Results.htm is put in the program folder.	OK
Comments		
This test tests some of the program functionality not mentioned in the requirements, but elements that should be tested nevertheless.		

8.4 DISCUSSION

8.4.1 THE CHILDREN FIELD

The *_children* field in the NodeModel was an implementation decision which was regretted. Since the child items of an Input item sometimes are shown and sometimes not, a list of the child items was needed in addition to the field *Nodes* inherited from TreeNode. *Nodes* contain all child nodes that are shown beneath a node in a tree, and at the time of implementation we could not find a way to hide the children when they were in the left tree. Therefore, *_children* was implemented, and when the Input items were moved between the trees, the items were moved in and out of *Nodes* from *_children*. This solution was not ideal, and there are probably better solutions to this problem. When this solution was discovered, it was decided to not change it for two reasons:

1. The quality of the code was not a main priority for this project
2. For any further work with this project, as described in *chapter 9.2 - Further work*, we recommended a new design and implementation. Thus, spending time on fixing poor implementation decisions such as this was not prioritized in the latter part of the project

We have no suggestions for alternative solutions.

8.4.2 USING THE RESULTS

The results from our system can be used by developers when handling robustness issues during designing and implementation of the software. Figure 40 shows how the user can first use our system and then use the results to design and implement the software.

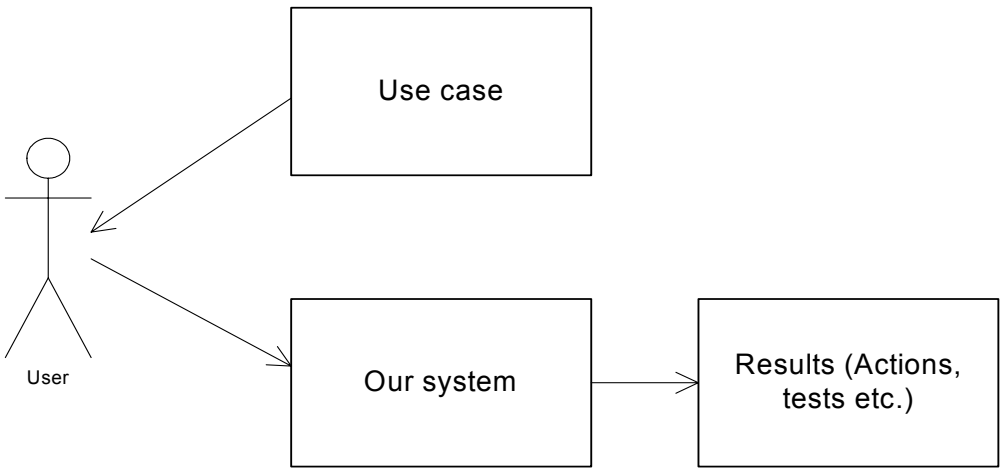

Figure 40: The results can be used in the user's software development process

When the developer has chosen all his Input items and child items, the *Finish button* in the program shall be clicked. The developer is shown the results as an overview of Failure modes and information of how to prevent these Failure modes, i.e. Barriers and Actions. The Tests will help the developer to verify that the Failure modes preventions are correctly implemented.

Figure 41 shows the htm result page. This example is a continuation of the example in Appendix E.1.6 – Data structure. The user in this example has chosen the Input items *Name* and *Zip code*. This result page gives the user an overview over which Failure modes, Barriers and Actions that should be considered when designing and implementing the software. For this example we have chosen the Failure modes *Illegal characters* and *Empty name* for *Name*. The developer is now aware of these threats in an earlier phase than before, and this allows him to improve the design and implementation cheaper than a correction in a later phase. The same goes for the Failure modes *Wrong input size* and *Illegal characters* for the Input item *Zip code*. In addition to Failure modes, Barriers and Actions, the developer is also shown Test suggestions. If test-driven development is preferred, our results make it easy to develop the Tests extracted from the tool before writing the code. Even though Tests are not developed early it is useful to be aware of these in an early phase.

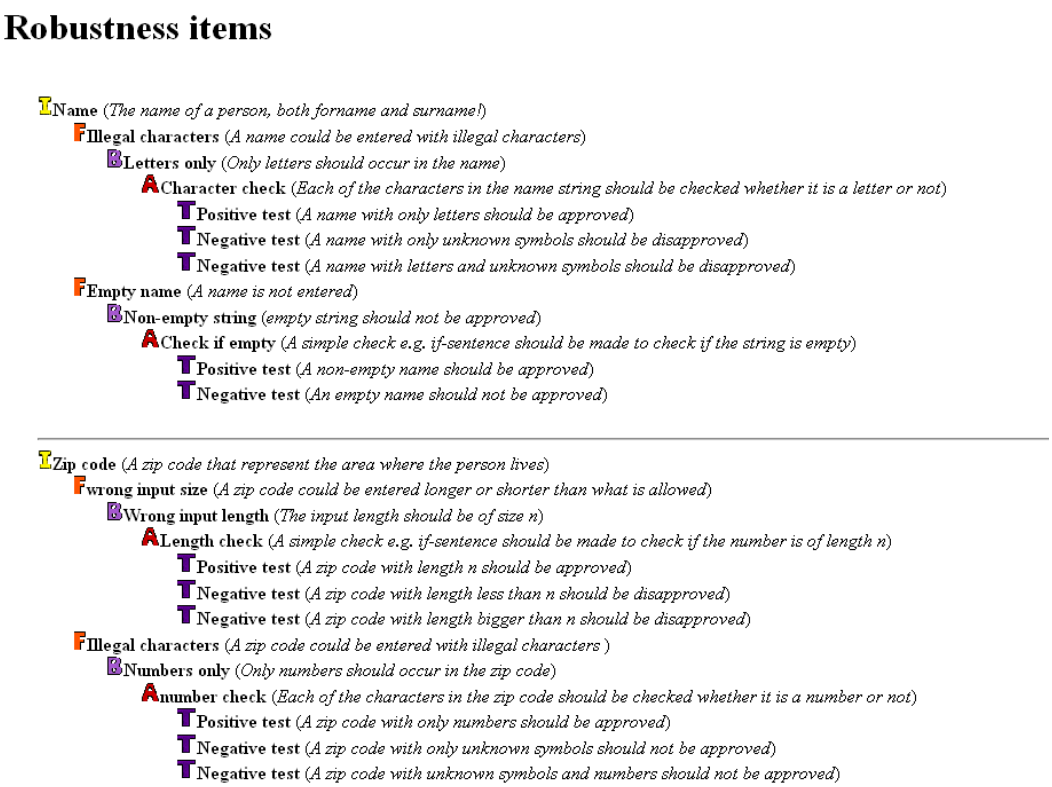

Figure 41: Result page from our program

The results helps the developer handle robustness aspects related to the given use case. For each Input item the developer has at least one Failure mode, Barrier, Action and Test. In most cases the Actions are the most important information for the developer regarding

implementation. Each of the Actions from this example is shown as pseudo code in Figure 42.

```
Character check:
for each character c in String s do:
        if(c->isLetter){
                continue;
        }
        else{
                output error message(The name does not only contain letters);
        }
end

Check if empty:
do:
        String s := inputStringName;
        if(s->isEmpty){
                output error message(The name field is empty);
        }
end

Length check (for Norwegian zip code):
do:
        int i := inputIntZipcode
        if(i->length is not 4){
                output error message(The zip code has unvalid length. The vaild length for Norwegian zip code is 4);
        }
end

Number check:
for each character c in int i do:
        if(c isNumber){
                continue;
        }
        else{
                output error message(The zip code does not only contain numbers);
        }
end
```

Figure 42: Pseudo code from Actions for the example

8.5 USER MANUAL

8.5.1 INTENTION

The intention of this tool is to collect and save robustness related experiences and choices made in the development of systems, and use these to increase the robustness in latter system development. The value of the tool increases each time it is used, as the user increases the information in the database each time any item is added.

8.5.2 ITEMS DESCRIPTION

8.5.2.1 INPUT ITEMS

An Input item is an input to a system, either coming from the user or from another system.

Example: *Phone number.*

8.5.2.2 FAILURE MODE

A Failure mode is a mode, way or manner in which an Input item can cause the system under development to fail during execution.

Example: A letter is entered into the Phone number and causing an exception when the program is trying to cast the string to an integer.

8.5.2.3 BARRIER

A Barrier prevents a Failure mode from occurring.

Example: Prevent non-integers from being accepted as a Phone number.

8.5.2.4 ACTION

An Action is used to realize the Barrier in your program.

Example: An if-statement checking if the Phone number is an integer.

8.5.2.5 TEST

A Test, in this context, is the same as tests in system development in general. It is used to confirm the functionality of an Action, see *section 8.5.2.4 - Action*.

Example: Enter “2223222A” in the Phone number field; make sure it gives a useful error message and that the system can continue execution afterwards.

8.5.3 USING THE TOOL

The first step in using the tool is to identify which Input items your system has. Once this list of Input items is ready, you can start using the tool. Look for your Input items in the *Input items database*, the leftmost tree in the program. Select all Input items that match your system’s Input items; they are then moved to the rightmost tree. For all additional Input items you have in your system that were not in the database, add them by right clicking or using the upper right menu. In Figure 43 the leftmost tree to select Input items from is shown. In the upper right corner the button *Add Input item* which can be used to add Input items is located. The three disabled buttons are activated when an item is highlighted. *Add item* is used to add a child item to an item, for instance a *Test* under an *Action*. *Delete item* removes an item completely from the database. In the down-right corner a description of the data model of the program is shown. The child item of *Input item* is *Failure mode*, which has *Barrier* as child item etc.

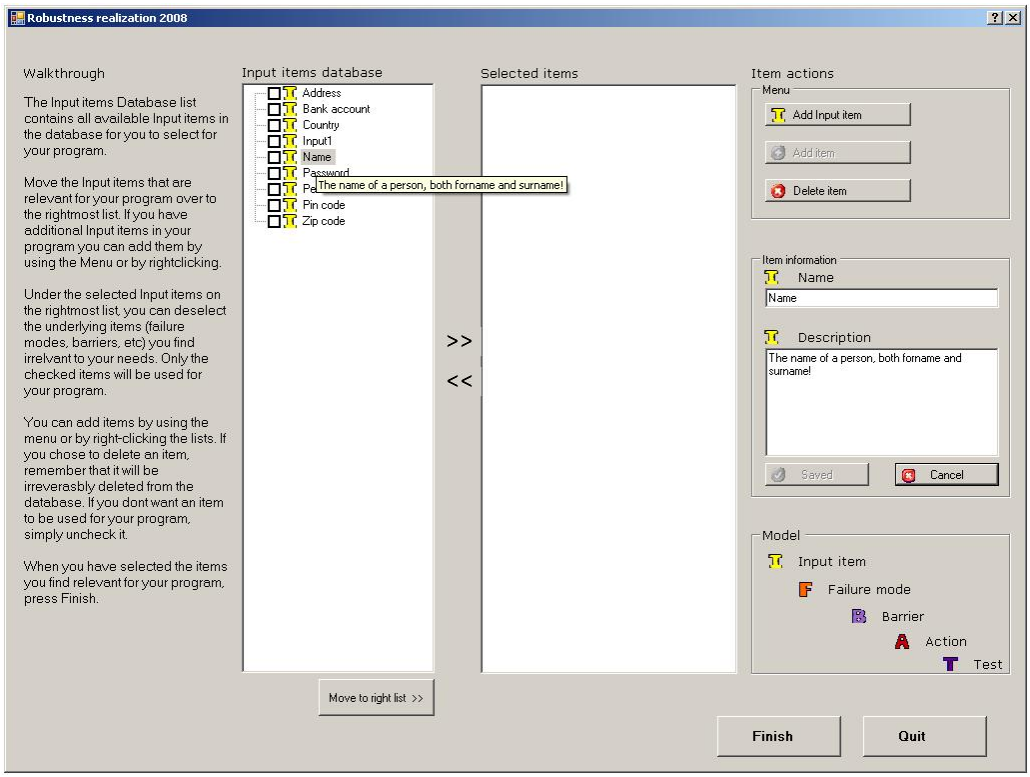

Figure 43: The first step, selecting input items

The second step is to select and add child items for all your Input items in the rightmost tree *Selected items*. If there are items that you find irrelevant for your program, deselect them by un-checking them and they are left out in the rest of the process. Deleting an item is only needed if you are sure that an item will never be relevant for any other systems. Remember that a **deletion cannot be undone**. For all the Input items you added yourself, relevant child items must be added. Figure 44 shows the rightmost tree with the selected Input items. As you see, *T1* and *T2* under *Name* are un-checked, which means that these items are not shown on the result page. A new Input item *Phone number* is added, with new child items which need to be specified.

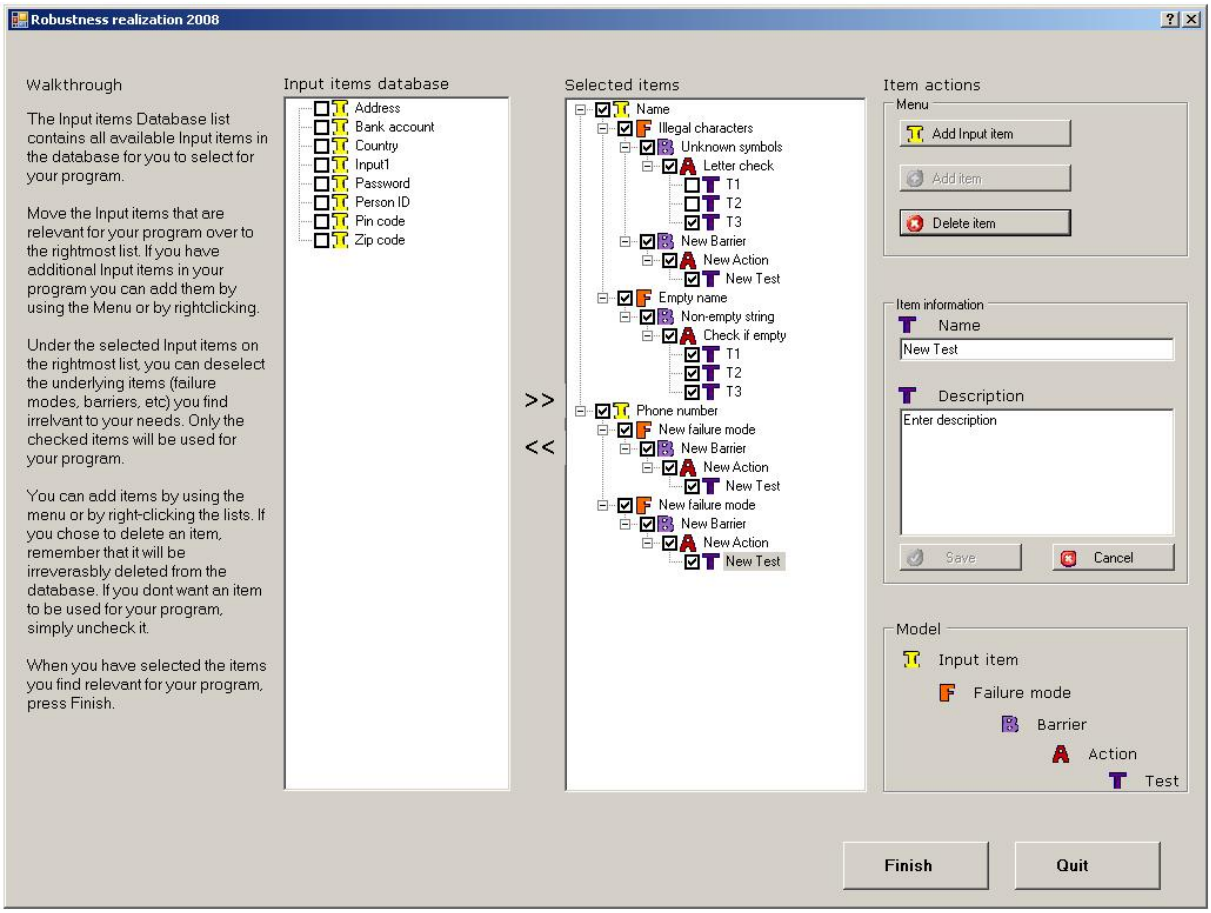

Figure 44: The second step, selecting child item

The final step is to click the Finish button to create the htm-file with the item names and descriptions for your selected items. It should pop up in your browser, but if it does not you can find *Results.htm* in the same folder as your program. Once you have the results, you can print them and implement the items in your system to increase the robustness.

9 CONCLUSIONS

9.1 COMBINING WITH JACOBSON'S METHOD

Jacobson's method was presented in *section 1.1 - State of the art*. We observed that combining Jacobson's method with our program could solve some issues and this allowed us to make some shortcuts. Jacobson's method describes how the actor should only be able to communicate with a boundary object. The boundary object takes care of the input validation, and when combining this with our program we were able to find actions for each Input item. Figure 45 shows the rules for Jacobson's method.

Figure 46 shows how our system can be combined with Jacobson's method. For each Input item in the use case, the user should apply our tool to find Failure modes, Barriers, Actions and Tests. It is the Actions and Tests that will be used further. The system provides actions to the use case on how to secure the input validation. Combining our system with Jacobson's method is advantageous for users that previously only used one of them. When using the Jacobson's method the actor-boundary object relation are served by our system, and this means that the use case will be served with Actions to better the robustness regarding the Input items from the user. When a user only uses our system it could be useful to combine with Jacobson's method to provide more benefits to the use case than only robustness realization.

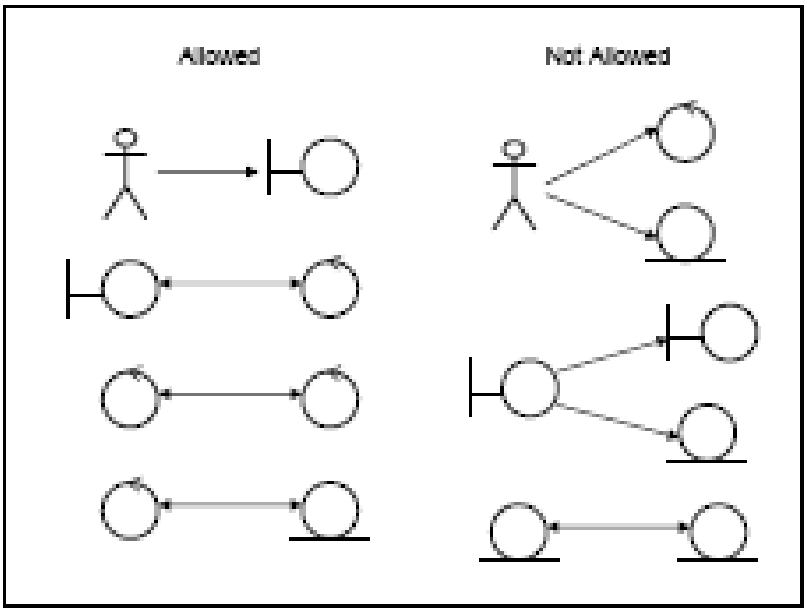

Figure 45: Jacobson's rules

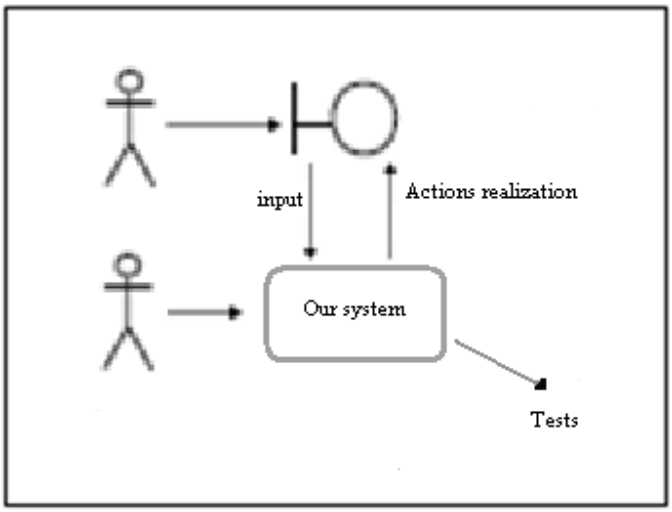

Figure 46: Our system interacting with Jacobson's input validation

9.2 COMBINING WITH TEST-DRIVEN DEVELOPMENT

Test-driven development is a software development technique that requires that the tests are written before the actual program code. This ensures rapid feedback after changes to the code. An automated unit test that defines the requirements of the code has to written before the code (Beck 2003). Since our tool support the developer with suggestions for Tests there is an opportunity of using our program in test-driven development. The results from our program make it easy to develop the Tests extracted from the tool. This means that the tests regarding the robustness in the development can be implemented by using our program in the test-driven development.

Using the tool in test-driven development was not a goal for this thesis, but the opportunity was realized at the end of the project. Thus there was no time to elaborate on this possibility, so we briefly scratched the surface when it comes to the possibilities. For any further development of the tool, the collaboration with test-driven development could be looked into more thoroughly.

9.3 FURTHER WORK

This section contains the work which we did not have either time or resources to finish as a part of this thesis, but were important for further development. As extra work and requirements came up during our work, evaluations were made to determine which tasks that would be prioritized in the limited time that was left. The tasks mentioned below were downgraded, mostly because they were not important to the project, but more of a final touch.

9.3.1 COPYING NODES

In both the business test and student test, copy functionality for the nodes in the trees was requested. This could be a part of the right click menus for the trees. It should not be a difficult task, as the NodeModel class offers methods for copying nodes.

9.3.2 UNDO

Undo functionality is common in most programs, and was requested in both our tests. This was considered an important task, but found too time consuming to complete. One way of implementing an undo function is to store many serialized files so that upon undo request, an older file could be loaded. However, as the Model is specified and implemented, this causes problems. The Model does not separate between the two trees in the program, it saves all the nodes without knowledge of which tree they belong to at the time of saving. This means that if an old file is loaded all the nodes are be put into the same tree, namely the left one. If this method should be used for implementing undo functionality, the serialization would have to differentiate between the two trees.

9.3.3 PROJECTS

The tool has no functionality for saving info on which items the user has moved to the right tree from time to time, as only changes made to the items are saved. In the business test it was requested that the choices made (the right tree of selected items) could be saved. The developers imagined that one would use the tool several times in a development project, and thus saving the trees would be needed. The main reason for not implementing this was the amount of work needed. Some major changes would have to be made to the tool, and the design and time-frame did not allow these changes. Our suggestion for this functionality is to use a separate serialized file for each project, and allowing loading and saving project in the tool. One common database of items would exist, but instead of moving the items between the trees they would be copied. The user could then make changes to the items that would only apply to the current project and not the common database.

9.3.4 CHECKBOXES AND DELETION OF NODES

The checkboxes next to the items have been an issue. As mentioned in *section 7.2 - Graphical user interface*, they are used to select which items are relevant for the user's design. During the experiments, however, it became clear that these checkboxes were not intuitive to use. When asked to remove the items that were irrelevant for the design, the users chose to delete the items instead of un-check them, even if the walkthrough explained the proper way to do it. Hence, the use of checkboxes has to be reconsidered; as the users seem to use deletion for the purpose the checkboxes were meant for. A solution of this problem can be to remove the checkboxes and separate between the database and the items in the rightmost tree, like suggested in *section 9.3.3 - Projects*.

9.3.5 DESIGN OF THE PROTOTYPE

This is the most important part of the further work in our opinion. Much of the work done in this project was to implement good GUI and functionality for the tool. The requirements for the project were created by the authors, and as this work has been research driven, there has been some trying and failing, leading to a design and implementation that is not ideal. Hence, for further work regarding the tool, we suggest starting over with the design phase and create a new design and implementation. This should not be too time consuming, and will, in our opinion, pay off in the long run. Implementing all functionality mentioned in this chapter would easier if it was included in the original design. The new design can benefit from the experiences and results of this thesis, and elements from the current design and implementation can probably be used. Some of the other suggestions for further work would create some issues with the current design and implementation, e.g. saving progress from time to time as mentioned in *section 9.3.3 - Projects.*

9.4 FINAL THOUGHTS

9.4.1 PROCESS

Our work process throughout this thesis has been iterative, changing between development and user tests. We started by developing a prototype of the tool, and then tested it on a group of students for input on usability and errors in the implementation. This was followed by a new phase of development before testing the updated prototype on developers in a software development company. Finally, we implemented the changes suggested and extracted from the last test, which resulted in the tool delivered with this report.

We have few regrets about our work process. The tool was developed for use in the industry, which we included during the entire development, from requirement specification through testing. During the early stages of the thesis we had a somewhat vague idea of how the tool would look in the end. Our mental model evolved throughout the development, and this led to a good result. However, if we had spent more time elaborating on how the tool would end up, we could have discussed our ideas with the industry and made changes correspondingly. Instead, we got this input during testing, and thus some suggestions we got were too late to incorporate into the system.

9.4.2 EXPERIMENTAL THREATS

In our experiments there were some threats that we could not prevent before the experiments started. The threats we revealed were the following:

- Selection of participants
- Hypothesis guessing

The selection of participants was done by advertising to students participating in the course *TDT4140 Software Engineering*[11]. They were offered a wage corresponding to NTNU's policy to participate, and the students that were interested volunteered for the experiment. People volunteering for experiments may be more positive in their answers than a non-volunteer, because they take an interest in the experiment. However, since the students collected a reward it was not necessarily the interest in the experiment that made them participate in the experiment. We therefore conclude that the volunteering for the experiment did not affect the results significantly.

It is hard to prevent people from guessing the experiment's hypothesis. The participants were asked to provide their opinions, and it was made clear that the qualitative descriptions

[11] http://www.idi.ntnu.no/emner/tdt4140/

were more important than the multiple choice answers. This may have prevented them from trying to guess the hypothesis. There were no answers that we suspected as guessing, and some of the participants did not answer all the written questions since they probably did not have a strong opinion or comment. There is no reason to believe that the participants were not sincere in their answers in any of the two test sessions.

9.4.3 RESULTS

The developed tool is the foundation of the thesis, and all requirements in *chapter 2 - Requirements specification* were fulfilled. Our expectations for this prototype were high, but as mentioned in *section 9.4.1 - Process*, we did not know how the tool would end up. Our hope during the early development phases was that it could be useful for the industry when we had completed the development. Later we realized that this was unrealistic goal, and we began considering the tool as a prototype instead of as a final product. Hence, the documentation of the tool has as high value as the tool itself. However, the potential for the tool is in our opinion high, and we got good response from the company that tested the tool. Since they previously had expressed some skepticism to the tool and its usefulness, we were satisfied with feedback after they tested it.

The design of the tool is good, but in retrospect some changes will be advantageous. These changes are needed to fulfill some of the tasks in *section 9.3 – Further work* to increase the quality of the design and to make the code more efficient. The code, or at least parts of it, can be reused in a new design if it does not divert too much from the current design. Our test results have high value, as they state what the users think of the tool as it stands. When creating a new design, these test results should be taken into consideration.

The research questions in *chapter 4 - Experiment execution* were used to guide our experiments. RQ1 looked to determine if the tool was easy to use. In *section 5.1.1 - Quantitative results* the student test results show that none of the students answered the most negative alternative in the questionnaire, while 50 % of the system developers did, see *section 5.2.2 - Quantitative results*. Still, 91 % of the total answers were above the most negative alternative, and hence we conclude with a positive answer to RQ1.

RQ2 looked to determine if the tool had utility value for the industry. To conclude upon this the answers to question 7 in Figure 27 was considered. One participant answered that it would not be useful, one that it could be and two participants that it would probably be interesting to use. We were satisfied with these results, especially since we also had several positive remarks regarding the value of the tool during the interview. The company has after the experiment shown interest in further development of the tool, which indicates that they see potential in the prototype. Thus we conclude that the tool is valuable for the industry.

PART IV
BIBLIOGRAPHY AND APPENDICES

BIBLIOGRAPHY

1. Bass, L., P. Clements, et al. (2003). Software architecture in practice. Boston, Addison-Wesley.
2. Beck, K. (2003). Test-driven development: by example. Boston, Mass., Addison-Wesley.
3. Braude, E. J. (2001). Software engineering: an object-oriented perspective. Hoboken, N.J., Wiley.
4. Huhns, M. N., V. T. Holderfield, et al. (2003). "Robust Software Via Agent-Based Redundancy."
5. IEEE (1998). "IEEE standard for software test documentation." IEEE Computer Society.
6. Kruchten, P. (1995). "The 4+1 View Model of Architecture." IEEE Softw. **12**(6): 42-50.
7. Reenskaug, T. (1979). Models - Views - Controllers**:** 2.
8. Ringdal, K. (2007). Enhet og mangfold: samfunnsvitenskapelig forskning og kvantitativ metode. Bergen, Fagbokforl.
9. Rosenberg, D. and K. Scott (1999). Use case driven object modeling with UML: a practical approach. Reading, Mass., Addison-Wesley.
10. Skjervold, Ø. and H. Haga (2007). "Robustness in Software Development."
11. Turlapati, R. and M. N. Huhns (2005). "Multiagent Reputation Management to Achieve Robust Software Using Redundancy."
12. Wohlin, C., P. Runeson, et al. (2000). "Experimentation in software engineering."
13. Zhou, J. and T. Stålhane (2004). "A Framework for Early Robustness Assessment."

APPENDIX A BUSINESS EXPERIMENT DOCUMENT

We are two master students that are developing a support application that will be useful for the software industry. Our main focus is to discover problems earlier than it is possible with today's practice. Textual use case is a known documentation of requirements, and will be used as the source of the users input in our program.

The experiments will be used to determine how user-friendly our program is and how useful this tool might be for your company.

Textual use case contains a sequence of tasks that shall be performed. The parts of the textual use case that are *input from the user in our users system* are the parts that are of interest to our application. We will first show small example.

A.1 EXAMPLE

The example shows how our application can be used. A textual use case describing the insertion of person data is elaborated, with focus on the input of zip codes. The following is a description of how the system shall handle this textual use case.

A.1.1 TEXTUAL USE CASE

The textual use case describes how the user inputs data and is presented Table 10. **A step in this use case is *register zip code*, which is the part this experiment will elaborate on.** Zip code is referred to as an *input item in* the application, and the application will extract failure modes for this item. The system will identify this input item, either automatically or manually through a user action. When the system has registered *zip code* as an input item, it displays the failure modes currently connected to this input item.

Table 10: Textual use case example

Register person information

Standard scenario

1. Register name
2. Register address
3. Register personal id number
4. **Register zip code**
5. Update database
6. User gets message whether the registration was successful or not

Exceptions

3a: Illegal personal id number

.1: System alerts user

.2: User returns to step 3 or aborts the session

4a: Illegal zip code

.1: System alerts user

.2: User returns to step 4 or aborts the session

A.1.2 FAILURE MODE FOR ZIP CODE

A failure mode describes how the input item can create a failure in a system. A failure mode is a possible way to fail, it is not necessary a failure. The application presents the user with failure modes that fit the input item zip code. In our case the application finds two failure modes:

FM1 (Too long) – The input length is wrong
FM2 (Illegal characters) – The input is not an integer

Also, the user chooses to insert a new failure mode:

FM3 (Not valid) *- The input is not a valid zip code.*

FM1 and *FM2* are general failure modes that can be used for many kinds of input items, not only zip codes. This also applies for the barriers, actions and tests.

A.1.3 BARRIERS

These are the elements that should prevent the failures from the taking place. The following barriers are suggested by the system:

B1 (FM1) – The input length should be of length n
B2 (FM2) – The input should be an integer
In addition, the user inserts a barrier for *FM3*.

B3 *(FM3) – The zip code should be a valid zip code.*

A.1.4 ACTIONS

Next the system presents the actions connected to *B1* and *B2*. These are:

A1 (B1) – An if-statement checking that the length of the input is n.
A2 (B2) – A check that the input is an integer.

In addition, the user creates a new action **A3** suited for *B3*. This action can for instance be a check towards postal systems that the zip code is valid.

A.1.5 TESTS

The system then present the user with tests to validate that the failure modes are taken care of through the barriers and actions. The system presents the following tests:

T1 – Input a number of length > n. This should fail.
T2 – Input a number of length < n. This should fail.
T3 – Input a number of length n. This should pass
T4 – Input a non-integer. This should fail.

In addition, the user must enter new tests to validate that *FM3* is handled correctly. He inserts:

T4 – Insert a valid zip code. This should pass.
T5 – Enter an invalid zip code. This should fail.

A.1.6 DATA STRUCTURE

The data structures are shown below in Figure 47.

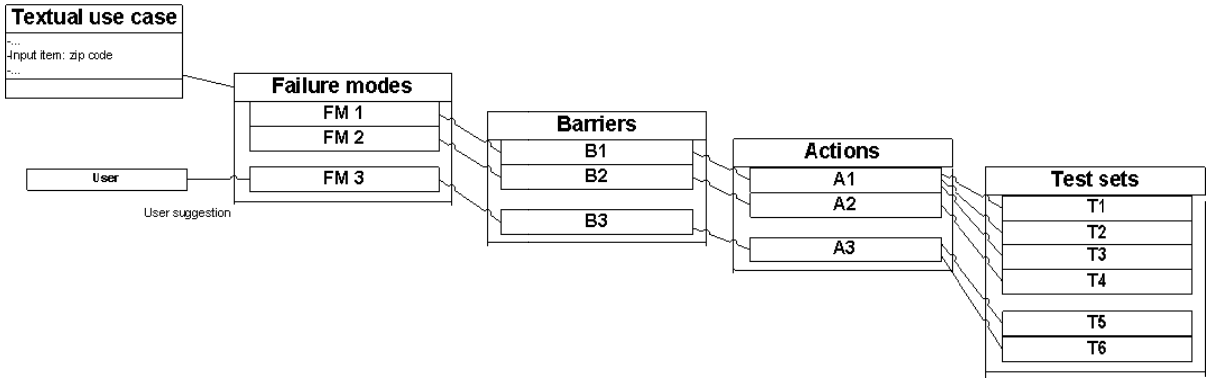

Figure 47: Relationships between example elements

The first four steps in the scenario will be useful input to our system. You shall now consider whether our system in your opinion is user-friendly or not. You shall also consider the usability of handling the input items, failure modes, barriers, actions and tests for further use. Please answer the questions at the end as precise as possible, and argument thoroughly for your choices.

A.2 TASKS

Your tasks are listed below. Please try to perform all the tasks before answering the questions. First of all we want you to navigate around in the program to make sure you locate the main functionalities in the program.

- CASE: The following textual use case is one of the use cases for the application you are developing. Find the input items in the use case and use our program to find failure modes, barriers, actions and tests.

Table 11: Textual use case for business test

Register person information

Standard scenario

1. Register name
2. Register phone number
3. Register personal id number
4. Register zip code

5. Update database
6. User gets message whether the registration was successful or not

Exceptions

3a: Illegal personal id number

.1: System alerts user

.2: User returns to step 3 or aborts the session

4a: Illegal zip code

.1: System alerts user

.2: User returns to step 4 or aborts the session

- Edit the failure mode called “Too long” for the input item zip code. The description should be: “The number contains fewer digits than allowed” and the name should be “Too short”.
- For the action called “Letter check” for the input item “name” you will not include more than test “T1” and “T3” to your set.
- Delete the barrier called “Unknown symbols” for the input item “Name”.

1) How easy was it to select input items from the leftmost list?

☐ This was intuitive

☐ It was pretty easy, but not intuitive

☐ It was pretty hard, but got it right

☐ This was hard to understand

Specify why it was easy or hard. Was there some functionality that you missed that would have made it easier for you?

2) How easy was it to add input items?

☐ This was intuitive

☐ It was pretty easy, but not intuitive

☐ This was hard, but got it right

☐ This was hard to understand

Specify why it was easy or hard. Was there some functionality that you missed that would have made it easier for you?

3) How easy was it to select the wanted failure modes, barriers, actions and tests under the input item?

☐ This was intuitive

☐ It was pretty easy, but not intuitive

☐ It was pretty hard, but got it right

☐ This was hard to understand

Specify why it was easy or hard. Was there some functionality that you missed that would have made it easier for you?

4) How easy was it to add failure modes, barriers, actions and input items?

☐ This was intuitive

☐ It was pretty easy, but not intuitive

☐ It was pretty hard, but got it right

☐ This was hard to understand

Specify why it was easy or hard. Was there some functionality that you missed that would have made it easier for you?

5) Was the difference between un-checking and deleting objects clear to you?

☐ The difference was intuitive

☐ It was pretty easy to understand, but not intuitive

☐ It was pretty hard to understand, but I saw the difference after all

☐ This was hard to understand

Specify why it was easy or hard. Was there some functionality that you missed that would have made it easier for you?

6) Did you notice the redundant options in the program?

☐ I did not notice because I mostly got it all by first try

☐ I missed some options to perform operations in the program

☐ I noticed this gradually and was happy about it

☐ I noticed this and found it confusing.

Specify why it was easy or hard. Was there some functionality that you missed that would have made it easier for you?

7) Do you think this tool could be a useful attachment to your software development in your company?

☐ Absolutely

☐ It will probably be very interesting to use this tool

☐ With some modifications this tool could be quite useful

☐ I do not think so

Specify why it was easy or hard. Was there some functionality that you missed that would have made it easier for you?

APPENDIX B BUSINESS EXPERIEMENT RESULTS

In this appendix all the answers from the professional test is written down. Since the participants were allowed to write in either English or Norwegian most of the answers are in Norwegian. This is because we do not want to modify any of the answers, but rather keep the originals.

QUESTION 1

“How easy was it to select input items from the leftmost list?” This question had following specified answers:

1. “Standard” virkemåte som gjør at det er enkelt å kjenne seg igjen
2. Prøvde først å få over elementene ved å klikke på de to pilene imellom listene. Oppdaget i etterkant at det var en egen knapp for dette lenger ned
3. Ikke normal standard arbeidsmåte. Pilene i skjermbildet gjør også at en forventer standard oppførsel (velge items, overføre mellom listene).
4. Intuitive due to previous learned behavior in other programs

QUESTION 2

”How easy was it to add input items?” This question had following specified answers:

1. Kan være vanskelig å se hvilken Input Item etc man jobber med
2. Bra med ikoner på knappene
3. The mix of input items/templates and actual items used is hard to understand
4. Hadde litt problemer med å se hvilken del av treet jeg befant meg på, men etter å ha gjort en feil ble det lettere

QUESTION 3

”How easy was it to select the wanted failure modes, barriers, actions and tests under the input item?” This question had following specified answers:

1. Foreslår at mer synlig bakgrunnsfarge enn lys grå benyttes
2. **Blank**
3. Easy, but was hard to get the concept difference between template and you were working on.
4. Selecting was no problem

QUESTION 4

”How easy was it to add failure modes, barriers, actions and tests?” This question had following specified answers:

1. **Blank**
2. Dersom man høyreklikker på et element og forsøker å adde, skjer det ingenting. Og når man derimot klikker på en addknapp rett etterpå, så addes det to stk elementer. Gjelder alle nivå.
3. Quite easy to understand, but the reuse of the same button was not a good choice. It was way to hard/cumbersome to do the actual work
4. This was no problem

QUESTION 5

”Was the difference between un-checking and deleting objects clear to you?” This question had following specified answers:

1. Det var intuitivt, men må ha undo
2. **Blank**
3. **Blank**
4. Hvis checking var “enabling” og deleting sletting

QUESTION 6

”Did you notice the redundant options in the program?” This question had following specified answers:

1. **Blank**
2. Som nevnt allerede. To matter å adde elementer på, men kun en funker.
3. Forstår ikke spørsmålet
4. Edit item knappen ble ikke brukt på grunn av redundans

QUESTION 7

”Did you think this tool could be a useful attachment to your software development in your company?” This question had following specified answers:

1. Skulle gjerne ha forsøkt det I et prosjekt. Integrasjon mot NUnit/Junit? Kopiere tester
2. Vanskelig å svare på forhånd. Hadde vært interessant å ta i bruk for å se hvor nyttig det er i praksis/i en reell case.
3. Slik det er nå er det altfor arbeidssomt
4. Burde være bra for å generere test prosedyre for UI validering

APPENDIX C BUSINESS INTERVIEW

This is the interview guide that was used in the interviews with the employees at the company in Trondheim which participated in the experiment. The questions were used as guidance, and if some questions proved unnecessary they would be skipped. Also, follow-up questions were added during the interviews when the employees gave answers that opened areas that we wanted to go deeper into.

C.1 QUESTIONS

The questions were divided into two parts: GUI considers the usability of the tool, while tool value will help determine if the developer found the tool valuable. For each question that gives a *yes* or *no* answer, remember to follow up by asking for reasons.

C.1.1 GUI

1. Were there any GUI options that you missed in our program?
2. Did you understand the intention of using two lists?
3. Do you have suggestions for improvements to the GUI?
4. What are you opinion on the presentation of the results?
5. Do you have other comments to the GUI?
6. Did you like the looks of the GUI?

C.1.2 TOOL VALUE

1. Do you consider the tool useful for developers?
2. Would you use the tool if it was available for you?
3. Do you think the database should be common for everyone, or personal for each developer?
4. What can be done to improve the tool?
5. Do you know or use competitive tools?

APPENDIX D INFORMATION BEFORE BUSINESS TEST

Vi er to femteklassestudenter ved NTNU, retning Datateknikk, som skriver en masteroppgave om robusthet i systemutvikling. Vi utvikler et verktøy som er tenkt brukt av utviklere for å øke robustheten i systemene de utvikler. Programmet tar utgangspunkt i input (*Input items*) til systemet som skal utvikles, og lar utvikleren bruke verktøyet for å få oversikt over hvilke robusthetsvalg han gjorde sist han hadde de samme Input item. Om et Input item ikke eksisterer i databasen, kan utvikleren legge det til selv. Vi ønsker å teste dette verktøyet på systemutviklere, både med tanke på brukervennlighet og bruksverdien.

Vi kommer til å gi dere en enkel case, og la dere teste verktøyet uten mye introduksjon eller opplæring, for å se hvor intuitivt programmet er for en førstegangsbruker. Det vil følge med en brukermanual som blir distribuert sammen med programmet. I etterkant av casen vil vi be dere besvare et enkelt spørsmålsskjema og delta på et kort intervju for å få deres meninger om programmet.

Har dere noen spørsmål før vi kommer ned til dere så ikke nøl med å ta kontakt på e-post <e-post> eller telefon <telefonnummer>.

Mvh,

Øyvind Skjervold

Håkon Haga

APPENDIX E STUDENT EXPERIMENT DOCUMENT

We are two master students that are developing a support application that will be useful for the software industry. Our main focus is to discover problems earlier than it is possible with today's practice. Textual use case is a known documentation of requirements, and will be used as the source of the users input in our program.

The experiments purpose is to determine whether how user-friendly our program is. Textual use case contains a sequence of tasks that shall be performed. The parts of the textual use case that are *input from the user in our users system* are the parts that are interesting for our application. We will first show an example.

E.1 EXAMPLE

The example shows how our application can be used. A textual use case describing the insertion of person data is elaborated, with focus on the input of zip codes. The following is a description of how the system shall handle this textual use case.

E.1.1 TEXTUAL USE CASE

The textual use case describes how the user inputs data and is presented in Table 12. **A step in this use case is *register zip code*, which is the part this experiment will elaborate on.** Zip code is referred to as an *input item in* the application, and the application will extract failure modes for this item. The system will identify this input item, either automatically or manually through a user action. When the system has registered *zip code* as an input item, it displays the failure modes connected to this input item.

Table 12: Textual use case for student test

Register person information

Standard scenario

1. Register name
2. Register address
3. Register personal id number
4. **Register zip code**
5. Update database
6. User gets message whether the registration was successful or not

Exceptions

3a: Illegal personal id number

.1: System alerts user

.2: User returns to step 3 or aborts the session

4a: Illegal zip code

.1: System alerts user

.2: User returns to step 4 or aborts the session

E.1.2 FAILURE MODE

A failure mode describes how the input item can create a failure in a system. The application presents the user with failure modes that fit the input item *zip code*. In our case the application finds two failure modes:

FM1 (Too long) – The input length is wrong
FM2 (Illegal characters) – The input is not an integer

Also, the user chooses to insert a new failure mode:

FM3 (Not valid) *- The input is not a valid zip code.*

FM1 and *FM2* are general failure modes that can be used for many kinds of input items, not only zip codes. This applies for the barriers, actions and tests as well.

E.1.3 BARRIERS

These are the elements that should prevent the failure modes from the taking place. The following barriers are suggested by the system:

B1 (FM1) – The input length should be of length n
B2 (FM2) – The input should be an integer
In addition, the user inserts a barrier for *FM3*.

B3 *(FM3) – The zip code should be a valid zip code.*

E.1.4 ACTIONS

Next the system presents the actions connected to *B1* and *B2*. These are:

A1 (B1) – An if-statement checking that the length of the input is n.
A2 (B2) – A check that the input is an integer.

In addition, the user creates a new action **A3** suited for *B3*. This action can for instance be a check towards postal systems that the zip code is valid.

E.1.5 TESTS

The system then present the user with tests to validate that the failure modes are taken care of through the barriers and actions. The system presents the following tests:

T1 – Input a number of length > n. This should fail.
T2 – Input a number of length < n. This should fail.
T3 – Input a number of length n. This should pass
T4 – Input a non-integer. This should fail.

In addition, the user must enter new tests to validate that *FM3* is handled correctly. He inserts:

T4 – Insert a valid zip code. This should pass.
T5 – Enter an invalid zip code. This should fail.

E.1.6 DATA STRUCTURE

The data structures are shown below in Figure 47.

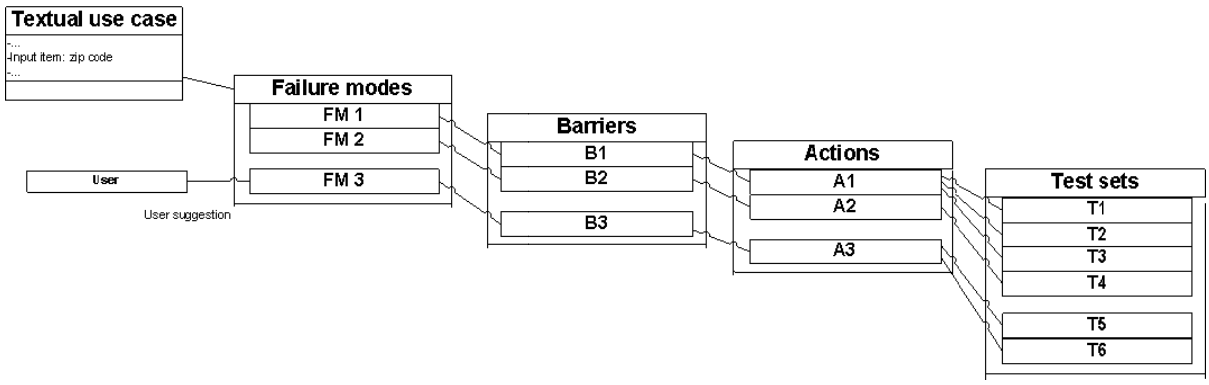

Figure 48: Relationships between example elements

The first four steps in the scenario will be useful input to our system. You shall now consider whether our system in your opinion is user-friendly or not. You shall also consider the usability of handling the input items, failure modes, barriers, actions and tests for further use. Please answer the questions at the end as precise as possible, and argument thoroughly for your choices.

E.2 TASKS

Your tasks are listed below. Please try to perform all the tasks before answering the questions. When finishing the tasks you may click “ok”, but the application will then terminate.

- Navigate in the program by using the example data
- Add a test under the input item “Address”
    - Test name: “T22”, Test description: ”String testing”
- Add an input item called “Personal id number” and add failure modes, actions etc so it has the same abilities as “Zip code” described in the example above. You will have to do this manually because we do not have a database connected yet, and it is therefore not yet possible to add already existing abilities to the input items.

- Edit the failure mode called “Too long” for the input item zip code. The description should be: “The number contains fewer digits than allowed” and the name should be “Too short”.
- For the action called “Letter check” for the input item name you will not include more than test “T1” and “T3”.
- Delete the barrier called “” for the failure mode “”.

1) **How easy was it to select input items from the leftmost list?**

   ☐ This was intuitive

   ☐ It was pretty easy, but not intuitive

   ☐ It was pretty hard, but got it right

   ☐ This was hard to understand

   Specify why it was easy or hard. Was there some functionality that you missed that would have made it easier for you?

2) **How easy was it to add input items?**

   ☐ This was intuitive

   ☐ It was pretty easy, but not intuitive

   ☐ This was hard, but got it right

   ☐ This was hard to understand

   Specify why it was easy or hard. Was there some functionality that you missed that would have made it easier for you?

3) **How easy was it to select the wanted failure modes, barriers, actions and tests under the input item?**

   ☐ This was intuitive

   ☐ It was pretty easy, but not intuitive

   ☐ It was pretty hard, but got it right

   ☐ This was hard to understand

   Specify why it was easy or hard. Was there some functionality that you missed that would have made it easier for you?

4) How easy was it to add failure modes, barriers, actions and tests?

☐ This was intuitive

☐ It was pretty easy, but not intuitive

☐ It was pretty hard, but got it right

☐ This was hard to understand

Specify why it was easy or hard. Was there some functionality that you missed that would have made it easier for you?

5) Did you notice the redundant options in the program?

☐ I did not notice because I mostly got it all by first try

☐ I missed some options to perform operations in the program

☐ I noticed this gradually and was happy about it

☐ I noticed this and found it confusing.

Specify why it was easy or hard. Was there some functionality that you missed that would have made it easier for you?

APPENDIX F STUDENT TEST RESULTS

These are the answers from the student test. Since the students were allowed to write in either English or Norwegian most of the answers are here stored in Norwegian. This is because we did not want to modify any of the answers, but rather keep the originals.

QUESTION 1

"How easy was it to select input items from the leftmost list?" This question had following specified answers:

1. Eneste som var å trykke på, så det var relativt enkelt å starte.
2. Det var enkelt å forstå hvordan dette skulle gjøres.
3. Veldig fin måte. Ble litt forvirret ved første trykk. Ingen forbedringer trengs. Vanesak.
4. Visste ikke hva meningen med programmet var, så koblingen mellom "Input items" og "Selection tree" var ikke intuitiv. Vanskelig å flytte elementer fra høyre til venstre liste.
5. Det er en tom boks, og man har ikke mange valgmuligheter å klikke på. Hvorfor er "Root" hektet av? Bør ikke ha hake i det hele tatt. Hva menes med "Root"? Trengs den?
6. Det var intuitivt, men mildt irriterende at jeg ikke fikk se under haken deres før etter at de var valgt.
7. I had a hard time understanding the difference between the two lists.
8. Vanskelig å forstå trestrukturen før man får prøvd seg litt frem. Ellers greit. Personlig liker jeg ikke at funksjoner kun er tilgjengelig med høyreklikk.
9. Tok et par minutter før jeg skjønte hvordan programmet fungerte. Men gikk veldig fin så fort jeg kom inn i det. Kanskje en litt lettere introduksjon på starten. Fikk ikke så mye ut av walkthrough som var der. Si hvilken tabell (venstre eller høyre) ting ligger i for eksempel.
10. Når du velger/"checker" av et item i venstre kolonne kan den forbli i kolonnen?
11. Skjønte ikke helt forskjellen på "input item list" og "selection tree", men å velge items var ganske selvforklarende.

QUESTION 2

"How easy was it to add input items?" This question had following specified answers:

1. Gikk greit å legge til når jeg klarte å markere rett boks.
2. Dette var også enkelt og greit.
3. Var bare å trykke på knappen. Kunne vel ikke vært lettere å lagd ny. Kanskje automatisk lagring?
4. Stor enkel knapp med selvforklarende tekst.
5. Det er mulig å klikke på "Add input items" helt til knappen endrer seg. Da har man forstått funksjonaliteten.

6. Er ikke mer å si her. Eneste er vel kanskje at du ikke bør ha fokus på noe annet enn ”Root” uten at knappen blir til noe annet.
7. Somehow I assumed that highlighting one input item would allow me to add another, so it took a few seconds before I realized I had to go up one level (But once I realized, it was fine).
8. Lett å skjønne når jeg fan tut at jeg måtte høyreklikke. Savner en Generell meny i programmet med ”edit” etc.
9. Dette gikk fint, men på dette tidspunktet hadde jeg allerede fått en god flyt i programmet. Kanskje nevne i instruksjonene at ”Root” må være markert?
10. Kopiering av deler eller hele trær for så å kunne redigere dem.
11. Når man først hadde skjønt modellen og lagt til fra input items list var add input items enkelt.

QUESTION 3

”How easy was it to select the wanted failure modes, barriers, actions and tests under the input item?” This question had following specified answers:

1. Dette gikk greit. Veldig oversiktlig og enkelt å navigere til ønsket felt.
2. **Blank**
3. Syntes fargene på ikonene gjorde det ganske uoversiktlige. Mer nøytrale farger hadde kanskje gjort seg?
4. **Blank**
5. Bare å klikke. +/- gjør det enkelt for Windows-brukere.
6. Vanskelig å komme med noen kommentar med tanke på at det bare gikk ut på å trykke på dem.
7. **Blank**
8. Veldig greit å bare klikke på + så kommer ting frem.
9. Programmet var veldig enkelt å forståelig så fort man kom inn i det, men kunne kanskje bli litt uoversiktlig i lengden. Muligens noe som skapte et skille mellom input items i den høyre tabellen? For eksempel en tykk strek.
10. Enkel nivåstruktur gir god oversikt.
11. Blir fort litt rotete med alle trærne nedover, men siden det nettopp er en trefunksjon var det greit å minimere for å rydde opp.

QUESTION 4

”How easy was it to add failure modes, barriers, actions and tests?” This question had following specified answers:

1. Var greit å legge til det som var ønskelig, men jeg brukte noen sekunder på å forstå hvor jeg måtte stå for å legge til det jeg ville.
2. Dette var greit. Fint at hele systemet er konsist slik at alt gjøres på samme måte.

3. Var bare å trykke på knappen. Kunne vel ikke vært lettere å lagd ny.
4. Ikke intuitivt at ”barriers” og ”actions” må opprettes før ”tests”.
5. Knappen som endret seg gjorde det enkelt
6. Er ikke mer å si her.
7. Easy when I realized that I had to go up one level first.
8. Lett å skjønne det, men det var noe vanskelig å finne ut HVA failure mode er.
9. Enkelt å forstå. Kunne bli litt uoversiktlig.
10. Greit med knappene som endrer tilstand/navn
11. Når man hadde skjønt modellen var dette enkelt.

QUESTION 5

”Did you notice the redundant options in the program?” This question had following specified answers:

1. Til å utføre den oppgaven vi ble tildelt var programmet enkelt å bruke. Fikk bruk for alle menyene. Disse var enkel å forstå å bruke.
2. La merke til høyreklikk på nodene, samt avkryssing + dobbelklikk for å legge til input item.
3. Brukte aldri ”edit item”.
4. En knapp for å flytte elementene mellom listene.
5. ”Edit item” knappen ble ikke brukt. Er sikkert kjekk for noobs (red. nybegynnere). Slettet test istedenfor å fjerne hake. Kan bedres ved å fremheve funksjonaliteten til haken.
6. Eneste er at undo-effekten har litt kort hukommelse.
7. Only noticed that I could add input items from both lists, which I liked (but I still don’t see why there are two lists. Wouldn’t it be the same to just leave them in the same list where you select them?).
8. Ja, fargekoding var nyttig og oversiktlig.
9. Skapte ingen problemer, så de var ikke i veien, var heller bare et pluss. Kan kanskje bli forvirrende med flere options.
10. Kopiering som nevnt tidligere.
11. ”Edit item”-knappen virker overflødig…?