A Software Tool for Risk-based Testing

TDT4735 Systemutvikling, fordypning

Autumn 2004

Lars Kristoffer Ulstein Jørgensen

Supervisor: Tor Stålhane

Preface

This project is a part of the graduate level course TDT4735 Systemutvikling, fordypning (Software Engineering, depth study) at Department of Computer and Information Science, Norwegian University of Science and Technology (NTNU), during the autumn of 2004. The course is a part of the Master of Science degree, in the ninth semester of the program.

The report discusses Risk-based Testing and how to develop a software tool to support Risk-based testing. A prototype of the software tool is presented.

I would like to thank my supervisor, Tor Stålhane, at the Department of Computer and Information Science, NTNU, for valuable help and feedback during the semester.

Trondheim, November 26, 2004

Lars Kristoffer Ulstein Jørgensen

Abstract

Testing is a time consuming part of software engineering. It is often done late in the project when there are not much time and money left. As a result, everything can not be tested. It becomes important to choose what to test. This paper describes a method for risk-based testing. A risk analysis is carried out to find the risk of each use case. Factors affecting the cost and the likelihood are presented. Valuations of these are used to find the risk exposure for each use case. Test cases are generated from the use cases and the test cases with highest risk are chosen. Requirements for a software tool to support this method are discussed and a prototype presented.

Table of Contents

List of tables

List of figures

1. Introduction

Considering risk is not new in software testing. *"Traditional testers have always used risk-based testing, but in an ad hoc fashion based on personal judgment"* [7]. The purpose of this project is to develop a software tool that will help the testers to use risk-based testing. A methodological way to use risk-based testing is proposed and used in a prototype of the software tool. First Risk-based testing is explained. We continue by looking at approaches found in the literature. Then the method chosen in this paper is described. Next we discuss how this can be implemented in the software tool provided, before the specification of the requirements.

2. Risk-Based Testing

2.1 Risk

Risk is a possible future, unwanted event that has negative consequences. According to Wikipedia [12], *"Risk is the potential future harm that may arise from some present action."* Risk is not a problem yet, but it can be if not the right actions are taken. When considering the risk we have to look at the cost and the likelihood that it will happen.

The **cost** is the loss associated with the event. These are negative things like loss of time, quality, money, control or understanding. As an example big changes in the requirements are likely to lead to losses of time and money if the design is not flexible enough.

The **likelihood** that an event can lead to harm is usually calculated as a probability from one to zero. If the probability is zero it will never happen and should not be considered as a risk. A probability of one means that the event will happen unless something is changed. To consider the risk, the risk exposure has to be calculated. The risk exposure is found by multiplying the cost and probability of the event. This will give the expected average loss.

Risk Exposure = Cost * Probability

It is also important to look at the degree to which we can **change the outcome**. How can we minimize or avoid the impact of a risk? Risks can be accepted, prevented or transferred. If the risk is accepted, the risk should be part of the calculation of cost. If there are many risks that must be accepted, the risk exposure can be used in the cost calculation. This will give the average loss. Prevention is a way to reduce the cost or the probability. This can be done by changing the condition so the event will not happen. In software engineering, to do more testing is a way to reduce the risk. It will reduce the uncertainty and may find critical faults which can be corrected. Transferring the cost means that the cost is put on another parts, like the customer, a subcontractor or an insurance company. To transferring the cost to the customer is in itself a risk, since it can give bad reputation that can again lead to loses of customers and market shares.

2.2 What is Risk-Based Testing?

The **purpose of testing** is to find faults in the system and build confidence that the system provide the functionality specified in the requirements and not cause losses to the customer [5]. The testing will also provide data on the risk of release. [10] Testing in software engineering "*is a process used to identify the correctness, completeness and quality of developed computer software. Actually, testing can never establish the correctness of computer software. It can only find defects, not prove that there are none. There are a number of different testing approaches that are used to do this ranging from the most informal ad hoc testing, to formally specified and controlled methods such as automated testing"* [12].

Testing is **time consuming**. From collected data only 20 % of the effort is spent on coding. The preceding phases and testing each consumes about 40 % of the effort. *"In many organizations, software testing accounts for 30 to 50 percent of software development costs. Yet most people believe that software is not well tested before it is delivered"* [15]. Testing is often done late in the project when not much time or money is left. This is a problem since testing needs so many resources. If the system should be delivered in time a lot of things will not be tested. Most managers will choose to deliver on time rather than a system that works perfectly. [10] Testing is often underestimated because it is expensive. If the estimates were realistic, it would take away resources from the rest of the process. The result would be that they fail to deliver enough functionality. *" It squeezing the test planning in the hope that, it'll be all right in the night and that faults can be fixed quickly in production without any disasters, then maybe, just maybe, they'll get away with it"* [10].

This practice will continue until the organization has experienced enough failures to change the development climate. All this put a lot of pressure on the testers. To prioritize what to test to test the most important areas becomes important. By identifying and prioritizing risks, we can make sure that the limited time and resources are used to test the most important things. Risk-based testing is an approach to decide what to test and test the right thing by using risk as a way to consider the cost of not testing.

Most risk based testing is **black box testing**. This technique is also called functional testing or specification-based testing and it checks the system against the requirement specification. It does not look at the internal structure, but provide the system with inputs and observe the outputs. Black box testing *checks "that the outputs of a program, given certain inputs, conform to the functional specification of the program. The term black box indicates that the internal implementation of the program being executed is not examined by the tester. For this reason black box testing is not normally carried out by the programmer. In most real-world engineering firms, one group does design work while a separate group does the testing"* [12].

Risk-Based testing should be used together with other techniques. Inspection and continuous code testing during implementation are also important. This is called **white box testing** and it checks *"that the outputs of a program, given certain inputs, conform to the internal design and implementation of the program. The term white box indicates that the tester closely examines the internal implementation of the program being tested"* [12]. Often most of the errors are found during this earlier phases of testing [11] and they are easier to corrects. Black box testing, however, is also essential to ensure the quality of the system before it is delivered.

2.3 Different Approaches

There are several ways to do Risk-Based Testing. We will look at a many of these in the following section. The approaches will then be compared.

2.3.1 Amland: A case-study using risk-based testing

Amland describes a case study [1] of the development of a retail banking application. During the project, they realized that there were not enough resources to test everything. Two approaches were used to decide what to test.

In the first part of the system to be tested, a combination of the **top 20 most important** transactions and all transactions with **more than 10 faults** from previous testing were tested. This was early in the project and very little information was still available from the construction teams.

Later when more information was available, more factors were considered. The **cost** of fault for **customer** C(c) and **vendor** C(v) in the areas of maintenance, legal issues, and reputation were considered for each function. The cost of fault for the vendor and customer were considered as equally important - thus the cost for each function was the sum of for each divided by two.

The **probability** of fault for each function P(f) were estimated. This depended on whether the code was changed, if there was new functionality, the design quality, the size and the complexity. These factors were given weights from one to five, where the higher are the most important indicator of a function with poor quality. Then, for each function of the system the probability for all indicators is then considered given values from one to three. The weights and values for each function are multiplied. The probability *"is calculated as the Weighted Average of a weighted function divided by the highest Weighted Average of all functions"* [1]. This gave a probability numbers between zero and one.

The **risk exposure** for each function RE(f) was calculated by multiplying the cost of fault and the probability using the formula:

$$\mathbf{RE(f) = P(f) * (C(c) + C(v)) / 2}$$

The testing resources where focused on the functions with the highest risk exposures.

Func.	Cost			Probability						Risk Exp. funct. Re(f)
	C(v)	C(c)	Avrg. C	New Func. 5	Design Quality 5	Size 1	Compl. 3	Weight Avrg.	Proba-bility P(f)	
Close Accnt.	1	3	2	2	2	2	3	7,75	0,74	1,48

Table 2.1: Example of calculated Risk Exposure for the function "Close Account" taken from Amland [1]

2.3.2 Bach: Heuristic Risk-Based Testing

Bach [2] uses a **heuristic analysis** to do risk-based testing. According to Wikipedia [12] *"a heuristic for a given problem is a way of directing your attention fruitfully to a solution. It is different from an algorithm in that it merely serves as a rule of thumb or guideline, as opposed to an invariant procedure."* [12]

The method used by Bach is not final or strict. Human experience is used to find the most risky areas. Bach presents two approaches to his heuristic Risk-Based testing, inside-out and outside-in.

In **inside-out** the tester study the product together with the developer and asks questions like "What can go wrong here?" For each part of the product the tester will look for vulnerabilities, threats and victims.

Outside-in begins with a set of potential risks and matches them to the details of the situation. A list of predefined risks is consulted and used to determine whether they apply here and now. Bach uses three kinds of lists. Quality Criteria Categories, Generic Risk List and Risk catalogues.

Quality Criteria Categories is designed to evoke requirements such as: capability, reliability, usability, performance, installability, compatibility, supportability, testability, maintainability, portability, and localizability

Generic Risk List is a list of risks that are applicable to any system:

- *Complex:* anything disproportionately large, intricate, or convoluted.
- *New:* anything that has no history in the product.
- *Changed:* anything that has been tampered with or "improved".
- *Upstream Dependency:* anything whose failure will cause cascading failure in the rest of the system.
- *Downstream Dependency:* anything that is especially sensitive to failures in the rest of the system.
- *Critical:* anything whose failure could cause substantial damage.
- *Precise:* anything that must meet its requirements exactly.
- *Popular:* anything that will be used a lot.
- *Strategic:* anything that has special importance to your business, such as a feature that sets you apart from the competition.
- *Third-party:* anything used in the product, but developed outside the project.
- *Distributed:* anything spread out in time or space, yet who elements must work together.
- *Buggy:* anything known to have a lot of problems.
- *Recent failure:* anything with a recent history of failure.

Table 2.2: This is an example of a Generic Risk List, taken from Bach [2].

Risk catalogues is a list of risks that belong to a particular domain. Below follows an example from installation. Different problems that may occur in this domain are listed.

- Wrong files installed
    - temporary files not cleaned up
    - old files not cleaned up after upgrade
    - unneeded file installed
    - needed file not installed
    - correct file installed in the wrong place
- Files clobbered
    - older file replaces newer file
    - user data file clobbered during upgrade
- Other apps clobbered
    - file shared with another product is modified
    - file belonging to another product is deleted
- HW not properly configured
    - HW clobbered for other apps
    - HW not set for installed app
- Screen saver disrupts install
- No detection of incompatible apps
    - apps currently executing
    - apps currently installed
- Installer silently replaces or modifies critical files or parameters
- Install process is too slow
- Install process requires constant user monitoring
- Install process is confusing
    - UI is unorthodox
    - UI is easily misused
    - Messages and instructions are confusing

Table 2.3: This is an example of a Risk Catalogue for installation taken from Bach [2].

The three kinds of **lists can be used to**:

1. Decide what component or function to be analyzed
2. Determine the scale of concern. Everything is assumed to have a normal risk unless there are reasons to believe something else.
3. Gather information about the things you want to analyze more
4. Determine each risk's importance
5. Record any other risk that are not on the list
6. Record any unknowns which impact the ability to analyze the risk
7. Check the risk distribution

Three ways to organize risk-based testing is suggested, risk watch list, risk/task matrix and component risk matrix. **The risk watch** list is a list of risks that you periodically review. **Risk/task matrix sort** the risks according to their importance. For each risk what tasks to be invested in to minimize the risk is listed. This is a tool that is most useful when resources for testing are negotiated. The **component risk matrix** breaks the product into 30 or 40 areas or components. In the left column the components are listed. In the middle the scale of concern, low, normal or high risk is listed. In the right column the risk heuristics for that component are listed. The risk heuristic indicates the risk for

that component. During testing the components are tested according to their risks in the matrix and focus on the risks that increase the need to test.

Component	Risk	Risk Heuristics
Printing	Normal	Distributed, popular
Report Generation	Higher	New, strategic, third-party, complex, critical
Installation	Lower	Popular, usability, changed
Clipart Library	Lower	complex

Table 2.4: This is an example of a component risk matrix taken from Bach [2].

2.3.3 Besson: A Strategy for Risk-Based Testing

Besson [8] describes how to do Risk-Based testing prioritizing the most vital functions. The method starts by identifying **vital functions** that may have faults of **high severity**. The severity is defined by measuring the negative impact a fault has on the business; high, medium or low. If the user cannot work with the software and wastes time and money because of the failure, the severity is considered to be high. A good example of a vital function is a login page for a Web application. The function is not considered if the user can still use the software and only causes some workaround for the users to achieve the goal.

Ways to get this information is to survey the end-user, ask domain experts or use statistics from the log of previous versions of the system. It is also important to look at the most frequently used functions, since more frequent use increases risk. **Test cases** are **designed** and are assigned to the functions. The test cases are **sorted** in order of **effort**, with the minimum time needed first and **run in this order**.

This method only selects functions with a high severity and sorts the test cases according to the time needed to run each test case. A prioritizing between the selected functions based on severity should also be considered. A problem that is commented from a reader is, the method *"is possibly flawed as it does not account for test case dependency. If "* the test cases are sorted only on time *"then you will find yourself attempting to run test cases that are dependant on those that require more effort. By creating test scenarios or a dependency list as a secondary input to this process overcomes this problem"* Steven Sampson [8].

2.3.4 Chen: An approach for risk-based regression testing

Chen [7] introduces a Risk-Based methodology for regression testing. Regression testing checks that the functionality that was working earlier is still working when faults are fixed. It validates modified software and provides confidence that the changed parts behave as intended and unchanged parts are not affected. Chen discusses a specification-based method for regression test selection.

The risk-based approach focuses on test cases which tests the risky areas. **The cost of fault** C(f) for each **test case** is estimated. Cost is categorized on a scale from one to five. There are two kinds of cost considered: The cost for the customer (loosing market share) and the cost for the vendor (high maintenance cost). The **severity probability** P(f) for **each test case** is found by looking on number of earlier defects and severity of these defects. The Risk Exposure is calculated for each test case. The formula for Risk Exposure RE (f) is taken from Amland [1].

$$\mathbf{RE\ (f) = P(f)\ x\ C(f)}$$

Scenarios covering one or more test case are created. A scenario is a sequence of steps describing the interaction between the user and the system. These scenarios are effective since they involve many components working together. A traceability matrix is made, showing what test cases each scenario covers. The scenarios should cover the most critical test cases and as many as possible. The Risk Exposure for each scenario is calculated. The scenarios with the highest Risk Exposure are run first. The traceability matrix is updated. New scenarios, with the focus on those test cases that has not yet been executed and with the highest Risk Exposure, are executed.

2.3.5 Gerrard: Risk-Based E-Business Testing

Gerrard [10] uses a risk-based approach to test e-business, but his method can apply to other kinds of system also. The process consists of five stages: risk identification, risk analysis, test scoping and test process definition. A table called Test Process Worksheet is the main working document in the method and is completed in stage 1 to 4. Each row in the Test Process Worksheet consists of a failure mode, also called risk. The columns consist of scoring and prioritization, assignment of test objectives, effort, costs and so on for this failure mode. Each stage is discussed below.

Stage 1: Risk identification
An inventory of potential failure modes is prepared. These are derived from checklists. These are similar to the checklists that Bach [2] is using, described in section 2.3.2. When the failure modes are identified, complex failure modes are split up and duplicates are discharged.

Stage 2: Risk analysis
In this stage a risk workshop is convened with representatives from the business, development, technical support and testers. Each risk is considered, and the probability and the consequence are assessed. The risk exposure is calculated. Table 2.5 shows an example.

Stage 3: Risk response
If the risk is testable, it is turned into a test objective using the risk description. An example can be for the failure mode: *"HTML used may work on one browser, but not others"* [10] can have the test objective "*Verify that HTML is compatible across supported browsers"* [10]. A risk effectiveness a score between one and five that relates

to how confident the tester is that the risk could be addressed through testing is given to each risk. A value of one indicates that the tester does not have any confidence. Five is given if the tester have high *"confident that testing will find faults and provide evidence that the risk has been addressed"* [10]. The product of the exposure and test effectiveness will give the test priority number.

ID	Risk	Probability	Consequence	Exposure	Test Effectiveness	Test Priority Number
	Client Platform					
04	Invalid or incorrect HTML may exist on pages without being reported by browsers or visible to developers or testers.	4	4	16	5	80
05	HTML used may work on one browser, but not others	4	4	16	4	64
06	Users with client machines set up to use foreign character sets may experience unusual effects on their browsers. Foreign characters may be rejected by the database.	3	4	12	4	48
	Component Functionality					
13	Spelling mistakes on Web pages may irritate users and damage our credibility.	5	4	20	5	100
	Integration					
19	Links to on- or off-site pages that do not work make the site unusable	4	4	16	5	80

Table 2.5: This example is a sample of risk analysis and test effective scores of the Test Process Worksheet. Taken from Gerrard [10].

Stage 4: Test scoping
Review activity, need involvement from all stakeholders and staff with technical knowledge. The meeting discuss and agree on a total budged for testing. The Test Priority Number and the estimated cost to run the test, are used to choose what test objectives to include in the scope for the testing. The responsible for each testing object is agreed.

Stage 5: Test process definition
The scope of the testing is agreed and it is now possible to compile the test objectives, assumptions, dependencies, and estimates for each test stage and publish a definition for each stage in the test process. The stage-by-stage test process is documented.

2.3.6 Schaefer: Prioritizing Test against Deadlines

Scheafer [9] focus on prioritizing what to test, by finding the most important and worst parts of the product. The **most important** parts of the system are found using factors like cost of failure, most visible and most used parts of the product. The **worst parts** of the system are found using defect generators like complexity, changed areas, new technology, new solutions, new methods, new tools, number of people involved, where there was time pressure and local factors.

Weights are assigned for each relevant cost factor and defect generator. For each part of the system, values are assigned for the factors and the defect generators. Higher values mean that the area is more important or worse. This is illustrated in table 2.5. These values are multiplied by the weights and added together. The highest values give the most risky parts and should be prioritized.

Area to test	**Business criticality**	**Visibility**	**Complexity**	**Change frequency**	**RISK**
Weight	3	10	3	3	
Order registration	2	4	5	1	46*18
Invoicing	4	5	4	2	62*18
Order statistics	2	1	3	3	16*18
Management reporting	2	1	2	4	16*18
Performance of order registration	5	4	0	1	55*3
Performance of statistics	1	1	0	0	13*0
Performance of invoicing	4	1	0	1	22*3

Table 2.6: An example calculating the risk, taken from Schaefer [9]

2.3.7 Stålhane: Finding failure modes

The **use cases** are used to find **failure modes**. For each failure mode the **consequences** (cost) for the stakeholders are identified. The severity of each failure mode is assessed [5]. When the risks are found, we need to know what make the problem occur and how we can prevent it. From the answer of these two questions, barriers inside the system are built. These control the data so only legal data is accepted. The main purpose for the test is then to show that the barriers work. This will prevent dangerous events from happening.

2.3.8 HazOp used to prioritize functions

HazOp is an example of a method used to identify hazards. Normally simpler and less formal methods will suffice. An example of the use of HazOp to prioritize functions of a system is described by Stålhane and Sivertsen [17].

UML use cases are used as a starting point. Use cases have no study nodes; thus a standard HazOp with guidewords could not be used. Instead, a functional HazOp is performed based on the functionality offered by the system. The procedure has the following steps:

Step 1: Prepare use cases for the subsystem to be analyzed.
Step 2: A warm up exercise looking at previous risk analysis.
Step 3: Perform function-based HazOp
-How can this function fail?
-Which consequences for stakeholders, the service receiver, the service provider and the development company? This will give a list of hazards
Step 4: Document result and obtain feedback
Step 5: Assess severity of each hazard

A score from zero to three indicating the severity is given to each hazard, where three is the most serious. Different stakeholders may assign different severities to the same hazard. For each subsystem and stakeholder group, the number of functions that receive each hazard value is registered. The score for each function is found in two different ways, the weighted average and the score to the majority of the functions. In the latter case, the median is used if the majority criterion did not give a unique result.

Subsystem	**Seriousness**				**Score**	
	3	2	1	0	**Average**	**Max.**
D1	1	-	1	-	2.0	2
B1	-	-	3	2	0.6	1
O1	-	-	3	1	0.8	1
X1	1	3	2	1	1.6	2
I1	6	-	3	-	2.3	3
G1	2	1	1	-	2.3	3
M1	-	1	2	-	1.3	1
A2	1	1	2	-	1.8	1

Table 2.7: An example of calculation of scores for different subsystems. Taken from Stålhane and Sivertsen [17]

2.4 Comparison of the approaches

Most of the examples described above use risk to **prioritize** what to test. Many use similar methods to the one of Amland [1] where factors that can increase the **cost** and consequences of failure for some parts of the product are found. Amland [1] look at the cost in respect of maintenance, legal issues, and reputation for the vendor and the

customer, while Schaefer [9] and Besson [8] look at what functions have the highest importance and cause bigger inconveniences of failure for the user. His important areas are more or less the same as Amland [1] calls cost.

Schaefer [9] consider many factors that will increase the **likelihood** for an error done by the developers – called defect generators. From these defect generators he can find the parts with most defects. Besson [8] looks on in what degree a failure prevents the user from using the system and frequencies of use to prioritize what to test. Chen [7] uses many of the same ideas as Amland [1], but looks at test cases, not functions.

Gerrard [10] use a different way to prioritize what to test. Instead of different areas of the product, a risk analysis on different the failure modes is performed. Tests are generated from the failure mode with the priority on the failure modes with highest risk. This is similar to Stålhane and Sivertsen [17] were HazOp is used to find **hazards**. The number and severity of the hazards for each function is used to prioritize the functions. This is another way to sort out problematic areas in a risk analysis. The risk analysis was done to decide what functions to put extra effort into, but could also be used to decide what functions to test.

Stålhane [5] analyze **how** use cases can **fail**. This is similar to Bach's inside-out where he looks at how functions can fail. The consequences for each failure are considered. Stålhane [5] assess severity for each failure mode. These two approaches does not try to give guidelines on what to test, they rather look at the risk in order to find possible faults. These methods can be helpful when use cases are gathered, but it is difficult to make a software tool that can help the tester.

Many ideas from the methods discussed are used in the rest of the method described in this report. As in the methods from Stålhane [5] [17], it uses use cases as a starting point. The corresponding tests are prioritized using many of the ideas from Amland [1] and Schaefer [7].

3. Our method for Risk-Based Testing

The method used in the rest of this paper uses risk to prioritize what to test for specification-based testing. In specification-based testing, the program is checked against a list of requirements. Use cases are used to describe functional requirements. The method can be summarized in five steps:

Step 1: Gather use cases for the system
Step 2: Perform a risk analysis to sort the use cases in order of risk
Step 3: Design test cases from the use cases with the highest risk
Step 4: Consider importance of each test case, to find a risk for each test case
Step 5: Estimate resources needed to run each and run the test cases with the highest risk until there is no time left

Each of these steps is discussed below.

3.1 Gather use cases for the system

The method starts by **analyzing the requirements** in the requirement specification to find use cases for the system. The requirement specification reflects the mutual understanding between the analyst and the client of the problem to be solved. The requirements describe what the system should do, but do not say how those requirements should be implemented. The requirements must be gathered early in the development process. Since the understanding of the system increase over time, they are normally changed and negotiated throughout the projects life cycle. The requirements are statements about the system that all stakeholders agree must be true to solve the customer's problem. A requirement specification has two kinds of requirements, functional and non-functional requirements.

"Non-functional requirements are constraints that must be adhered to during development. They limit what resources can be used and set bounds on aspects of the software's quality; thus they tend to restrict the freedom of software engineers as they make design decisions" [12]. Non-functional requirements may describe the level of quality, the environment and technology to be used, the project plan and the development methods. **Non-functional requirements** must also be verified and, if possible, tested.

Use cases, which are the starting point for this method, is a way to describe **functional requirements**. *"Functional requirements describe what the system should do*" [12]. From the functional requirements it should be possible to create a set of use cases that describe the system. A use case is a *"set of scenarios tied together by a common user"* [14]. A scenario describes a sequence of interaction between one or more users and the system. The use cases will divide the system into functions, so it will be easier to perform a risk analysis. It has also "*become the most common practice for capturing functional requirements. This is especially the case within the object-oriented community where they originated, but their applicability is not restricted to object-oriented systems, because use cases are not object orientated in nature"* [12].

To **gather use cases** you have to find the users that will interact with the system. Each user will need to do different things with the system. The same person can have different roles, and will then be a different actor. An actor does not need to be a human; it can also be an external system that needs to communicate with our system. When all the actors are found, it is time to look at what they need do with the system to find the use cases. When a list of all use cases is collected, the testers can go the next step in the method, to evaluate the risk of each use case.

3.2 Risk analysis

To prioritize the testing resources, we need to perform a risk analysis to find the **use cases** with the **highest risk**. The risk is found by multiplying cost and probability so we will look at factors affecting each of these. Areas with the highest cost of failure and highest probability of faults need to be considered. Below, factors influencing the cost and probability are considered. These factors are called risk categories in the rest of this report. Each of the risk categories is given a weight. Each use case is given a set of values – one for each risk category. The values and weights are given values from one to three where three indicates the highest impact. The value two denotes the normal impact. A description of each category will describe how to give the different values. This is described in section 3.2.3. Area is used below as the functions that need to be executed to follow all the steps in a use case.

3.2.1 Cost factors

There are several ways to find cost factors. These are factors that will cause bigger loss if there are failures for the use cases. Schaefer [7] look at how critical, visible and used each area is. Amland [1] have another perspective; he looks at the consequences the failures have for the vendor or customer. Consequences can be loss of reputation, higher maintenance cost or legal action. A combination of the two views can be used in our method to find the factors affecting cost. Some of the factors are related. A failure in a critical area can lead to loss of reputation. How these cost factors are valuated is discussed in section 3.2.3.

Failure in some areas may be **more critical** than other areas. To get an idea of how critical a failure may be, its consequences must be analyzed. If the customer cannot work with the software and wastes time and money because of the failure, or if it leads to lost or corrupted data, it will give a high cost. If failures in that area cause user to use some workaround to achieve the goal, it is less critical and the cost can be consider medium. An area is less critical if a failure is only hindering the user so it causes workarounds or just makes the program less appealing.

Visible areas are areas where many users will normally experience failures if they are present. The forgivingness of the user should also be considered. Software for untrained or naïve users like the general public, needs careful attention to user interface and has to be more robust than functions only used by expert users.

How often a function is **used** is also important when the cost is considered. Some functions may be used every day, others only seldom. Some functionality is not visible from the outside, but often used, such as process control systems.

Failures in some areas can cause **loss of reputation** for the customer or vendor. A loss of reputation can result in loosing customers and market shares. More critical, used and visible areas as discussed above will normally cause higher loss of reputation. Failures that may be brought up by the media can be damaging for the company

Maintenance will on the average consumes 50-75% of the total cost of development [17]. About half of this goes to developing new functionality, but correcting faults also is a high cost. Areas with many faults may be more expensive to maintain.

Legal consequences can be relevant in some systems. Violating the law can have big consequences for the companies affected, by fines or other forms of legal action. These may for instance be governmental restrictions on privacy and information. Examples of application of such a concern is where personal information is stored or in finance where failures can lead to losses for the company.

3.2.2 Defect generators

The probability of failure increases in areas with more defects. We will look at several factors that can generate defects. In Section 2.3.6, defect generators used by Schaefer [9] are listed. Most of them are discussed here and some extra added. Some of the factors are software metrics related to the code, like complexity, the number of faults found during inspection or earlier testing and changes during development. Other factors are new methods, tools or technology and how many and how qualified the developers that worked on each area are. Factors taking the concentration away from fault removal activities like time pressure and optimization are also discussed. How all these defect generators are valuated is discussed in section 3.2.3.

Complexity is the most important defect generator, but is difficult to assess. The complexity predicts problematic areas in the code and the likelihood for defects increases with higher complexity. About 200 different measures for complexity exist [9], but none of these are validated. We can do several complexity analyses, based on different aspects of complexity. The **size** can indicate the complexity. The size is normally measured by line of code (LOC) excluding comments and blank lines. It is commonly agreed that only executable LOC should be counted. Line of codes is language dependent and dependent on how the developer writes the code. Research has indicated that there may be an optimal program size that can lead to the lowest defect rate [16]. An example of a metric for cyclomatic complexity is **McCabe's Number**. This is calculated from number of simple paths in the program. A guideline of the complexity is that a number between one and ten is a simple program, with low risk. A McCabe's Number between eleven and twenty is more complex, medium risk. A program with a complexity higher than this has a high risk [16]. To find McCabe's number, the internal structure has to be considered.

This is a time consuming operation and unless it is not already done, it is not necessary to do it just to use this method. If there are no metrics available, the developers can just give their assessment of the complexity. Most developers can give a reasonable **estimate** of the complexity as low, normal or high.

Number of faults found during inspection or earlier testing phases can give an indication of where to expect many defects later. When number of faults is used the size should also be considered. An estimate for quality is the number of defects divided on the size of the code. The size can be LOC or functional points, if it is calculated [16]. Using defects found during "*inspection, the determination of a defect is based on the judgment of the inspectors. Therefore, it is important to have a clear definition of an inspection defect"* [16]. If there is an older version of the system that has been tested we can also look at where there were many defects in that version and transfer the experience to new system.

Areas with many **changes** during development may have more defects than other areas. Changes often seem easy, but are often done under time pressure. They often cause problems because they are not analyzed thoroughly. The changed areas may have a bad design from start or the changes may have destroyed the original design. The number of changes done during the development process should be counted. Amland [1] uses **design quality** as one of the important factors indication the probability of fault in his risk analysis. It is measured counting number of Change Requests to the design. The design quality depends on the function, by the design and the application experience of the designer.

Developers using **new methods, tools or technology** may affect the number of faults. The developers may do more errors in the beginning of the process, but fewer later when they learn how to use the new technology. There is always risky to be one of the first in the market using new methods, tools or technology because there are often problems that have not been found yet.

The **number of people** involved can influence the code quality. It is normally better with a small group of highly qualified people, than bigger groups of not so qualified people. The bigger groups of not so qualified people may seem to do as much for the same cost, but they are more likely to do errors that may cause problems later. Areas were **less qualified** people have been employed may need more testing. Often companies employ their best people for the most complex parts so it does not mean that areas were less qualified people have been employed always contain more faults.

Time pressure during development can influence in how well the implementation is done. When there is not much time available, the programmers are more worried about getting the job done than about avoid errors and often skip quality control activities. Overtime may cause people to loose their concentration and they will do more errors when they write the code. Some areas need to keep the machine time and memory use low. These areas must be optimized, which require extra design effort. In areas with much **optimization** or time pressure, resources will be taken away from defect removal activities and results in more faults. As mentioned in section 2.2, inspections and early

testing phases are where most errors are found so taking shortcuts here may lead to extreme failure rates.

There may be **other local factors** for this project that is worth to consider. Communication between people is important. It will suffer if the programmers are distributed geographically. There may also be conflicts between managers that affect number of faults. Important employees may have quit the job. It is also possible to study the testing done in earlier project.

3.2.3 Perform the Risk Analysis

All risk categories may not be relevant to all projects. The risk categories may be too difficult or need too much effort to gather. For each project it is important to decide early in the project what risk categories to gather information about. Some categories may be skipped later when the risk analysis is performed, if the data is not good enough. There is also a limit to how much information that will be useful. If too many categories are chosen, it may be seen as a waste of time and it is too much effort to consider every category thoroughly. This may affect the quality of the collected data.

Some of the defect generators, for instance complexity, are useful to collect just after implementation of each use case. The information may be selected by the testers or the person responsible for developing each use case. The developers have a better understanding of each use case and can easier give information about it. A problem is that some developers may give their use case lower or higher values than other developer would have done. If the testers are active in the process, they may check that the developers use the same measures, thus reducing the measurement errors. The most relevant cost factors and defect generators are given higher weights. When the use cases have been given values for cost and probability, the risk can be calculated.

The **weights** for the **cost factor** and **values** given for each use case for that cost factor are **multiplied**. The sum of these products gives the total cost value. To keep the number low and not dependent on how many factors used, the total cost sum are divided by the number of cost factors used. This will give the average cost for each use case.

Average cost = ∑ (cost (weight)*cost (value)) / Number of cost factors

Cost factors	**Main-tenance**	**Visibility**	**Reputation**	**Total Cost**	**Average Cost**
Weight	*2*	*2*	*3*		
UC1	1	1	2	10 (2*1+2*1+3*2)	3.33 (10/3)
UC2	3	3	2	18 (2*3+2*3+3*2)	6.00 (18/3)

Table 3.1: An example of average cost calculation for the use cases UC1 and UC2

To assess the **probability**, the multiplication between the values and weights is done the same way as for the cost factors. To get a true probability, the sum of the products of the

values and weights for each use case is divided by the highest possible sum a use case can get. In this method, the maximum value that is possible is three so the total probability sum is divided on the sum of each weight multiplied by three. This will not give a real probability for a failure, but a number that indicate if there is a high or low probability of failure. If the number is close to 1 the probability of failure is high.

Probability = ∑ (defect generator (weight) * defect generator (value)) / ∑ (defect generator (weight) * defect generator (max value))

Defect Generators	Complexity	Changes	New techn.	Total sum	Probability
Weight	*3*	*1*	*2*		
UC1	2	1	2	11 (3*2 + 1*1 + 2*2)	0.61 (11/18)
UC2	1	2	2	9 (3*1 + 1*2 + 2*2)	0.50 (9/18)

Table 3.2: An example of calculation of the probability for the use cases UC1 and UC2. The total sum is divided on 18 found by multiplying each weight by three (3*3 + 1*3 + 2*3)

From the average cost and the probability, we can find the **risk exposure** for each use case as follows:

Risk Exposure = Average cost * Probability

When the risk exposure is found for all the use cases, this is collected in a in a list **sorted** by risk. The method will continue by finding test cases for the use cases with the highest risk exposure.

	Cost	Probability	Risk Exposure
UC2	6.00	0.50	3.00
UC1	3.33	0.61	2.03

Table 3.3: The Risk Exposure is calculated. UC2 has a higher risk exposure than UC1.

3.3 Design test cases

Test cases need to be designed. Kaner [4] describe a test case as a question you ask the program to gain information. A test case may not be too specific, more like an idea on what to check in the program. The important thing is that the test cases are capable of revealing information. They are not necessarily designed to expose a defect. I will define a test case as a set of instructions designed to detect a particular class of defect, by bringing about a failure [13]. The execution of each test case can result in many tests. A test case consists of test item, input and output specification and environmental needs.

According to Stålhane [5] it is important to think testing from the beginning of the project. *"As soon as anybody state a requirement, somebody must be responsible fro making a test for it. A requirement without a test will be ignored and forgotten"* [5]. This is also important in the Rational Unified Process, where testing should start as early as possible. [15] Thus, test cases are often produced early in the process and usually before the risk analysis. Since generating test cases is an important part of our method, and use cases are used as a starting point for the risk analysis, we will describe how to generate test cases.

The test cases can be generated from the use cases. Haumann [15] describe a method using use case scenarios. A use case scenario *is "an instance of a use case, or a complete "path" through the use case"* [15]. All possible use case scenarios are generated using all the alternate paths. For each use case scenario, at least one test case that will make it execute are identified. Without test data, the test cases can not be executed or implemented. *"Therefore, it is necessary to identify actual values to be used in implementing the final tests"* [15].

The resources for each test case are estimated i.e. how much time in minutes each test case will need to run. It is important to include the set up time and the time needed to clean up data bases afterwards. Each test case must be connected to a use case and each use case may have many test cases. To use the risk analysis from the previous step to prioritize the test cases, each test case has to be related to one use case. In cases were a test case cover more than one use case, the most relevant use case is chosen. In table 3.4 the test cases TC1 and TC2 are connected to use case UC1, while the test cases TC3 and TC4 are connected to the use case UC2. How this is used to prioritize the test cases is discussed in the following section.

3.4 Consider the importance of each test case

Some test cases may be more important for the use case than others. Each test case is thus given low, medium or high importance for that use case. This value, together with the risk exposure for the use case it belongs to gives the value for each use case. The values given for the importance depend on what we want to prioritize; the use case risk or the importance for the test case. In table 3.4 low is given 0.8, medium is given 1.0 and high is given 1.2, this will adjust the risk exposure for the use case with 20 %. Other values may be used for each analysis. The product of these values and the risk exposure is called test case risk. The list of all test cases is sorted by the test case risk.

	Risk Exposure	Importance	Test Case Risk
TC1 (UC1)	2.03	High (1.2)	2.44
TC2 (UC1)	2.03	Medium (1.0)	2.03
TC3 (UC2)	3.00	Medium (1.0)	3.00
TC4 (UC2)	3.00	Low (0.8)	2.40

Table 3.4: An example of calculation of Test Case Risk: TC1 has higher Test Case Risk than TC4, because it has high importance, even though the Use Case it tests has a higher risk exposure.

3.5 Run the test cases

The test cases are run in the order defined in step 4, with the test case with the highest risk first. It is important to keep track on the test status: test cases that have been run and how much resources are used and what is still available. When a test case has been run and accepted, it is removed from the list and the next test case are run.

	Test Case Risk	Time Estimate	Time left before execution
TC3 (UC2)	3.00	7	20
TC1 (UC1)	2.44	12	13
TC4 (UC2)	2.40	6	1
TC2 (UC1)	2.03	6	

Table 3.5: An example of running of the test cases: There is only enough time to run TC3 and TC1.

In this example it would be possible to run both TC4 and TC2 instead of TC1. It would be a possibility to add a time factor to favour test cases with low running time, but to make the method less complicated this is not done. If it is assumed that the test cases does not have very different time estimate this will not be a problem. Another way to punish less efficient test cases is to give low importance for TC1 because you know it is time consuming. Gerrard [10] propose another approach. *To "document all of your assumptions carefully, then allow management to choose whether to take the risk or pay the price. This is a decision to be made by management, not you. You never know, this time they might just fund the test"* [10]. Also in this method, documenting every done is important.

The method has now prioritized the test cases using risk. The test cases are generated from the use cases. A risk exposure for each use case has been calculated from cost factors and risk generators. The risk exposure for the use case is multiplied with the importance for connected test case. This will give a test case risk value. The test cases with a high test case risk will be prioritized.

4. The Software Tool

The method described in the previous section is intended to be implemented as a software tool. The use cases and test cases and their related data are stored and can be updated during the project. Relevant areas of cost and probability of failure are chosen and given weights. The program will identify the most critical test cases.

4.1 How will the software tool help the tester?

The software tool will help the testers to implement a risk-based approach to testing. It helps them to collect the necessary information to perform a risk analysis. This will again give them an easier choice on which tests to prioritize. The software tool will also keep track on what to test and the status for each test during testing.

The most important point is that the software tool will help to implement a testing process that focuses on risk. As discussed in section 2.2, there are many benefits to be gained by turning the focus in testing towards risk. When the management wants an early release, the testers will have arguments for more testing. Using the software tool, the testers can explain and show the managers the risk of not to do more testing.

4.2 Possible problems

During the development process, there may be many changes to the requirements. New requirements may be included or old requirements may be changed. It is important that the data in the software tool is updated. It will not store all information needed for the testing. There is traditionally a lot of documentation on testing. If this should be stored in the program, we may end up with a lot of redundant information. That is why only use case and test case names are stored, not the whole descriptions.

Too much information can work against its intention, to help the process. The risk analyses may not be motivating. It can be difficult to gather accurate data. The uncertainties may be so big that the risk analysis does not seem use full. The costs of using the method and software tool can seem to be too high. Hopefully the benefits will be higher than the cost of using the software tool. The problems caused not to identify the functions with highest the risk, will probably more expensive than the extra effort using the software tool.

5. Specification of requirements

The software tool is not intended to support every aspect of testing. It will be a help the tester to gather relevant info and perform a risk analysis so that the test cases can be prioritized.

5.1 Functional requirements

The functional requirements for the software tool are described and use cases are used to clarify them.

5.1.1 Create a new project

When the testers are involved in the process they need to start a new session for that project in the software tool.

Name:	**Create new project**
Actor:	Tester
Summary:	A new project is started and the user wants to add the project.
Basic Course of Events:	1: The user creates a new project 2: The user gives the project a name
Alternative Path:	The user can select an existing project and give it a new name
Trigger:	A new project is started

Table 5.1: Use Case 1, Create new project

Requirements:
1: The user must be able to create a new project
2: The user must be able to give the project a name
3: The user should not be able to create two projects with the same name
4: All data concerning the project can be saved
5: The project should be possible to load for each time the program start

5.1.2 Create a new risk category

The risk categories are the cost factors and the defect generators. These are stored so the categories used in earlier project are possible to use. Each category needs a short description to make it is easy to know what the category means.

Name:	**Create a new risk category**
Actor:	Tester
Summary:	The user wants to add a new cost factor or defect generator.
Basic Course of Events:	1: The user selects what kind of risk category to create - a cost factor or a defect generator 2: The user creates the selected category 3: The user gives it a name 4: The user inserts a description of the risk category 5: The user inserts a default weight, 1, 2 or 3
Alternative Path:	In step 1 the user can chose an existing category, and edits that category instead, the user continues in step 2.
Exception Path:	If the user gives the new category a name that already exist in step 1 the user has to give it another name.
Trigger:	The user needs a new category for the risk analysis

Table 5.2: Use Case 2, Create a new risk category

Requirements:
6: The user must be able to select what kind of risk category to create
7: The user must be able to create a new risk category
8: The user must be able to give the risk category a name
9: The user must be able to insert a short description of the risk category
10: The user must be able to give a weight of 1, 2 or 3 to the risk category
11: The user must be able to edit the data of an existing category

5.1.3 Select risk categories

The risk categories to use in the risk analysis are chosen. It is important that the tester decide what risk categories to select before the data collection starts. If some categories are not relevant they can be removed from the risk analysis later.

Name:	**Select risk categories**
Actor:	Tester
Summary:	The tester has created a new project and wants to select cost factors and defect generators to be included in the risk analysis. Each of these is given a weight.
Basic Course of Events:	1: The user selects a category to use in the risk analysis 2: The user gives the category a weight, 1, 2 or 3
Trigger:	A new project is created

Table 5.3: Use Case 3, Select Risk category

Requirements:
12: The user must be able to select the risk categories to be used in the risk analysis.
13: The user must be able to give the category a weight, 1, 2 or 3.

5.1.4 Add a new use case

The use cases are normally designed during the functional requirement. When they are written they should be to the tester so they can be added to the software tool.

Name:	**Add a new use case**
Actor:	Developer or tester
Summary:	The user wants to add a new use case that is part of the system. The user can be the developer or the tester.
Basic Course of Events:	1: The user creates a new use case 2: The user gives the use case a name 3: The system saves the use case
Alternative Path:	In step 1 the user can select an existing use case so the name can be change in step 2
Exception Path:	If the user gives a name that already exist in step 2 the user will be prompted for a new name.
Trigger:	The use cases are ready to be included in the program

Table 5.4, Use Case 4, Add a new use case

Requirements:
14: The user must be able to create a new use case
15: The user must be able to give the use case a name
16: Two use cases can not have the same name
17: The system must be able save the use case
18: The user must be able change a name of an existing use case

5.1.5 Give values to a use case

The user gives values to the cost factors and defect generators for each use case. The risk is calculated as described in section 3.2.3.

Name:	**Give values to a use case**
Actor:	Developer or tester
Summary:	The user wants to store values for the cost factors and the defect generators for the use case.
Basic Course of Events:	1: The user selects an existing use case 2: The user selects a risk category 3: The user gives the value 1, 2 or 3 for the category 4: The system saves values for the use cases 5: The system calculates the risk and shows all the use cases sorted according to risk
Loop:	After step 3 the user can go back to step 2 and give values to other categories.
Alternative Path:	The user can chose to read the description of the use case after step 2. The user then continue with step 3
Trigger:	The user wants to give one or many values for a use case

Table 5.5: Use Case 5, Give values to a use case

Requirements:
19: The user must be able to select a category for a use case and give it a value, 1, 2 or 3.
20: The system must be able to save the value given for the use case
21: The user must be able to give values to all the categories for the use case
22: The system must give a list of all use cases that have not yet been given values
23: The system must give a list of all use cases, ordered by risk
24: The user must be able to see the description about a risk category

5.1.6 Add new test case

When the test cases are generated the name, importance and time estimate is stored in the software tool. The value for importance is multiplied with the risk for the use case as described in section 3.4.

Name:	**Add new test case**
Actor:	Tester
Summary:	The tester wants to add a new test case.
Basic Course of Events:	1: The user creates a new test case 2: The user gives the test case a name 3: The user chose what use case the test case is related to 4: The user gives the test case a low, medium or high importance 5: The user gives a time estimate for the test case execution in minutes 6: The system saves the test case data 7: The system returns a list of all test cases, together with a calculated risk
Alternative Path:	In step 1, the user can select an existing test case and edit the data.
Exception Path:	If the user gives a name that already exist in step 2 the user is prompted for a new name.
Trigger:	The user has created a new test case and wants to add it in the system.

Table 5.6: Use Case 6, Add new test case

Requirements:
25: The user must be able to create a new test case
26: The user must be able to insert a test case name
27: Two test cases can not have the same name
28: The user must be able to select a low, medium or high importance for the test case
29: The user must be able to insert a time estimate for the test case in minutes
30: The system must be able to save all the data about the test case
31: The system must return a list of all test cases with a calculated risk
32: The user must be able to change the name, importance level and time estimate for an existing test case

5.1.7 Change status for a test case

Each test case has a status with the default “not run”. When the test cases are run the status has to be changed “run”.

Name:	**Change status for a test case**
Actor:	Tester
Summary:	The user changes the status for a test case and updates the time estimate.
Basic Course of Events:	1: The user selects a test case 2: The user change the status for the test case 3: The user updates the time estimate 4: The user gets a list of all the test cases which have the status “not run” sorted by risk.
Alternative Path:	The user can skip the step 3, if there is no change. The user can start at step 3 and only change the time estimate
Trigger:	The tester has executed a test case

Table 5.7: Use Case 7, Change status for a test case

Requirements:
33: The user must be able to select a test case to change the status
34: The user must be able to change the status for the test case, not executed and executed
35: The user must be able to update the time left for testing
36: The user must get a list of all the test cases with the status not run sorted by risk

5.2 Non-functional requirements

Non-functional requirements can describe quality, the environment and technology to be used, the project plan and the development methods.

ISO 9126 [19] defines a set of software quality characteristics - functionality, reliability, usability, efficiency, maintainability and portability. Usability is the most important of these characteristics for this software tool. It must be easy to use and learn and use. Reliability is also, however, important, but not prioritized here.

The software tool will be implemented using MS Visual Studio .NET.

6. Implementation

There was not enough time to finish the software tool. I have thus prioritized to implement a prototype to show how the software tool is supposed to work. In the prototype the inputs are not saved every time the application is run. There is also some functionality that is not working. Some examples of the software tool in use and screen shots are shown below.

6.1 Select risk categories

Before the risk analysis can start the most relevant cost factors and defect generators need to be selected. Figure 6.1 show the dialog box where the cost factors are selected. The check boxes left of the cost factors show if the cost factor is chosen for the risk analysis. *Critical* and *Visible* are chosen, while *Maintenance cost* is not chosen in this example. It is also possible to create a new risk category and to edit or delete an existing risk category.

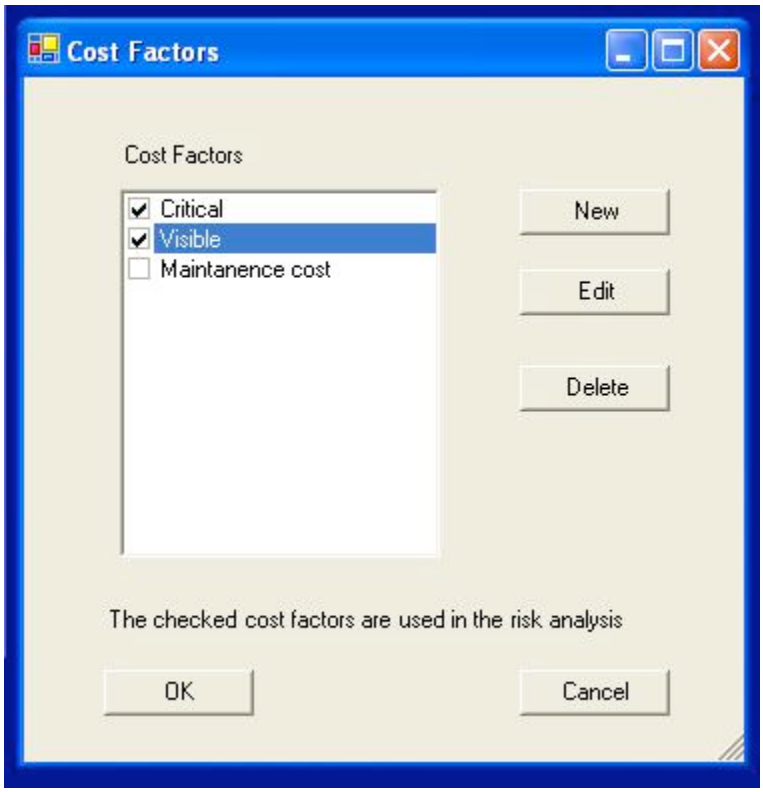

Figure 6.1: Example of Cost Factors dialog box

6.2 Create a new risk category

To create a new risk category, the button *new* is clicked in the *Cost Factor* dialog box, in figure 6.1. Figure 6.2 illustrates a new risk category dialog box. A name, a description and a weight from one to three are given. The description can help the tester to know what kind of values to give. *OK* will save the new category and close the dialog box, while *Cancel* will just close the dialog box. A similar dialog box is used to edit an existing category.

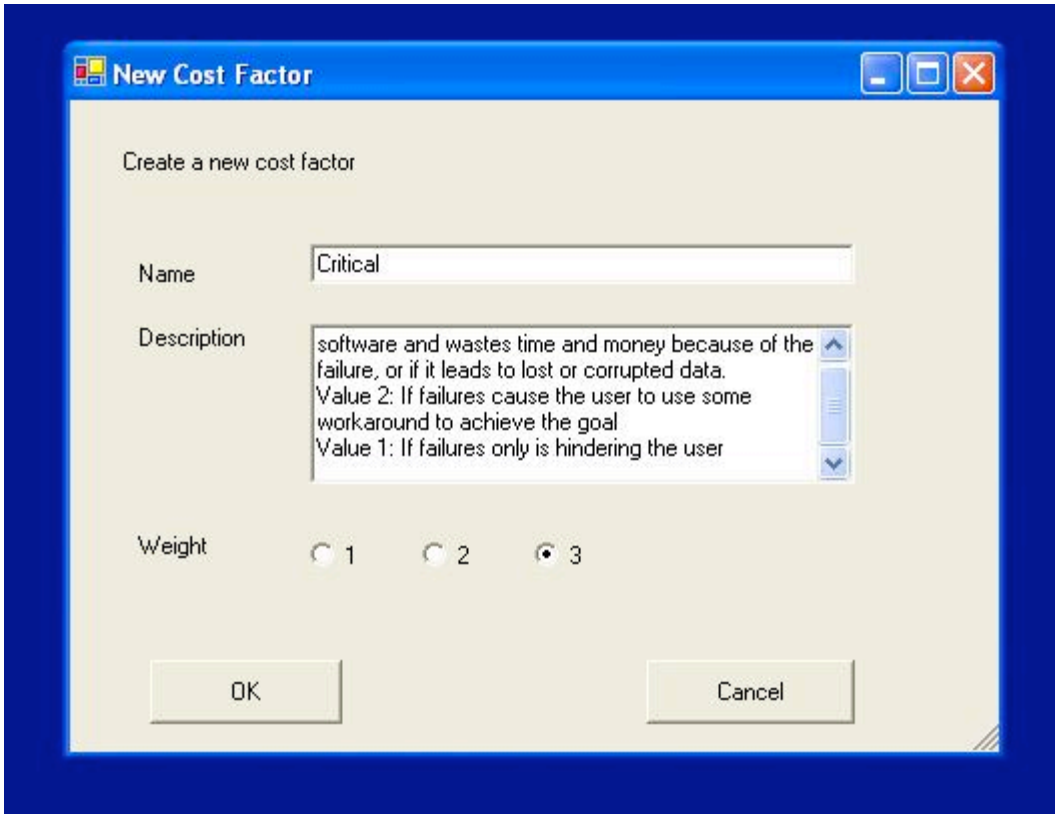

Figure 6.2: Example Create new cost factor dialog box

6.3 Create a new use case

The user has to add the use cases. In this example use case scenarios are chosen, but in a real tool a full use case should be used. A use case scenario “Create a New File” is created in figure 6.3. The description is supposed to help the user to know what the use case does when it is used later in the risk analysis. The button OK will create the new use case scenario and go back to the dialog box in figure 6.5. Cancel will go back, without creating the use case scenario.

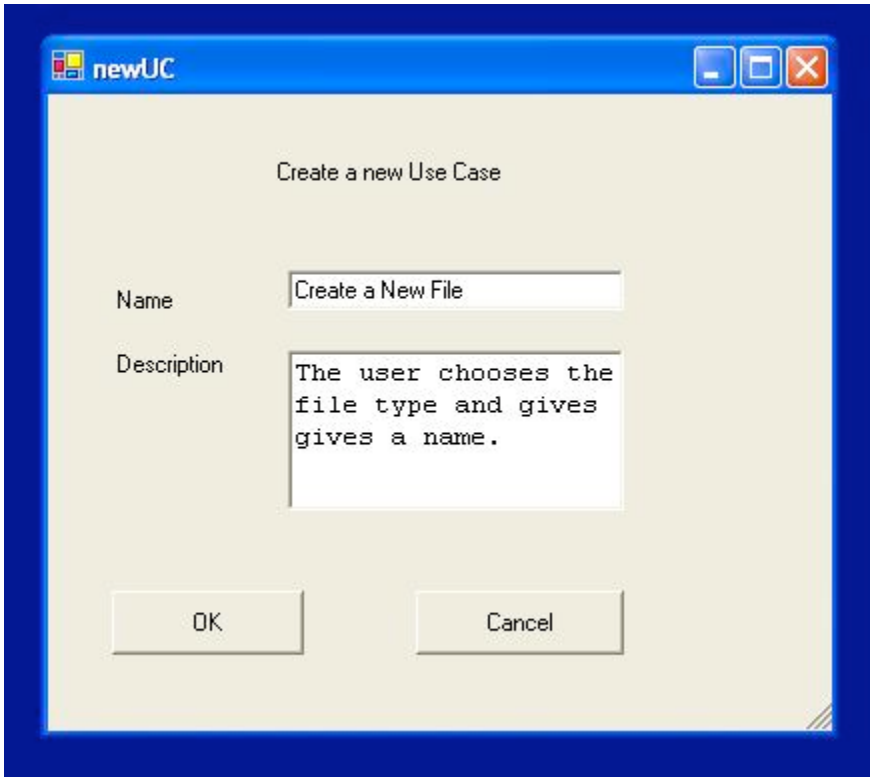

Figure 6.3: Example of create new use case dialog box

6.4 Give Values

Each use case scenario is given a set of values one for each chosen risk category. Figure 6.4 show an example where we give a value to the use case scenario “Open file” for the risk category “changed”. A description of the risk category is provided in the text field. This will help the user to know what value to give for the category and explain what the risk category means. The button *next* will lead to a dialog box with the next risk category for the use case scenario.

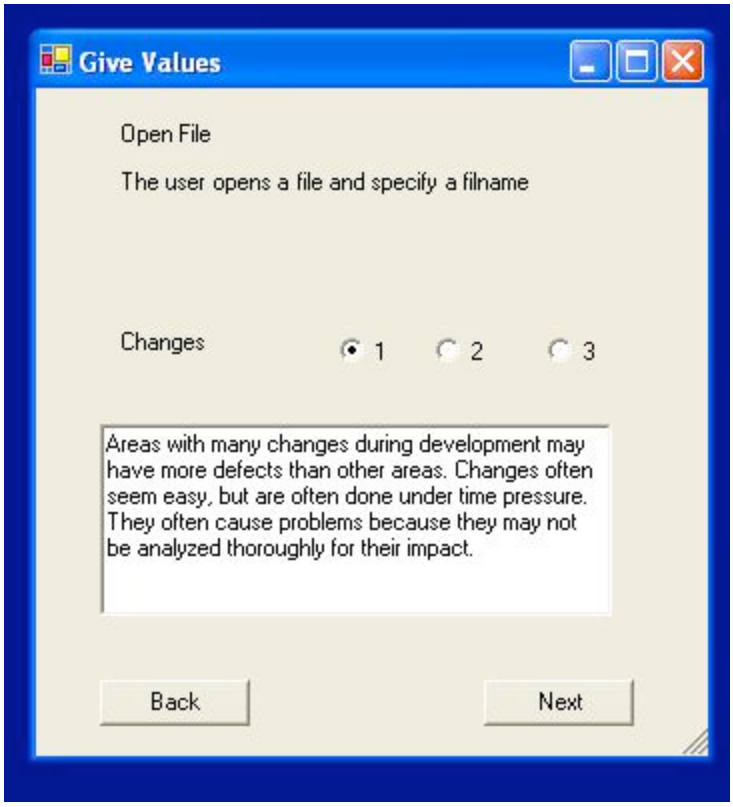

Figure 6.4: Example of give values to a use case dialog box

6.5 Show use case list

When values for each risk category are given to a use case, the risk is calculated. In Figure 6.5, “Save File As” and “Open File” has been given values. “Open File” has the highest risk. To give values to the other use cases, one of the use cases is selected and the button *Give Values* is pressed. The dialog box in figure 6.4 is opened. An existing use case can also be edited or deleted.

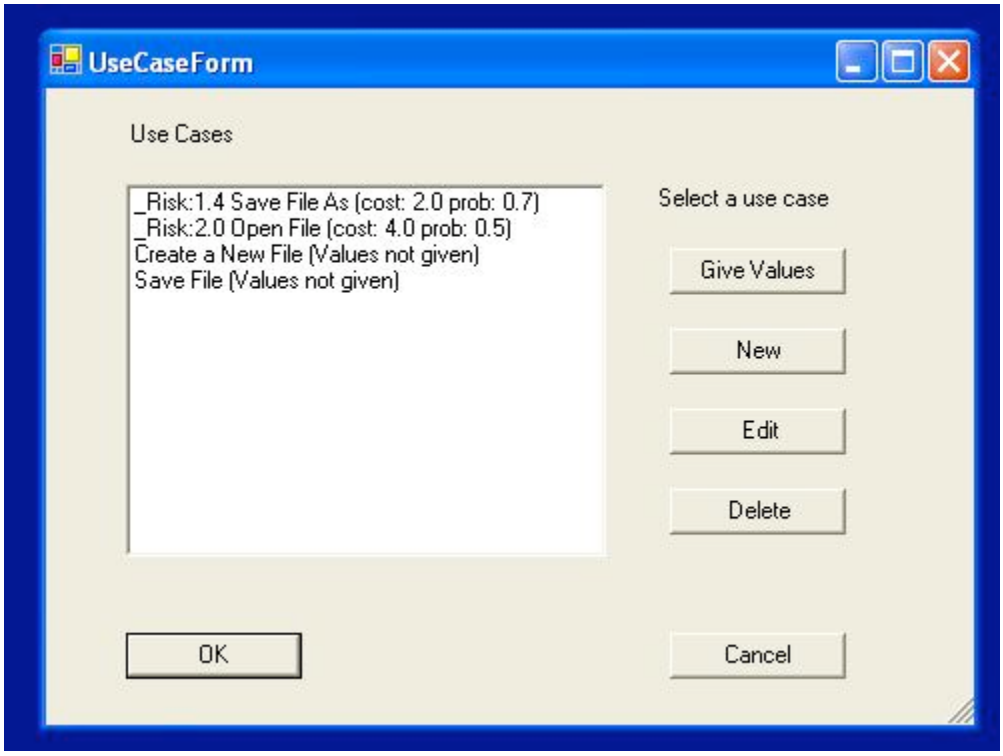

Figure 6.5: Example of use case list dialog box

6.6 Create a new test case

Figure 6.6 shows an example where the user creates a new test case. A name and estimated time must be given. The user must also select the use case it is connected to. The level of importance, low, normal or high is given.

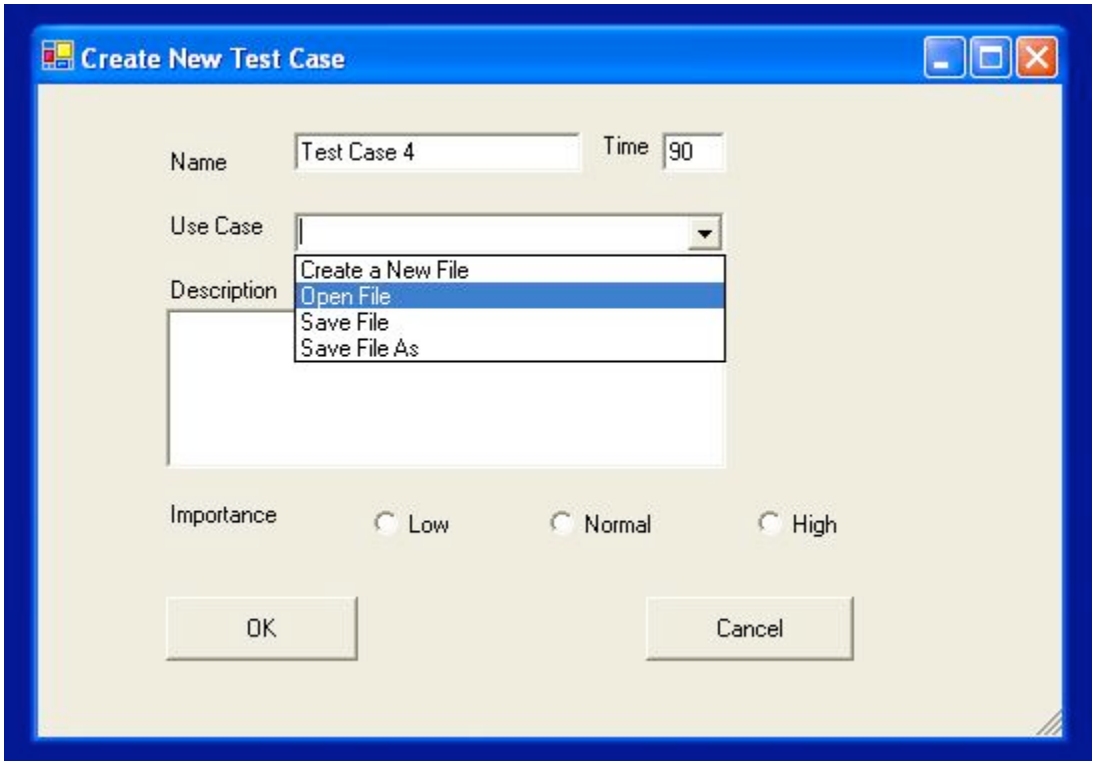

Figure 6.6: Example of Create New Test Case dialog box

6.7 Show test case status

Figure 6.7 show an example of the test case status dialog box. This will be used during testing to keep track of the status and the resources. The test cases are ordered by risk, and the test case with highest risk is supposed to run first. The functionality for this dialog box is not implemented in the current version of the prototype.

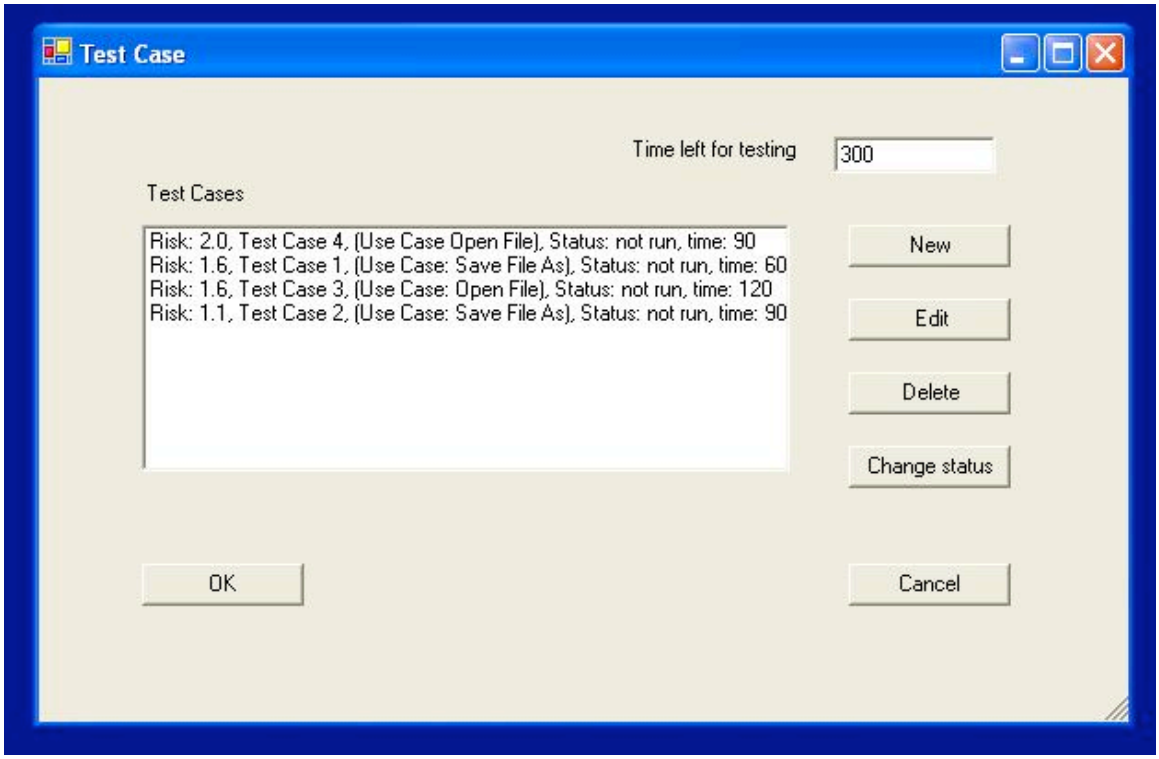

Figure 6.7: Example of Test Case dialog box

7. Conclusion and Suggested Further Work

In this report I present a literature review on Risk-Based Testing. There are several ways to do Risk-Based Testing, from using risk to find the test, to using the risks as a way to prioritize what areas to focus during testing. The choice was a method to prioritize use cases in a risk analysis. Test cases are designed from the most risky use cases and resources for testing are estimated. This method was chosen because it seemed to be a good way to prioritize the tests. It was also a suitable method to use in a software tool to support the testers. How the software tool can help the tester and possible problems were discussed. The requirements describe the software tool. These may later be extended for more functionality, if needed. There was not enough time to implement the tool, but a prototype with some of the functions is presented.

It is too early to tell if the software tool will be really useful in a testing process. Further work includes implementation of a full version of the software tool. The software tool must be tried in a real software project to check if it is useful and if any improvements to the method or the software tool are needed.

8. References

[1] Ståle Amland, Risk Based Testing and Metrics, November 1999, http://www.amland.no/risk-based.htm

[2] James Bach, Heuristic Risk-Based Testing, November 1999, http://www.satisfice.com/articles/hrbt.pdf

[3] James Bach, Risk and Requirements-Based Testing, IEEE Computer Society, June 1999, http://www.satisfice.com/articles/requirements_based_testing.pdf

[4] Cem Kaner, What is a Good Test Case, May 2003, http://www.kaner.com/pdfs/GoodTest.pdf

[5] Tor Stålhane, Risk Based Testing 2 (slides), 2004

[6] Tor Stålhane, Risk Based Testing 1 (slides), 2004

[7] Yanping Chen, Robert L . Probert, A Risk-based Regression Test Selection Strategy, 2003, http://portal.acm.org/citation.cfm?id=782116&jmp=abstract&dl=GUIDE&dl=ACM

[8] Stephane Besson, A Strategy for Risk-Based Testing, StickyMinds.com, August 2004, http://www.stickyminds.com/sitewide.asp?ObjectId=7566&Function=DETAILBROWSE&ObjectType=ART

[9] Hans Schaefer, Strategies for Prioritizing Tests against Deadlines Risk Based Testing, Undated, http://home.c2i.net/schaefer/testing/risktest.doc

[10] Paul Gerrard, Risk-Based E-Business Testing, 2002, ISBN 1580533140, page 3 – 29 and 51 - 80

[11] Proceedings of the STANZ conference 2002, Wellington 25.-26. Nov 2002

[12] Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Main_Page , Nov 2004

[13] Timothy Letherbridge, Object-Oriented Software Engineering, 2002, ISBN 0-07-122689-3, page 110 – 124, 234, 373 – 375

[14] Martin Fowler, UML Distilled Second Edition, 2000, ISBN 0-201-65783-X, page 39- 47

[15] Jim Heumann, Generating Test Cases from Use Cases, 2001, http://www.therationaledge.com/content/jun_01/m_cases_jh.html

[16] Stephen H. Kan, Metrics and Models in Software Engineering, second edition, 2003, ISBN 0-201-72915-6, page 119

[17] Tor Stålhane, Gunhild Sivertsen Sørvig, Risk Analysis as a Prioritizing Mechanism in SPI, EuroSPI 2003

[18] Hans Van Vliet, Software Engineering Principles and Practice, 2000, ISBN 0-471-97508-7, page 14 - 17

[19] Draft for CD, ISO/IEC 9126-1, Sep 2004, http://www.idi.ntnu.no/~stalhane/tdt4235/docs/9126-utdrag.DOC

Glossary

Black box testing: A kind of testing technique where inputs are given and the out puts are observed. The technique focuses on the requirement and is also called functional testing.

Defect: See fault

Dialog box: A window in a graphical user interface that requests information from the user.

Error: Is the action of writing incorrect code

Failure: "Lack of ability of a component, equipment, sub system, or system to perform its intended function as designed. Failure may be the result of one or many faults" [12].

Fault: "An abnormal condition or defect at the component, equipment, or sub-system level which may lead to a failure" [12].

Functional requirements: "describe what the system should do" [12]

Non-functional requirements: "constraints that must be adhered to during development. They limit what resources can be used and set bounds on aspects of the software's quality; thus they tend to restrict the freedom of software engineers as they make design decisions" [12].

Requirement The requirements describe what the system should do, but do not say how those requirements should be implemented.

Risk is a possible future, unwanted event that has negative consequences.

Risk Analysis Risk analysis is a technique to identify and assess factors that may jeopardize the success of a project or achieving a goal. This technique also helps define preventive measures to reduce the probability of these factors from occurring and identify countermeasures to successfully deal with these constraints when they develop. [12]

Risk Exposure: cost of event multiplied by the probability the event

Scenario describes a sequence of interaction between one or more users and the system.

Test Case set of instruction designed to detect a particular class of defect, by bringing about a failure [13].

Testing is a process used to identify the correctness, completeness and quality of developed computer software [12].

Use Case “A way in which the system can be used, described as a step-by-step sequence of actions, along with the system’s response and certain other information. “[13]

Use case scenario is an instance of a use case, or a complete "path" through the use case [15].

White box testing: checks “that the outputs of a program, given certain inputs, conform to the internal design and implementation of the program” [12].