Development and Evaluation of Tool Support for Retrospective Analysis in Requirements Engineering

Master Thesis at
Department of Communication Systems
Software Engineering Research Group
Lund Institute of Technology
Lund University

Mikael Jönsson
Per Klingnäs

Supervisors
Lena Karlsson
Björn Regnell

Abstract

Release planning is an essential part of the development process in market driven software development. To obtain a successful release plan for a specific software system, prioritizations must be done to decide which requirements shall be implemented in each release. The prioritization is typically done after criteria as expected customer value and estimated cost of implementation. The outcome of the prioritizations does not always correspond to the true outcome, which can relatively easy be measured when the software has been implemented and available on the market for some time. To improve the release planning process, the sources of these inaccurate prioritizations can be identified, by using retrospective analysis, and hopefully adjusted. One such type of retrospective analysis is the PARSEQ method, developed at the Department of Telecommunication Systems at Lund University. The PARSEQ method is divided into four steps where each step in some extent can be supported by a tool, but the method as a whole lacks this possibility. Because of the inexistence of such a tool, the process is considered impractical and time-consuming and therefore hard to test and evaluate in real development processes.

The purpose of this thesis was to develop a tool that supports all four steps of the PARSEQ method. The tool should then be evaluated to examine if it is appropriate for the PARSEQ method and fully supports all steps.

The result of the development work is a program written in Java in size of about 10.000 lines of code. All four steps of the PARSEQ process are supported, including three different kinds of prioritization methods. Requirements can be imported to and exported from the program to allow an interface with possibly already existing requirement management tools in the development process. An evaluation of the project’s development process was done using the PARSEQ method with the tool. This evaluation gave some minor improvement suggestions concerning the project’s requirement management but no serious errors were found.

The evaluation of the program was done during a user evaluation with users very familiar with the PARSEQ method. The outcome of this evaluation was on the whole positive. Some parts of the process were found less time-consuming and more flexible than without the tool. Other parts were considered to be in need of some further program development to achieve an even better support for the method.

Acknowledgements

We would like to thank our supervisors Björn Regnell and Lena Karlsson, at the Department of Telecommunication Systems at Lund University in Sweden, for the opportunity to do this thesis. Particularly, we would like to thank Lena for all her support, throughout the project, in everything from answering questions and helping us gather information to being a constant source of feedback on both the development of the tool and this report.

Contents

1 Introduction

This report is a part of the result of a Master Thesis done by two Master of Science students at the Department of Telecommunication Systems at Lund University in Sweden. The work was done over a period of 20 working weeks during the spring semester in 2005 and corresponds to 2 times 20 credits.

At the time of this thesis, the authors were both studying at the Computer Science and Engineering program with software systems as area of specialisation. The authors also have similar interests in the development process and requirements engineering field.

1.1 Background

In market driven software development, release planning is an essential part of the development process. Software is almost all the time produced and delivered in several releases, each one with some changes and improvements made since the last release. For each release, a selection of which functionality and features that shall be implemented, expressed as requirements in the requirement specification, must be done. The decisions made during release planning do not always turn out to be the most appropriate after the release has been on the market for some time. Features considered to be very essential during the early decision phase might for example be rated as unnecessary when the customers have used the product. By understanding the inappropriate decisions and why they were made, it is possible to identify potential improvements to the release planning process.

One method of understanding how inappropriate release planning decisions are made is the retrospective analysis. The core of a retrospective analysis is to, by evaluating earlier work, gain knowledge from the past in order to improve the process in the future. At the Department of Telecommunication Systems at Lund University, a retrospective analysis method called PARSEQ (Post-Release Analysis of Requirements SElection Quality) has been developed. The method consists of four main steps described in detail in this report. When a PARSEQ evaluation is performed today, each step can individually be supported by various tools, but there exist no tool that supports the process as a whole. The lack of such a tool makes the PARSEQ process in practice impractical and time-consuming. This inconvenience means that the process is hard to test and use in real development processes. To make PARSEQ faster and easier to use, a computer-based tool is therefore needed.

1.2 Objectives

The goal of the thesis is, from a given requirements specification, to develop and evaluate a tool for retrospective analysis of requirements. The developed tool shall give support for all the four steps in the PARSEQ method: *Requirements sampling*, *re-estimation of value and cost*, *root cause analysis* and *elicitation of improvements*.

1.3 Limitations

The tool that was to be developed would have many similarities to other requirements engineering tools, but its purpose is not to store and manage large amounts of requirements.

There is a line between the PARSEQ method and the objective of this thesis. The purpose of this thesis was not to evaluate the PARSEQ method, but to build and evaluate the tool for it.

1.4 Outline of the report

After this first section, three sections follow that cover software engineering theories relevant to this thesis. In section 2 Software engineering, a summary of general software engineering concepts is done. The intention of this is to give the reader a broad picture of the area of software development in general. In the next section, 3 Requirements engineering, a more detailed description of requirements engineering is done to describe the part of the software engineering that is most relevant in this thesis. Finally in section 4 Retrospective analysis, the group of specific process improvement methods in general and the PARSEQ method in particular that is the ground for this work are described.

In section 5 Method, the implementation and evaluation methods used during the work is described. This chapter is followed by the results from the implementation and evaluation in section 6 Results. In section 7 Analysis, an analysis of the results achieved is done. A discussion concerning the result of the work and proposed future further development of the tool can be found in section 8 Discussion.

In section 9 Conclusions, the conclusions drawn from the work are presented.

In section 10 References and 11 List of figures and tables, the references used during the work can be found, together with a list of figures and tables.

In the last section of the report, 12 Appendix, additional information and documents concerning the work is located.

2 Software engineering

According to Sommerville [1] software engineering is:

“...an engineering discipline which is concerned with all aspects of software production from the early stages of system specification through to maintaining the system after it has gone into use”

By referring to all aspects of software Sommerville means that software engineering not only concerns the technical processes involved in software development. Other important areas included can for instance be project management and development of tools and theories. In other words, software engineering is the engineering approach to all the activities and processes present during software development.

2.1 Software process

A specific set of these activities constitutes what is called a *software process*. Sommerville [1] defines the software process as a set of four specific, fundamental process activities.

1. *Software specification* is the activity where the functionality of the software itself and the constraints of its operation shall be stated.
2. During the *software development* the software that meets the demands in the specification is produced.
3. When the software has been produced a *software validation* has to be done to ensure that the demands have been fulfilled.
4. Due to changing customer demands a *software evaluation* must be done to meet new needs.

All these activities are in one way or another present in every software development project. A big difference in organizing the activities exists and also different levels of detail are used when describing the activities. One project’s way of organizing the activities might be unique while some other ways might be widely spread, depending on how suitable it is to adapt that particular way of organizing to a specific project. Also parameters as timing of the activities as well as results and outcomes from each activity might differ from project to project. Sommerville [1] point out that if an inappropriate process is used, a probable reduction of quality and usefulness of the software project will be introduced.

2.2 Software process models

Some processes have been proved to be more suitable than others for some types of projects. In order to make it easier to use these specific types of processes, models are created. A software process model is therefore a simplified description, an abstraction, of a specific software process [1]. In addition to the fundamental activities mentioned above, a model can include other important components of the software development. An example of such an important component can be the roles of people involved in the project.

According to Sommerville [1] the type of models can vary depending on their focus. Some models are described on the basis of the flow of work throughout the process. In this case the activities in the model are represented by actions taken by humans. In other models the activities are represented by data-flow and a bigger focus is on how input is transformed during the process into output. A third kind of viewpoint is to focus on the actors' different roles in the project and the activities for which they are responsible. The choice of model type and what to focus on might differ from time to time according to the specific nature of the current project.

Software process models of today can be categorized into different generic groups [1]. Each of these groups represent one approach to developing software. The groups can be seen as paradigms for a process from a particular perspective of today's software development.

1. *The waterfall approach* represents a model where each activity is completed before the next one can start. A waterfall with no option to go back separates the steps from each other.
2. In *evolutionary development* the idea is to let a rapidly developed initial system grow by continuous customer input. This means that the main activities of the process must be interleaved and repeated in relatively short iterations.
3. In a *formal transformation* model the objective is to convert a formal mathematical system specification into a program by using mathematical methods. If the transformation is succeeded, it can be proven that the developed program meets its specification requirements.
4. Another general model take for granted that parts of the system already exists. By integrating these parts it is possible to do a *system assembly from reusable components*.

Besides these generic groups of process models, other types of models have been developed. An example of one such type of model is hybrids of two or more of the groups presented above.

To create a greater understanding of the main parts of and differences between various kinds of models, a few of them will be presented and illustrated more in detail. First the most fundamental and earliest process model, the waterfall model, will be presented to create an understanding of the origin of process modelling. The next model to be presented is the one that best describes the software process in this project, the evolutionary development model. Finally an example of a hybrid model will be given to show how the process model evolution has resulted in some of today's most commonly used models.

2.2.1 Waterfall model

The simplest way to model a software process is to consider the fundamental activities as independent steps in a one-way proceeding process. This is exactly what is done in the waterfall model. This model assumes that one activity in the process must be fulfilled before an irreversible step can be taken to the next activity. To form a process after this model is very inconvenient. Problems and mistakes discovered during a late phase of the project can be hard to deal with due to the lack of possibility to go back and modify. According to Lauesen [2] the whole idea of strict phases of the waterfall model is wrong. Lauesen means that the model only represents an ideal and that real projects cannot be carried out this way. This means in practice when the waterfall model is used, the stages will have to overlap each other to make information feedback possible [1].

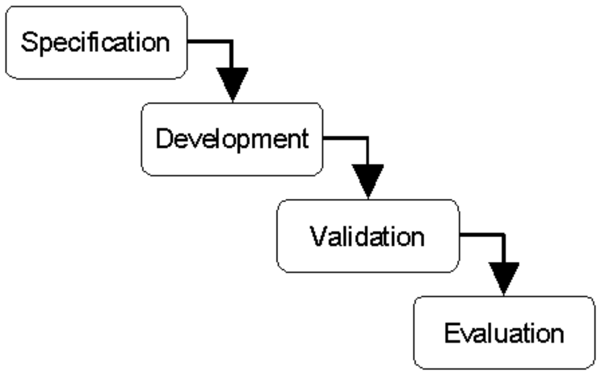


Figure 1 The Waterfall model

2.2.2 Evolutionary development

The base of evolutionary development is to early in the process create an initial prototype implementation. By exposing this prototype to the customer, the system will be developed by adjusting and adding features due to the comments collected [1]. This procedure then repeats throughout the project time and the result is a continuous evaluation of the system. Doing specification, development and validation sequentially as in the waterfall model is not possible in this method. Instead these activities must be carried out in parallel and with instant feedback and updates between each other.

The main advantages of the evolutionary development lie in the possibilities to develop the specification of the customer requests incrementally. By doing this throughout the whole process, a system that meets the immediate needs of the customer can be created. Another advantage within the same domain is that the user develops a better understanding of their problem, which can be directly reflected in the outcome of the project [1].

In contrast to the advantages mentioned above, especially three problems have been identified in the model [1].

1. The lack of visibility is a problem for the project manager. To measure the progress of the project the manager needs regular deliverables. This can be hard to produce in an evolutionary development. To produce documentation of every version of the system is very cost-inefficient.
2. The nature of continual change in this model tends to create a poor software structure.
3. Sometimes, special tools and techniques may be required to allow rapid development. The lack of compatibility with other programs and techniques together with possible skill needs to use these programs and techniques may be a problem.

Sommerville [1] do point out that the evolutionary development might be the best way to carry out projects of small to medium-sized systems. In development of larger systems, the problems might be too great and other models are more suitable.

2.2.3 Spiral model

When developing a large system, one single software process model might not be enough to cover all the different parts of the system. The approach used to develop one specific part of the system might not work at all in another part. For this purpose, hybrid models containing essential parts from a range of models have been developed. One such model, referred to as the spiral model, was originally proposed by Boehm [3]. Instead of describing the development process as a sequence of activities, Boehm uses a spiral where each loop represents a phase in the process. These phases are not fixed activities and may therefore be defined in the way the project needs. Each phase do however have a fixed internal structure that consists of four sectors [1].

1. Initially the objectives for that phase are defined. Constraints are spotted and a detailed management plan is drawn up. Risks are identified and due to the outcome of this identification, alternative strategies may be planned.
2. For each of the risks found in the prior sector, a detailed analysis is carried out. After that, measures are taken to reduce the risk.
3. In the next sector, a development model for the system is chosen based on the risk evaluation. Depending on the characteristics of the identified risks, the most suitable development model is chosen for the forthcoming parts of the project. This model can for example be a traditional evolutionary model or the waterfall model.
4. The last sector in each phase contains planning for the rest of the project. A decision of whether to continue with a further loop is made in this sector. If so is decided, plans are drawn for the next phase in the project.

The most important difference between the spiral model and other software models is the consideration of risk. Because risk often causes project problems such as time and cost overruns, risk minimisation is an important task in project management. This is the reason to why the spiral model considers risk identification as such an important part of the process [1].

3 Requirements engineering

One of the main issues in today’s software development is the *requirements engineering process*. Sommerville [1] describes requirements engineering as the process of finding, analysing, documenting and checking the *requirements* of a system. Requirements engineering also includes the processes of maintaining and updating these requirements.

3.1 Requirements

The requirements of a system are the description of its services and constraints, in other words, the requirements describe what the system should and should not do. It is often difficult for the engineers to formulate the demands on a system into good requirements, because the problems to be solved by the system are often very complex and it can also be hard to understand the nature of the problem.

The requirements (demands) come from users and other *stakeholders* [2]. A stakeholder can be defined as anyone who has some kind of indirect or direct influence on the system and anyone that will be affected by the system, such as: the users, the developers, the users’ and developers’ companies, sponsors, customers, authorities, etc.

Sommerville [1] divides requirements into three description levels:

1. *User requirements* are high-level abstract requirements in natural language and diagrams that describe what services the system should provide and under which constraints the system must operate.
2. *System requirements* are detailed descriptions of services and constraints.
3. *Software design specification* adds more detail to the system requirements and is used to bridge the requirements engineering and design activities.

Requirements are also often classified as being either *functional*, *non-functional* or *domain requirements* [1].

Functional requirements define what the system should and should not do, how the system should react to certain input and what services it should provide.

Non-functional requirements describe the limitations on the functionality and services of the system. They can, for example, be timing-constraints and standards.

Domain requirements come from the system's application domain, and can be both functional and non-functional. They can, for example, describe limitations or features needed on the system due to the environment it should operate in.

3.2 Requirements specification

The document containing the requirements is called the *requirements specification*. The presence of the requirements specification is obvious throughout the whole development process [2]. In the early stages the requirements specification is a vital piece of the system analysis and often a part of the contract between customer and developer. Later on, the specification is an important input to the design phase and its extension, the implementation of the project. During test and evaluation stages the requirements specification plays an important role in forwards- and backwards-tracing between demands, requirements and program features. In all, the requirements engineering and its key object, the requirements specification, are present and significant to project outcome during the whole process progress.

3.2.1 Good requirements specification

According to the IEEE Standard 830-1998 [24] a good requirements specification should be:

1. Correct
2. Unambiguous
3. Complete
4. Consistent
5. Ranked for importance and/or stability
6. Verifiable
7. Modifiable
8. Traceable

These characteristics are summarized and commented by Lauesen [2]:

Correct
Correctness is achieved when all the requirements are correct, in other words when it reflects a customer need or expectation. A typical example of how incorrect requirements are produced is when the analyst misunderstands the customer's needs.

Unambiguous
When all parties agree on what each requirement means, the specification is unambiguous. To help ensure unambiguity, formal specifications (for example math notations) may be used.

This is, however, only a problem if the developer believes he understands what the customer wants, when in fact the customer wants something else. If the developer should find a requirement ambiguous he will ask the customer for a clarification and the problem will never arise.

Complete
Completeness means that all the customer's expectations are covered. However, in practice it is unrealistic to include every single requirement since many requirements are too trivial to be worth specifying. If every single requirement was specified and included, the specification would be so long that it would lose understandability. Instead it is important to make sure that all non-trivial requirements are specified and to ensure that all business goals and critical issues are covered.

Consistent
By consistency it is meant that there should not be any conflicting requirements or groups of requirements. An example of inconsistency is if there is one requirement stating that a warning should be yellow and another requirement stating that it should be red.

To avoid inconsistency it is recommended that things are stated in one place only, and references are made from other places as needed. However, since this might make the specification more difficult to read a short version of the requirement together with the reference is better.

Ranked for importance and stability
All requirements are not equally important and some will change more often than others. Each requirement should have a priority as well as an expected change frequency.

The reason for setting the expected change frequency is to help the developer identify functions that should be easy to modify, so that he can pay special attention to these functions when designing them.

Verifiable
A requirement is verifiable if it is possible, within economically limits, to check that the product meets it.

Verifiability is important throughout the development process as well as in the finished product. It may be very costly to deal with an unmet requirement when the product is supposed to be finished. Further are the courts good at deciding whether or not a requirement has been met and this may also become costly.

Modifiable
Specifications that are easy to change and that maintains consistency when changed are modifiable. There are several ways to help ensure modifiability: the requirements should be numbered, a consistent terminology should be used, there should be an index and requirements should refer to each other rather than be repeated.

Traceable
Traceability is defined as having requirements that are both backwards- and forwards-traceable. A requirement is backwards-traceable if it is possible to see which goals and domain-oriented documents it comes from, and it is forwards-traceable if it is possible to see where it is used in design and code.

3.3 Requirements engineering processes

Sommerville [1] partitions requirements engineering into four generic, high-level requirements engineering activities that deal with creating and maintaining a system's requirements documentation: *feasibility study*, *requirements elicitation and analysis*, *requirements specification* and *requirements validation*.

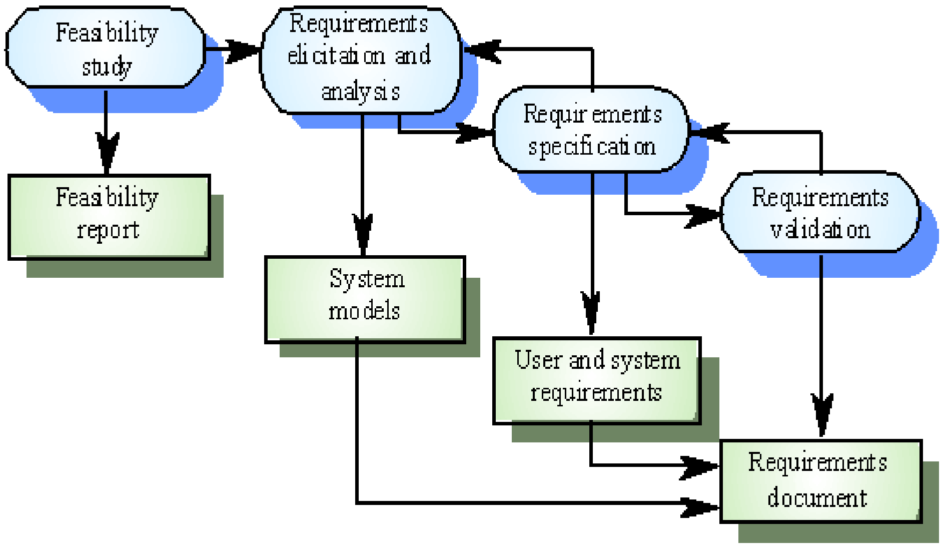


Figure 2 The requirements engineering process [1]

In addition to these four activities Sommerville includes a fifth activity, *requirements management*, which deals with the fact that requirements change.

Requirements specification has already been covered and the other four activities are summarized in the following sub-sections.

3.3.1 Feasibility studies

The purpose of the feasibility study is to determine whether or not the system contributes to business objectives. Therefore, the requirements engineering process should start with a feasibility study for all new systems. To decide the systems business value Sommerville [1] lists a number of questions for which answers should be sought:

1. Does the system contribute to the overall objectives of the organisation?
2. Can the system be implemented using current technology and within given cost and schedule constraints?
3. Can the system be integrated with other systems which are already in place?

The outcome of the feasibility study should be a report recommending whether or not it is worth to continue with the system development process.

3.3.2 Requirements elicitation and analysis

In the next stage, after the feasibility study, requirements are elicited and analysed. This process often involves several techniques (see Lauesen [2]) to, together with stakeholders, find and formulate the requirements.

Elicitation and analysis is an important but also difficult process. The difficulties are due to the fact that [1]:

1. Stakeholders may be unable to express what they want from the system, often they only know what they want from the system in the most general terms and they may be unaware of the implementation cost of their demands.
2. It is hard for the requirements engineers without experience in the customer’s domain to understand the stakeholders’ requirements expressed in the stakeholders’ own terms.
3. The requirements engineers have to discover all potential sources of requirements. Since there often are many such sources, the requirements engineers get requirements expressed in different ways among which they have to discover conflicts and similarities.
4. There may be political factors that influence the requirements of the system.

5. The importance of certain requirements may change and new requirements may emerge, since the economic and business environment in which the analysis takes place is dynamic.

3.3.3 Requirements validation

After the requirements specification is complete it has to be validated. In the validation the specification is conducting several checks to ensure that it fulfils the criteria of a “Good requirements specification” (See section 3.2.1).

It is important to validate the requirements, both after the requirements specification is complete and continuously throughout the project, because the cost of repairing an error originating from a faulty requirements specification greatly increases towards the end of the development [1].

3.3.4 Requirements management

Requirements management is the process of storing and managing changes to the requirements. Requirements will inevitably change over time as customers can change their minds, the domain can change and other factors that affect the system can change. Requirements can also turn out to be wrong, too expensive to meet and other problems can arise that forces a change to the requirements.

3.4 Release planning

Lauesen [2] points out that most successful projects end up with a series of releases, even though their goal from the beginning was to have one single delivery of the product. There are many reasons for this. The developers may suddenly realize that they cannot keep the deadline. Another reason could be that the project might require several prototypes or iterative development to ensure the correct functionality and to receive feedback from customers. Another example is in large-scale development of a continuously evolving product where the company constantly must implement new requirements to keep, and get new, customers and stay ahead of competitors. The process in the latter example is called *market-driven requirements engineering*.

Therefore *release planning* is an essential part of a company’s requirements engineering process. Release planning is about prioritizing the requirements and deciding in which release certain requirements should be included.

3.4.1 Market-driven release planning

In market-driven release planning, as explained by Karlsson [4], the company must make every release attractive to the customer. This means that sometimes the company might wait with including some features to make later releases attractive. There also needs to be a balance between new features and improvements of old features and it must be the most appropriate features to include and improve. The releases must also be made at appropriate intervals, with customer needs and competitor's releases in mind. To make it even harder for the developing company, we must not forget that each feature takes a certain amount of time and cost to include or improve.

In conclusion, the developing company must have a carefully developed release plan where customer needs, development cost and development time for every release must be balanced in the most appropriate way.

3.5 Requirements prioritization

There are often more requirements on a system than can be implemented at once [5]. The requirements must therefore be prioritized so that the most appropriate set of requirements is included in the intended release (see section 3.4.1).

To decide which requirements that are of most significance the prioritization is often done by using two criteria, for example *cost* and *value*, where cost can be a measure of the development cost for estimated time and resources that will be needed and value can be the business value. The aim is, of course, to maximize the value to a minimal cost.

When it comes to prioritizing requirements, there are a number of different techniques. Three commonly used requirements prioritising techniques are the *planning game*, *pair-wise comparisons* (PWC) and the *$100-test*.

The pair-wise comparisons method is a part of the *analytical hierarchy process* (AHP) and the *incomplete pair-wise comparison* (IPC) is a modification of PWC. Therefore, these are also described in this section.

3.5.1 Planning game

When using the *planning game* (PG) to prioritize, all the requirements are written on what is called *story cards*. A story card contains a short description of the requirement and other useful information like requirement/story number, the requirement's date and more. Planning game originates from the extreme programming methodology developed by Kent Beck [6] and is the commonly used prioritization technique in extreme programming.

The developers estimate how long/how much it would cost to implement each story. Then the developers sort the stories into three different piles based on how accurate they believe their estimations to be: (1) those stories that they can estimate precisely, (2) those that they can estimate reasonably well, and (3) those that they cannot estimate at all [6].

At the same time, the customers sort the same stories into three different piles based on their significance: (1) those without which the system will not function, (2) those that are less essential but provide significant business value, and (3) those that would be nice to have [6]. After this, the stories can also be prioritized within the piles if desired.

3.5.2 Pair-wise comparisons

In *pair-wise comparisons* (PWC) all possible pairs of requirements are compared to determine which of the two is of higher priority [5]. After the comparisons have been made, the priorities are calculated using the matrix multiplication parts of AHP (See section 3.5.4).

This means that if you have n requirements you need to do $n(n-1)/2$ comparisons for each criteria, this makes the PWC technique time-consuming as the number of requirements increase. However, this redundancy makes the technique insensitive to judgement errors and furthermore, a consistency check can be included in PWC where judgement errors can be identified.

3.5.3 Incomplete Pair-wise comparisons

Due to the fact the PWC becomes very time consuming as the number of requirements and the number of criteria increases it is desirable to decrease the number of necessary comparisons. Harker [7] has developed a method, known as incomplete pair-wise comparisons (IPC), to reduce the number of needed comparisons to be between n and $n(n-1)/2$ for each criteria.

After the n first comparisons have been made, IPC uses an algorithm to calculate the missing values. This algorithm puts the comparisons in a directed graph where each node represents a requirement and each arc represents a comparison between the requirements (nodes). The algorithm then uses the geometric mean of a random set of paths between two nodes to calculate the missing comparison. The algorithm is not using all possible paths, because the immense number of paths would make it unrealistic to perform the calculations since the number of paths grows exponentially as comparisons are added. Further, the IPC algorithm also calculates which comparison that should be made next by calculating which comparison that will give the most useful information.

Also a value of how useful the information from the next comparison would be is calculated. This value is used as a stopping rule, if that value is below a given constant the next comparison is considered to give too little information and the prioritization is completed.

In conclusion IPC can result in substantial timesavings by the reduced number of comparisons. In addition, by always doing the comparisons that give the most information first, the accuracy is kept as high as possible.

See [7], for a detailed description on the IPC algorithm.

3.5.4 The analytic hierarchy process

AHP, designed by T.L. Saaty [8], is a model that includes PWC and is used to aid the decision-making.

Joachim Karlsson and Kevin Ryan [9] describes the four steps in AHP used for decision making and how to check the results for consistency:

1. Set the requirements in the rows and columns of a matrix. With *n* requirements this will result in an *n*x*n* matrix.
2. Now a pair-wise comparison is performed for all the requirements. The scale used can be 1/9, 1/8 … 1/2, 1, 2, 3 … 9. To explain how the scale is used, imagine putting the two requirements next to each other and the more significant one of the requirements are, the farther to that side in the scale the number is taken. This will result in a matrix, called the *comparison matrix*, with ones in the diagonal and the result from the comparison in all the other positions. When translating the relative significance in each comparison to a numerical value, different scales can be used. The scale mentioned above is the original scale by Saaty [8], but research has been done to investigate the effects of different kinds of scales [10].
3. Now the eigenvalues of the matrix are estimated using a method called *averaging over normalized columns*. Which means that the column's sums are calculated and used to divide each element, belonging to that column, with. This results in a new matrix, in which the row sums are calculated and divided by the number of elements, *n*. The result is referred as the *priority matrix*, which is an estimation of the eigenvalues.
4. Finally, the estimated eigenvalues are assigned to the corresponding requirements and give a percentage of how significant each requirement is.

After completing the four steps the prioritisation is done and a consistency check can be made thanks to the redundancy given. The consistency check can be divided into two steps:

1. The *consistency index* (CI) is calculated: $CI = (\lambda_{\max} - n)/(n-1)$ where $\lambda_{\max}$ denotes the maximum principal eigenvalue of the comparison matrix.
2. The *consistency ratio* (CR) is calculated: $CR = CI / RI$ where RI is the consistency indices of randomly generated reciprocal matrices from the scale 1 to 9, also known as *random indices*.

Generally, a consistency ratio less than, or equal to, 0.10 is considered acceptable. In practice though, consistency ratios higher than 0.10 is not uncommon.

3.5.5 $100-test

The *$100-test* [11] is a simple and fast prioritization technique that is also easy to use when there are several people involved with the prioritization. Every participant is given $100 of “pretend money” each. The money is then used for “purchasing” the requirements they want, spending more money on the requirements they believe is more significant. When everybody has used up their $100 the results are summarized so that a ranking of the requirements is made.

There are, however, a few drawbacks with this method that should be kept in mind. Firstly, the technique can only be used once in every project, because the participants will be influenced by the results, once they are known. For instance, if you are a participant and your most wanted feature is high up on the list but your next favourite feature did not make it to the list, you can put more money on that second feature next time. Counting on that the other participants will, once again, put their money on your favourite feature so that it will still make it to the list.

Secondly, even if a prioritization have not been done yet, a tricky participant may put all of his money on a requirement he wants but are not as important as other obviously important requirements, because he knows that the other participants will put their money on the more important requirements.

4 Retrospective analysis

George Santayana, a Spanish-American 19th and 20th century philosopher [25], once stated that:

"Those who cannot remember the past are condemned to repeat it" [12]

With this quote as a starting point, Joseph Juran identified the retrospective analysis as a method of learning from work experience already in 1988. He named the practice the "Santayana review" after the great philosopher [13]. The main idea of a Santayana review is very simple; by taking time to examine what happened in the last project and learn from it you can use the new knowledge to improve the outcome of the next project [13].

4.1 Retrospectives in practice

When carrying out retrospective analysis, different approaches can be used in order to decide how to gather new knowledge. According to Nolan [14] the best results are achieved by "learning from success". In practice this means that measuring and understanding of old projects should be focused on the parts that made them successful. Extracting these successful processes and introducing them into new projects can achieve the best results in a rapid way. The biggest limitation with this approach is the need for an analysable and finished successful project to use as knowledge source. If no such project exists, the analysis can impossibly be done. This is of course in particular a problem for young and inexperienced development teams. On the other hand, if such a project does exist, the introduction of the analysis method seldom receives resistance from the involved development teams because of its nature of focusing on good practice instead of bad [14].

The other approach to use when doing a retrospective analysis is obviously to focus on bad practice. One particular case of big interest is to analyze projects that failed big time. Because of this reason, retrospective analysis is sometimes referred to as "Post-mortem reviews" [15]. Besides the direct knowledge that prevents mistakes from earlier projects, other advantages can be drawn. One such advantage is the increased sharing of experiences within the development team and also communication of the understanding to other teams [16]. Another advantage is the increased individual recognition and remembering of new knowledge gained during the project for each team member participating in the retrospective analysis [16]. There exists several ways to perform retrospective reviews. Dingsøyr et al [17] has presented a lightweight retrospective review based on focused brainstorming and analysis by grouping problem issues on post-it notes. This lightweight review does follow a general structure that seems to be suitable for most kind of retrospective processes [Birk, Kerth]. This structure is divided into three main parts.

1. A *preparation* part is the first phase. During this phase a better understanding of what has happened is achieved by going through project documentation. Also a goal for the analysis is determined in this phase.
2. The next part of the analysis consists of *data collection*. Techniques proposed for collecting data by Birk et al [16] are *semistructured interviews*, *facilitated group discussions* and *KJ sessions*. The latter is a group methodology for collecting and structuring data developed by Japanese ethnologist Jiro Kawakita [16].
3. To finalize the process an *analysis* phase is done. This can be done in several different ways. One approach, suggested in the lightweight post-mortem review by Dingsøyr et al [17], is to use Ishikawa diagrams as an analysis tool to show causes.

The software development of today is a very dynamic and flexible process. This demands a continuous improvement and evolution of the companies' development processes. According to Kerth [15] the most important step toward this improvement is by doing retrospective analysis. As mentioned above, the running point in a retrospective analysis is to create future advantages by analysing actions taken in the past. By doing this, the developer can hopefully prevent mistakes made in the past being repeated and a continuous progress in process evolution can be done.

4.2 The PARSEQ method

In this section we summarize the PARSEQ (Post-Release Analysis of Requirements SElection Quality) method, which is a retrospective analysis method developed at the Department of Telecommunication Systems, Lund University, Sweden [18].

To be able to perform the PARSEQ method the company must have multiple releases of the product as well as access to requirements from prior releases. Each requirement must be tagged with the release it was implemented in, or if it was postponed or excluded. Further the company must have access to employees who have decision-making experience from earlier releases as well as a facilitator with experience from retrospective analyses. PARSEQ is mostly used for user requirements, because the goal is to increase the business/user value by improving the release planning process.

In the method a sub-set of requirements from previous releases is systematically analysed and a set of root causes to suspected incorrect requirements selection decisions are identified and analysed. The goal is to find improvements that will increase the company's ability to plan coming releases, by learning from earlier mistakes.

The PARSEQ method basically consists of the four steps that are briefly described in the following sub-sections and shown in Figure 3.

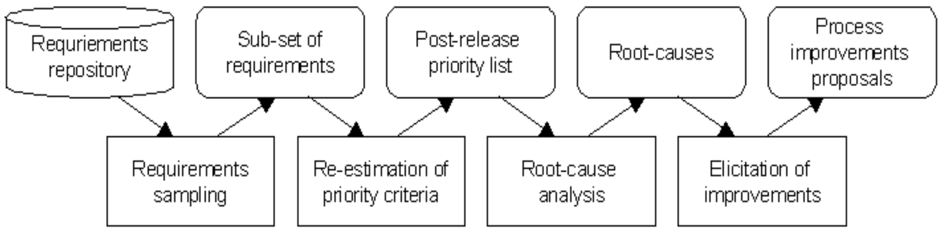

Requriements repository
Sub-set of requirements
Post-release priority list
Root-causes
Process improvements proposals
Requirements sampling
Re-estimation of priority criteria
Root-cause analysis
Elicitation of improvements

Figure 3 The PARSEQ process, the lower rectangles represent the four steps [18]. The five upper icons represent inputs and outputs to the four process steps.

4.2.1 Requirements sampling

Requirement candidates from previous releases, releases that have been on the market long enough to allow an estimation of the included requirements current market value, are chosen to be included in the analysis. These candidates are either requirements that were included in one of the releases or requirements that were rejected and have not been included yet. Hence, the sample should include requirements that have not been included as well as requirements that have been included. Further the sample should contain enough requirements to represent the product but not so many requirements that they cannot be handled in one session.

4.2.2 Re-estimation of priority criteria

The sample created during the requirements sampling step is post-released analysed and a re-evaluation based on what have been learned and on what is known at this point, is done. The thought is that the new estimations of the criteria are more accurate since the releases have been on the market for some time. With the new estimations it can be seen which of the requirements that should have been included in the releases with today's knowledge. If the result is that the exact same requirements should have been included, the company has either not learned anything since the planning of the old releases, the market has not changed at all or the company did a very good job during the release planning.

The goal of the re-estimation is to discover planning decisions that would have been made differently with today's knowledge. These differences are noted and used in the root-cause analysis. Hence, the output of this step, and input to the root-cause step, is a list of requirements that were implemented too early or too late.

4.2.3 Root-cause analysis

The aim of the root-cause analysis is to find and understand why release-planning decisions are made. Root-causes for the old decisions are sought and the differentiating requirements from the re-estimation step are mapped to one or more of these root-causes to illustrate the decision disposition and to be used as an input to the next step.

4.2.4 Elicitations of improvements

In this, last step, of the PARSEQ method, the goal is to elicit release-planning improvement proposals from the root-causes found in the previous step.

The goal is to get a list of high-priority areas of improvements that can be introduced in the release-planning process and thereby improving the accuracy of coming releases.

5 Method

Before the development could begin, it was necessary to learn how the PARSEQ method works. It was also necessary to get a picture of what features should be included in the program, what the program should look like and how it should work. An understanding of the PARSEQ method and the program was achieved by theory studies, in the form of articles, and discussions with the customers. The theory studies and discussions continued, as needed, throughout the entire development, especially when it was time to implement the different prioritization techniques. A good understanding of PARSEQ and the implemented techniques was a prerequisite for the success of the project.

5.1 Implementation

The nature of the implementation process was strongly affected by the close collaboration with the customer, moreover the supervisor of the work. This fact made some important choices of implementation procedures possible. One of the biggest advantages of these choices was the possibility to develop a system by using short iterations and frequent customer feedback. Another advantage was the chance to discuss questions and in cooperation work out uncertainties in the requirements specification. These advantages were factors that shaped the development process in a way that resulted appropriate for this specific project.

The main structure of the development process followed an evolutionary software process model (see section 2.2.2). Initially a prototype with the fundamental functionality implemented was produced. The selection of what should be considered as fundamental functionality was together with the customer extracted from the requirements specification. When the prototype was considered independent enough, a first user test was carried out. The outcome of this test, in terms of new and modified requirements, together with requirements from the original requirements specification were then selected for the next iteration of the implementation process. This procedure of requirements selection, implementation and testing was then repeated throughout the development process and the final system is an evolution of the initial prototype.

The entire implementation is done in Java on personal computers running Windows and Unix. As a development platform, the open source tool Eclipse [26] was used. Both the choice of implementation language and platform were requirements present in the original requirements specification. The vast majority of the implementation work was done using the *Extreme Programming* (XP) practice pair programming [19]. Briefly explained this means that two programmers sit in front of the same computer. One programmer has the main responsibility to write code and focus on the method at hand. The other one has the job to continuously perform code review. Other important issues for the code reviewer are things such as making suggestions for improvements, attentive corrections and keep the overall class structure solid.

The modeling of the software developed was strongly affected by the choice of evolutionary development as the main work procedure. Also the selection of an object-oriented programming language had a great effect on the modeling work. After receiving and reviewing the initial requirements specification a rough model of needed packages, classes and objects were made. When modeling the main structure of the system, special consideration of possible future refactorings were taken into mind. This decision was a result of the knowledge by experience from earlier projects that the evolutionary development can demand significant changes in class hierarchies and system structures. Also sketches of the first graphical interface were made early in the project. Since a great role of and advantages achieved from the system is via the direct user interaction, a big effort was early made to attain a good usability.

5.1.1 Requirements engineering

This project’s type would, according to Lauesen’s [2] definitions, be a mix of in-house development and contract development with emphasis on in-house development. In-house development is when a project is carried out inside a company for internal use and contract development is when the developing company delivers a system to the customer according to a contract and requirements specification.

The customer initially provided us with a requirements specification. We did, however, also have a close collaboration and good communication with the customer. Therefore, all requirements were not documented and some of the documented requirements were changed.

Setting up some kind of requirements engineering process, where we could formally prioritize the requirements in the specification with, for example, the planning game method, was discussed. However, it was concluded that for a project of this small size that was unnecessary. Instead the requirements to give high priority were decided in collaboration with the customer.

Since requirements are bound to change over time (see section 3.3.4), the architecture and implementation of the program was done keeping requirements management in mind. Hence, even though we did not really have any formal requirements engineering process in this project, changes to requirements and new requirements should not pose any problems.

5.1.2 Configuration management

The use of a configuration management tool like Concurrent Versions System (CVS) [27, 28, 29] was discussed. However, after initial research and experiments we concluded that the administration and compatibility issues with the resources at our disposal, would give a too large time-overhead to be worth it for a project of this size.

According to Whitgift [20] it is suitable to use directories to hold configurations in projects with no more than ten thousand lines of source code. Therefore we instead chose this way to manage the configurations, releases and source code of our project.

5.1.3 Graphical user interface

The main purpose of developing the tool was to a great extent to provide a more efficient way to perform an existing analysis process. This means that if the outcome should be successful, the tool must have a high degree of usability. This comprehension was held in focus throughout all parts of the process involving user interaction.

Usability is often broken down into usability goals to capture the most important parts of what often is referred to as good interaction design [21]. This break down was also made in this project.

1. *Effective to use:* The general goal of the whole project is to develop a tool that is capable of letting the user to carry out their task, the PARSEQ process, in an effective way.
2. *Efficient to use:* By making the user interaction mainly consist of standardized actions, as for example pressing buttons and click & drag with the mouse, the aim is to create a system where the user in an efficient way can complete his/her tasks. Each part of the process is also put in a well-defined and separate step, which hopefully will provide a clear structure of the program and an efficiency of use.
3. *Safe to use:* The system prevents the user from making serious errors by for example asking for confirmation before files are written over when saving. The division of the process into separate steps also provides the user an easy way of correcting errors made in a previous stage.
4. *Have a good utility:* As mentioned in section 5.1, the most important features and functionality were implemented first. By using this approach the system hopefully provides an appropriate set of functions to enable the users to carry out the most important tasks in the way they want to do them.
5. *Easy to learn:* To achieve a high learnability, all user interaction is done via well-known WIMP (Windows, Icons, Mouse and

Pull-down menus) interfaces [21]. The purpose of this is to create a way of interaction where any user recognizes and understands the possibilities and consequences of interacting. The program does also have a built-in help module where each window has its own step-by-step description of how to use.

6. *Easy to remember:* The natural sequence of completing a step in the process before another is started and the constraints built-in in the program to guide the user through these steps have the intention of creating a high memorability. By in this way providing an obvious guidance through the process, the way of using the program is hopefully easy to remember.

The development process of the user interface has followed an evolutionary model as well as the rest of the program. In the initially produced prototype a simple high-fidelity user interface prototype [21] was included. This prototype then served as a seed from where the rest of the windows and interaction components grew out.

Further, Ben Shneiderman’s eight golden rules [21] were kept in mind when designing the graphical interface and strongly affected the design of the windows throughout the program:

1. *Strive for consistency*
2. *Enable frequent users to use shortcuts*
3. *Offer informative feedback*
4. *Design dialogs to yield closure*
5. *Offer error prevention and simple error handling*
6. *Permit easy reversal of actions*
7. *Support internal locus of control*
8. *Reduce short-term memory load*

5.1.4 Testing

Most testing during the development was done using informal test techniques [22]. The main part of the test work was done in parallel with development and each new module of code was tested before being completely integrated with the rest of the system. Together with these low-level unit and integration tests, larger high-level system tests were regularly performed. These testing techniques were deemed to be suitable for this project because of its relative small amount of code and few numbers of extraordinary complex algorithms.

All testing of the graphical user interface during the main development phase was done by “quick and dirty” testing [21]. The use of this test method means that all information feedback from the customer was given through informal meetings, focused on quick input rather than carefully documented findings.

5.2 Evaluation

Three different evaluations were performed. One user evaluation with the goal of confirming the programs usability and functions, as well as finding possible improvements to these areas. There was also a retrospective analysis of the project conducted, with the PARSEQ method and Rainbowie as the support tool. The third evaluation was on Rainbowie's IPC algorithm.

The results of the evaluations can be found in section 6.2 and an analysis of the results can be found in section 7.

5.2.1 User evaluation of Rainbowie

A user evaluation was done together with, and at the same time as, a full PARSEQ evaluation of the Rainbowie tool itself. All parts of Rainbowie, except the pair-wise comparisons technique and the $100-test, were evaluated. The user evaluation was made by two users, which are very familiar with the PARSEQ method, together with the two developers.

While the developers took notes, asked and answered questions, the users were encouraged to freely comment and discuss all parts of the program that was used.

5.2.2 PARSEQ evaluation of the Rainbowie implementation

Since Rainbowie was developed to support the PARSEQ method it was considered appropriate to do a PARSEQ evaluation of it.

The evaluation was performed by two users together with the two developers using the planning game as the prioritization technique with value and cost as the criteria. Because the two users were the supervisors of the thesis, they were also the intended customers.

From the 52 product requirements (PK*xxxx* in Appendix C – Requirements specification) a sample of 20 randomly selected requirements were listed in an Excel-sheet together with their release status, to be used for the evaluation.

First the users carefully sorted the cards in the planning game by the value it had to them, how important they found the specific features to be. Then the developers sorted the cards by the actual cost, in time, it had taken to implement the features or an estimate of the time it would take to implement them if it had not already been implemented.

After the prioritization, the cost/value graph was studied to find requirements which release deviated from the optimal release. What were sought for were in other words, requirements with high values and low costs that had been implemented in a late release and vice versa.

When the identification of deviating requirements was done, they were added to the root-cause matrix. Then the PARSEQ evaluation was ended with completing the last step in the method by elicitating root-causes, possible reasons for the deviations and possible improvements to, if possible, avoid the deviations next time. The conclusions were entered in the root-cause matrix and exported to an Excel-file. The session was then completed by ranking the possible improvements for importance.

5.2.3 Comparison between Rainbowie and Focal Point

In the developed tool, three prioritization methods are implemented. In two of these methods, the $100-test and the planning game, the result of the prioritization is the direct outcome of the assigned values by the user. In the third method, the pair-wise comparisons, the result is based on a modified version of Harker's [7] IPC method (see section 3.5.3 for the method and section 6.1.1 concerning the modification). Since the IPC method only calculates estimations of the true results and the modification makes the method less accurate, a probable loss of precision will be introduced. To examine the approximate size of this loss of precision, a comparison between the developed tool and a commercial tool for performing pair-wise comparisons called Focal Point [30] was done. The purpose of this evaluation was not to make a statement of whether or not the implemented algorithm is usable in practice. The aim was rather to make an attempt to create an understanding of the possible size of the errors in the estimated values.

The evaluation was done by performing a pair-wise comparison for two criteria of ten requirements in both Focal Point and in Rainbowie. During the Focal Point prioritization, the so far assigned values were registered after each compared pair from nine (the smallest number of comparisons allowed for ten requirements in Focal Pont) to 45 (all possible comparisons). For Rainbowie, two series of pair-wise comparisons were done. First a series with all 45 pairs, compared in a semi-randomly generated order by Rainbowie (see section 6.1.1), was completed, i.e. a normal Rainbowie prioritization. After each pair, the so far assigned values were registered in analogy with the procedure for Focal Point. The objective of this first comparison was to evaluate how well the program as a whole could perform a pair-wise comparison compared with a commercial tool. An analysis of the progress of the assigned values throughout the whole prioritization series were also desired and therefore were all 45 comparisons completed.

The second series was obtained by letting Rainbowie do the same first 19 comparisons that were made in Focal Point. The objective of this second series was to evaluate if the results from internal modified

IPC algorithm differed from the results from Focal Points built-in algorithm in a significant way. The reason why only 19 comparisons were made was because this is the number of comparisons that Focal Point recommends the user to do for 10 requirements. Since the values assigned after 19 completed comparisons are recommended by Focal Point as estimations of the true values, 19 comparisons were chosen to be the stop point for this series.

Finally a second prioritization was made with Focal Point with 19 comparisons to examine if the recommended number of comparisons always resulted in the same assigned values. If that was not the case, a check was done of how large differences between the two prioritizations that could be considered acceptable by Focal Point. This examination was done by letting Focal Point perform two prioritizations with the recommended number of comparisons, 19 comparisons in this case. The value assigned for each comparison made was decided by the registered values from the earlier made full prioritization. Since Focal Point use some kind of random comparison generation, the 19 comparisons made where not the same in both prioritizations. This difference in the 19 chosen comparison to make can result in a different outcome from the two prioritizations, and therefore is this examination interesting to do.

The objective of this examination was to obtain a reference point of how accurate the pair-wise comparison method is considered in today’s industrial applications.

6 Results

The purpose of this thesis was to create a tool that can be used to examine the possibilities of more efficiently using the PARSEQ method in software development and research. Because of this formulation of the purpose, the result of the work can be divided into two main parts. The first part of the results is the implemented tool itself. By understandable reasons, this part was the most time consuming and work demanding section of the thesis.

The second part of the results is the tool's ability to examine the possibilities of more efficiently using the PARSEQ method in software development. A big part of this result could not possibly be examined within the limitations of this project because of the scarce time resources. To extract a definite answer to the question whether the tool can make the PARSEQ method more efficient, it probably needs to be applied to a larger scale of research work. The results from the evaluation in this thesis are therefore the outcomes of a relatively brief user evaluation of the system itself and subjective opinions from experienced PARSEQ analyzers about the ability to ease up the method.

6.1 Results from the implementation

The implementation work was, as mentioned above, the single most time consuming activity in the project. Therefore the result of the implementation is considered to be the main part of the results from the project in total. The physical result of the implementation is a program that covers a majority of the requirements given in the requirement specification. The complete requirements specification can be found in Appendix C – Requirements specification. A specified compilation of the implemented functionality will follow in the next section followed by the known limitations and unmet requirements. Also a rough description of the software architecture built will be presented in this section. For a more detailed version, see Appendix B – Detailed software architecture. The implementation resulted in a program called Rainbowie of about 10.000 lines of Java code.

6.1.1 Implemented functionality

To create a better overview of the results in this part, the compilation of the implemented functionality will be divided in the same classifications as in the requirement specification.

Development requirements
The tool is developed with the use of incremental development and the programming language used was Java. The open-source platform Eclipse [26] was used as development tool throughout the project. The program is written and can be run on workstations running Windows. Together with the program itself, this project report was written and delivered before the deadline of 30th of June 2005.

Common product requirements
When the program is started, a main window appears where the user can make choices by clicking buttons or menu items. By opening the help dialog, possible actions to take are presented to the user. This window, as well as almost all other windows in the program, can be minimized and maximized.

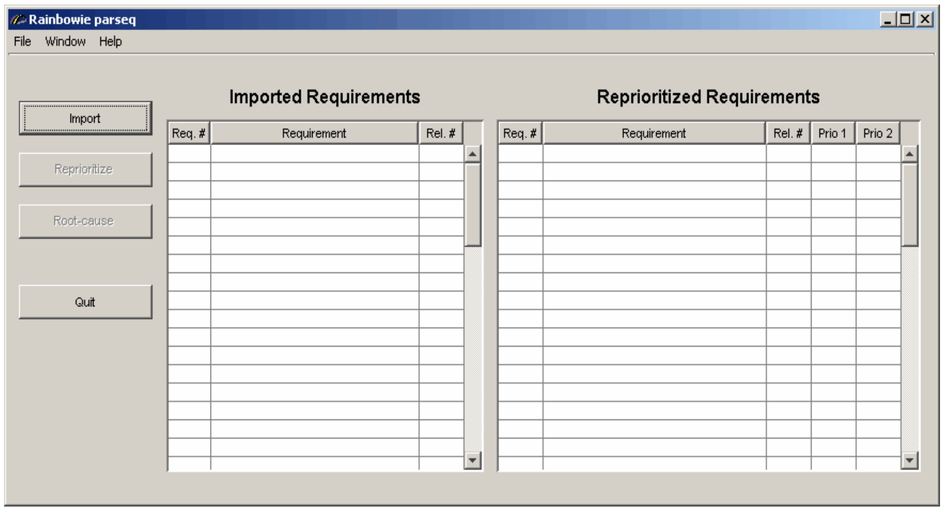


Figure 4 The main window of the program

Requirements for sampling
In dialog with the customer, an early decision was made to concentrate on sampling requirements from Excel spreadsheets only. This decision was made based on the facts that Excel is today a common way to handle requirements and that it is quite simple to extract requirements from ordinary requirement tools to Excel. The limitation of using Excel spreadsheets only was also considered to be a reasonable solution since importation from other programs, pure requirement tools in special, probably would result in a too high cost/benefit-rate according to the time required. Since the benefits from the possibility of sampling requirements from existing tools, which already has an easy interaction with Excel, would be relatively small in comparison to the time needed to implement that functionality, the Excel-only strategy was sustained.

When requirements are imported, they are displayed in a table with three columns: Requirement number, requirement description and release number. The possible number of requirements to import has no upper built-in restriction in the program. A lower bound of two requirements is set since a reprioritization with a single requirement is pointless. Imported requirements can be manually added and/or edited in all three columns by using a special edit window. The reason for this special window is to prevent erroneous editing in the imported list. If a requirement shall be tagged with a release number only, this can be directly done in the imported list. Anytime during the sampling process the user can choose to return to the main window by either pressing the accept button to continue the process with the current list or the reject button to start over again with the sampling.

Requirements for reprioritization

For reprioritizing the imported requirements, the user can choose between three techniques to use: Planning game, $100-test and pair-wise comparisons. The criteria *cost, value* and *risk* together with up to two own defined criteria can be chosen among, when doing a reprioritization. During all reprioritization, the previously set release number is not shown to prevent the user from being affected.

In planning game prioritization every requirement is represented by a card, with the requirement description written on it, placed on a desk. If a requirement is only represented by a requirement number, the number is written on the card instead. In the requirement specification there exists a requirement (PK3310) that says that each card initially shall be given a random position on the desk and cards shall not be put on top of each other. The algorithm for a random placement on the desk was implemented and integrated in the system without any problems. An algorithm for preventing cards to be put on top of each other where also implemented but the integration with the system led to a problem. Since another, informal, requirement from the customer was that each card should be large enough to be able to hold a significant part of the requirement description; the initial placing of the cards sometimes could not be done. This problem appeared especially when many cards where to be placed on a workstation with a low screen resolution, i.e. when many cards had a relatively small desk to be placed on there was not room enough to find an empty place for all cards. In the balance between large cards and prevention from placing cards on top of each other it was in discussion with the customer decided to stay with the large cards. By dragging and dropping the cards with the mouse, the user can put the cards in any of the three boxes marked “High”, “Medium” or “Low”. Within each of these three boxes, the user can reorder the cards in the same way as moving cards on the desk. When the user is finished with the reprioritization, a window for confirming the prioritized ordinal list of requirements will appear.

In the $100-test all requirements are presented in a table where the last column is editable. In this column, the user types in the desired amount of $ for each of the requirements. Below the editable column information of the number of spent $ are displayed to make it easier for the user to keep track of the progress.

An additional way to reprioritize requirements, not present in the requirement specification but stated as an informal requirement, has been implemented. This method can be explained as a combination of the two methods mentioned above. The user starts by reprioritizing requirements in the planning game method. When this is done, the reprioritization continues with the $100 method. The main difference in this $100 method, compared with the regular, is that the order of the list of requirements presented must be sustained. This means that the user cannot assign $ values to the requirements in the list that would modify the internal order of the list of requirements. When this reprioritization is done, the result is a list similar to the one obtained from a regular $100 method.

The pair-wise comparison technique is the most complex of the three techniques and therefore contains the most complicated functions. The priority values of each requirement are calculated by using the AHP. The scale used for comparing two requirements with each other is the Saaty scale [8], 1/9…1/3, 1, 3…9. If the user completes the comparison process and performs all *n(n-1)/2* comparisons (with *n* = the number of reprioritized requirements), the priority values are calculated with a true PWC-calculation. The user can however choose to finalize the reprioritization as soon as *n* comparisons have been done. If so is done, the values will be calculated with a modified IPC-algorithm. The reason for the modification of the IPC algorithm is because of the limited resources of time in the project. The authors found it more valuable make sure to be able to integrate a small, modified version of the IPC-algorithm than to try to, but not be able to ensure to succeed, make an implementation of the full, rather complex, algorithm. The modification of the algorithm consists mainly of the removal of the stopping rule functionality. This means that the algorithm to calculate all missing values by using a directed graph is done and a regular AHP calculation can be done to find the priority values. It is instead the next step in the algorithm that is removed. When a true IPC-algorithm calculates the next comparison to make according to the amount of potential information that will added by that comparison, and possibly suggests a stop in the prioritization, the modified algorithm does not. In the modified algorithm the next comparison to make is instead randomly chosen among the remaining comparisons and leaves the decision whether to stop the prioritization or not to the user. To ensure that the *n* first comparisons made can create the directed graph needed for the IPC, a true random selection of the comparisons cannot be made. Therefore a special semi-random algorithm is implemented to select the *n* first comparisons randomly within special restrictions. In the algorithm two adjustable threshold values are implemented. When few comparisons have been made, it is possible to use the average mean of all possible paths between two nodes in the graph when calculating a missing comparison. When the number of comparisons increase, making the number of all possible paths grow exponentially, a given value is set where the algorithm switches to using a random set of paths through the graph instead. Therefore, the two threshold values decide when the algorithm shall switch from all possible paths to a randomized number of paths and the number of random paths to use.

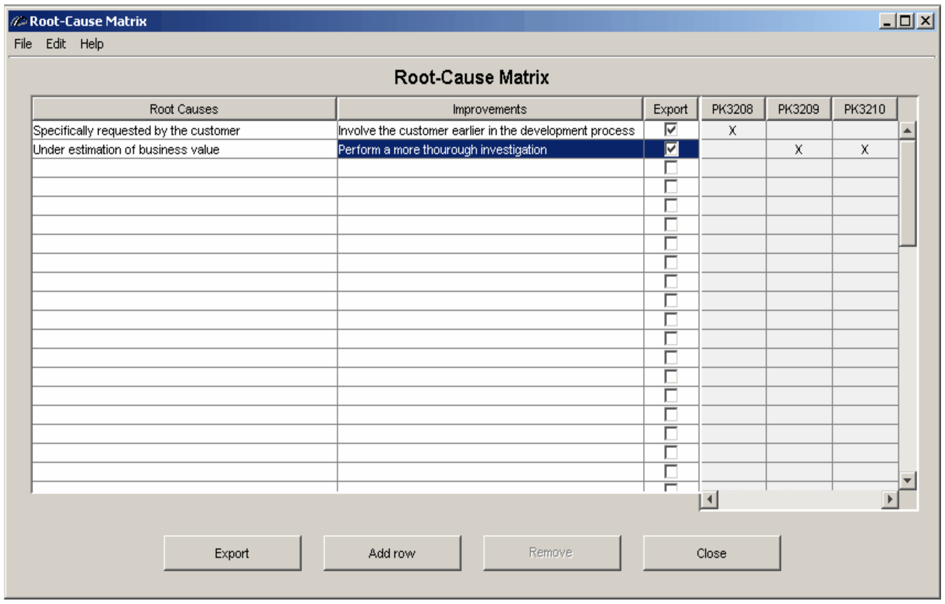
Requirements for root-cause analysis

The reprioritization made by the user can be presented in a graph with the chosen prioritization criteria on the axes. In the graph, each requirement is represented by an icon, the requirement number and the first few words of the requirement description. The assigned release number decides the look of the requirement icon. Requirements missing a release number are represented by a "+"-sign while assigned release numbers are represented by a number of circles. The number of circles is decided by which release number that is assigned.

When the planning game reprioritization method has been used, the graph is divided in three areas along each axis, i.e. nine areas in total. These areas correspond to the three boxes, "High", "Medium" and "Low", for each criterion in the reprioritization method. Within each area, the requirements are positioned out following the assigned reprioritized order.

In the graph for other reprioritization methods the requirements are positioned according to the assigned prioritization value in percent along each axis. In this graph, together with the requirements, there are also two support lines instead of the nine areas mentioned above. These two support lines are defined by the functions: $y = 2x$ and $y = 0.5x$ [9]. The purpose of these lines is to easily identify those requirements that have a more than twice as large priority value of one of the criteria relative to the other. Since one of the objectives in the PARSEQ analysis is to find originally incorrect prioritized requirements (see section 4.2.2), the ones identified by the support lines are highly interesting for further analysis.

When the user finds a requirement to include in a further analysis, it can be added to the root-cause and improvements matrix by double-clicking or by using a button in the graph window. The added requirement will then be added as a new column in the root-cause matrix that already contains two columns for root-causes and improvements.



Root-Cause Matrix

Root Causes	Improvements	Export	PK3208	PK3209	PK3210
Specifically requested by the customer	Involve the customer earlier in the development process	☑	X		
Under estimation of business value	Perform a more thourough investigation	☑		X	X
		☐			
		☐			
		☐			
		☐			
		☐			
		☐			
		☐			
		☐			
		☐			
		☐			
		☐			
		☐			
		☐			
		☐			
		☐			
		☐			
		☐			

Figure 5 The root-cause matrix with three requirements added

Requirements for improvements elicitation

In the root-cause matrix, the user can enter root-causes for the originally incorrect prioritization and improvements suggestions. This information can then be linked to one or more of the requirements added as columns in the table. Each requirement can also be linked to more than one root-cause and improvement suggestion.

When this final phase of the PARSEQ process is completed, the user can export the results, the root-cause table and the requirements graph, to an Excel spreadsheet for further editing and printing possibilities.

Quality requirements

The code is fully commented to ease possible extensions of the program in the future. Each class and method is described in large extent to make code maintenance uncomplicated and straightforward.

A detailed help module that provides a step-by-step description of how to use the program is implemented to ensure new users to be able to use the program. This help module can be used both as a guidance through the program for new users as well as a reference book for more experienced users.

A great effort has been made to create an uncomplicated interface for adding new prioritization methods. To create a complete plug-in interface was not possible due to, among others, details in the graphical interface and possible graph representations. Though, a comprehensive description of how to integrate a new prioritization method with the program is written and delivered with the final release.

6.1.2 Known limitations

The program was mostly developed and tested on a *personal computer* (PC) that was provided by the department. It was a PC that was running Microsoft Windows 2000 as *operating system* (OS) using Java version J2SE 1.4.2_07 SDK and Eclipse 3.0 as development environment. The program has only briefly been tested on other Java versions and operating systems.

When the consistency ratio is calculated for the PWC method, random indices are used as defined by AHP [9], up to order 15. For matrices of higher order than 15, the same RI value as for an order 15 matrix is used because no literature found presents the values for higher order matrices or how to calculate the values.

The algorithm for calculating assigned values when using an IPC is only a modified version of Harker's IPC algorithm. The reasons for this modification are already motivated in section 6.1.1.

6.1.3 Unmet requirements

In the requirement specification there are requirements that could not be met during this project due to one reason or another, where one reoccurring reason is the time limit of the project. In this section these requirements will be discussed more in detail and the reasons behind the decisions to exclude them from the program are explained. Also, the excluded requirements was discussed and approved to be postponed by the customer. References will be made to the final requirements specification of the project that can be found in Appendix C – Requirements specification.

In the requirement group concerning common product features, a requirement of a help button in every window of the program can be found (PK3103). The help buttons were replaced by help menu items because of convenience issues in some windows. The placing of a relatively big button tended to be very inappropriate in some of the smaller windows. By offering the same connection to the help dialog, via the menu, in all windows, a high consistency could be kept.

As mentioned earlier in section 6.1.1, import of requirements can only be done from Excel spreadsheets. The benefits achieved from implementing import interfaces towards other programs than Excel where estimated to be significantly smaller than the estimated cost in time needed to implement the functionality. Because of these circumstances, the decision was taken to focus on import from Excel files only.

Part of requirement PK3310 says that the story cards in the planning game window should not overlap each other. This has not been achieved since it very much depends on how many cards there are and how high the screen resolution is. It was a choice between larger, overlapping, cards with more information and smaller, non-overlapping, cards with less information and it was decided that easy access to the information on the cards was more important.

The requirement demanding that the comparisons should be in a random order (PK3317) has only partly been met, because the implementation of IPC demands that a directed graph is constructed from the *n* first comparisons (see section 6.1.1). Therefore, the *n* first requirements are not fully randomized.

A modified version of requirement PK3408 was implemented. Instead of marking requirements that after the reprioritization gets an “optimal release” that differs from its actual release, every release has a unique icon to tell them apart. This indicates to the user which requirements to investigate in the root-cause analysis.

Instead of implementing report writing and printing support, as PK3502 demands, the root-cause matrix and graph can be exported to an Excel sheet. Excel is widely used in industry and at the department; it also has excellent exporting, printing and report making support. We would, within the given timeframe, never be able to implement report writing or printing features that would be better than the features Excel have.

Requirement PK3503, concerning database storage of the results, was also considered a “nice to have” feature that would demand quite some time to implement. Further, to save the improvement suggestions to a database in a good way would require the program to have some kind of database support (like SQL) and a database server would need to be available. It was considered enough to be able to save the root-causes and improvement suggestions to an Excel-file.

There was a requirement stating that at least one industrial evaluation of the tool should be made (UK2006). However, at the time of the printing of this report this has not been done due lack of interest from the local industries and/or lack of time at the interested industries. Instead an in-house evaluation was conducted together with the customer.

6.1.4 Software architecture

The program is divided by functionality into seven packages where each package contains several classes. The steps in the PARSEQ process, three in this case because the last two steps have been merged into one in the tool; importing, reprioritizing and analysis, are each represented by an own package. This means for example that the classes relating to functionality used for prioritizing is collected in one package. In addition to these packages there exists two packages containing classes to represent the graphical user interface. Another package holds universal classes used throughout the whole program, i.e. the representation of a requirement. Finally there is a utility package that contains a set of utility classes.

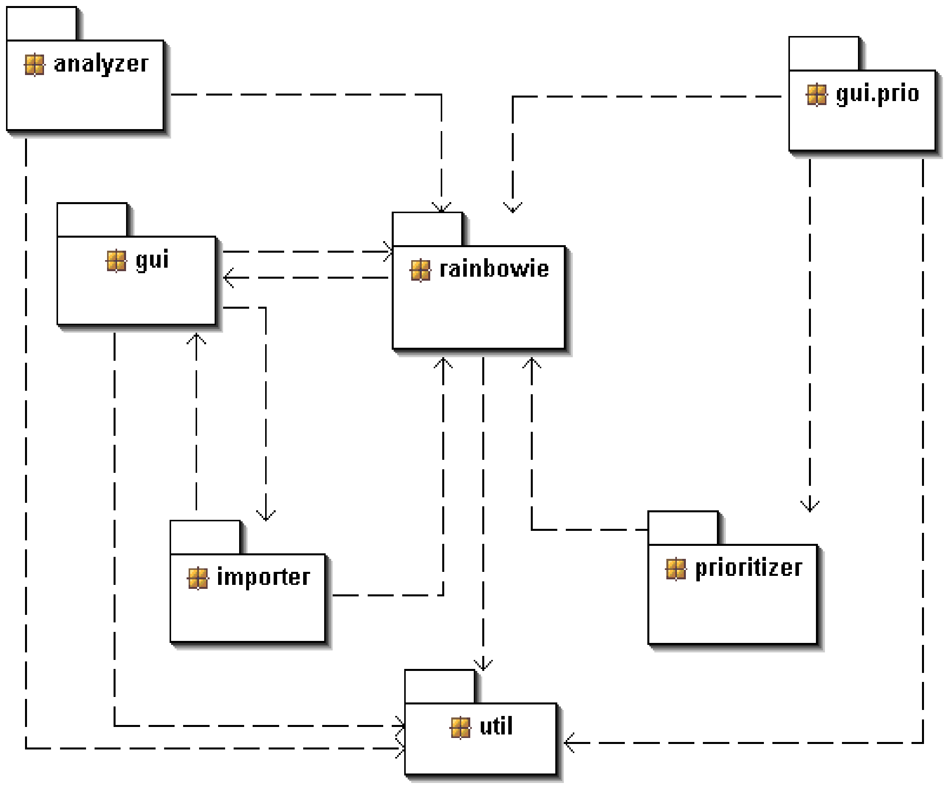


Figure 6 Package structure of the program

This chapter will provide a brief description of the packages mentioned above and the classes within these. To obtain a more detailed description, the reader is advised to consult Appendix B – Detailed software architecture.

The rainbowie package

This package contains the classes considered to be universal throughout the program. In the package there are classes to represent a requirement and a list of requirements in the system. The main class containing the main method and lists of imported and reprioritized requirements is also located here. All communication between the steps in the PARSEQ process is done via the lists of requirements in the main class.

The importer package

This package holds the functionality to import requirements from other programs. At this point, the program is only able to import from MS Excel spreadsheets that means this functionality is the single one located in this package. If a future extension would provide functionality to import requirements from for example other requirement engineering tools, this functionality should be added here.

The prioritizer package
When the reprioritization is done by the user, all needed calculations and functionality not directly connected to the user interaction is done in a separate class. For each prioritization method there is one such corresponding class located in this package. This means in practice, for example for the $100 method, that there is a class in this package keeping track of things like the number of users that has taken part in the reprioritization.

The analyzer package
When all requirements have been reprioritized, the process shall continue with the analysis phase. Internally in the program this means that the priorities assigned by the user shall be translated to positions in the analysis graph. To achieve this, each prioritization method corresponds to a class in the Analyzer package. All analyzer classes extend the Analyzer super class that contains fundamental methods for creating points and support lines for the graph. Depending on the unique characteristics of each prioritization technique, these methods are overridden to make a visualisation possible.

The package also contains a class for physical representation of a graph point and the attributes assigned to this.

The gui and gui.prio packages
Because of the relatively large number of windows in the program, some of the classes in the gui package are placed in a sub package called prio. In the sub package, classes managing user interaction during the reprioritization step are found. Each prioritization technique has its own window and therefore its own class in this package. Together with these a class to simplify a program extension with a window for a new prioritization method is located in this package.

In the package above gui.prio all other windows in the program are placed. All of these, as well as the prioritization windows, has a range of internal classes to take care of user interaction, drawing, table rendering and so on.

The util package
Frequently in the program, utility classes are used for wide range of functionalities. All these classes are collected in the utility package. The functionality provided by these classes is for example showing of the help dialog, setting standard sizes for window components and setting window alignments.

6.2 Results from the evaluation

In the following sub-sections the results of the three evaluations that were performed are presented, as facts and figures.

6.2.1 User evaluation of Rainbowie

The user evaluation resulted in a confirmation that the program's usability was good enough, as well as the implemented functionality worked as intended. There were, however, also a number of usability improvements suggestions and new requested features as a result of the evaluation. These improvement and feature suggestions are listed below:

1. Support for importing and handling earlier priorities.
2. Extensive undo support throughout the program.
3. Make it possible to change and adjust priorities in an easy way later in the process.
4. Ability to merge, group and split cards in the PG.
5. Make it possible to directly drag a card from one pile to another in the PG, without having to put the card on the desk first.
6. Visualize which requirements that belong to which pile in the PG's Confirm window.
7. Implement a slider in $100-window that should be used to assign the $. When one slider is pulled up, the other sliders should be lowered. Also, implement functionality to lock desired slider, so that they will not be changed when another is pulled.
8. Change the name of the Reject-button in the PG and the Print-column in the root-cause matrix.
9. Make it possible to map several improvements suggestions to one root-cause in the root-cause matrix.
10. Visualize in the graph window which requirements that have been added to the root-cause matrix.

6.2.2 PARSEQ evaluation of the Rainbowie implementation

As can be seen in the result of the prioritization in Figure 7, there are six deviating requirements:

1. PK3210 Had high value and low cost, but was not implemented until release 2

2. PK3201 Had high value and low cost, but was not implemented until release 2
3. PK3408 Had high value and medium cost, but was not implemented
4. PK3504 Had high value and medium cost, but was not implemented until release 2
5. PK3410 Had medium value and low cost but was not implemented until release 4
6. PK3104 Had medium value and low cost but was not implemented

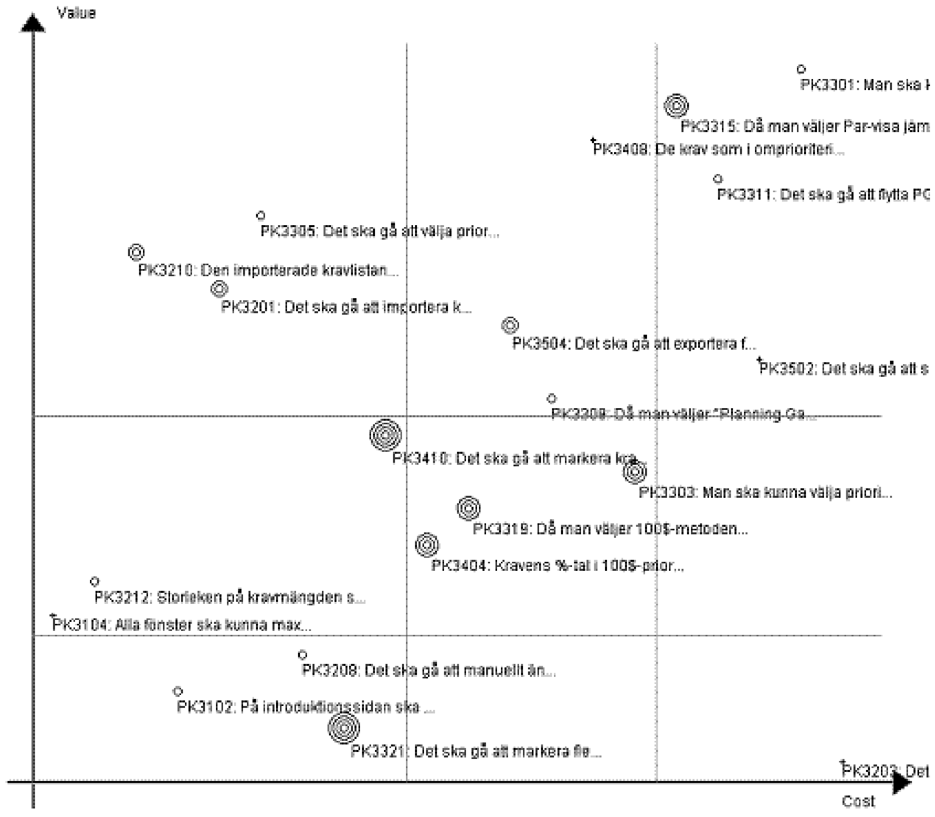


Figure 7 The resulting graph from the prioritization of the requirements in the PARSEQ evaluation of Rainbowie.

Table 1 shows the resulting root-cause matrix after finding the root-causes to the deviating requirements and after elicitating process improvement suggestions to help avoid the same root-causes next time.

Root Causes	Improvements	PK3210	PK3201	PK3408	PK3504	PK3410	PK3104
Inadequate elicitation the requirement was not there from the beginning	Even earlier prototyping and more discussion about the specification with the customer	X				X	
The first prototype did not need it	Not an erroneous decision		X		X		
Was expected to be harder to implement before an existing library was found	Look harder for existing solutions before solving the problem ourselves		X		X		
The requirement received a lower priority since it turned out that it is better to show the release numbers	Write the requirements at a higher level, not solutions and more discussion about the requirements			X			
Expected the solution to be harder than it was and weighted between different solutions	More discussion with the customer					X	
Partly implemented	Not an erroneous decision						X

Table 1 The resulting root-cause matrix after the PARSEQ evaluation.

From this table four improvement suggestions were found to be viable and ranked for importance:

- More discussion about the specification with the customer
- Write the requirements at a higher level, not as solutions
- Look harder for existing solutions before solving the problem ourselves
- Even earlier prototyping

This ranked list of process improvement suggestions concluded the results from the PARSEQ evaluation of the development of Rainbowie.

6.2.3 Comparison between Rainbowie and Focal Point

The registered values during the pair-wise comparisons in Focal Point and Rainbowie, as described in section 5.2.3, are presented in the tables and graphs below. The requirements used in the prioritization are called by their requirement numbers: REQ001 – REQ010.

The results displayed in the graphs shall be compared two and two, one graph for Focal Point and one for Rainbowie. Figure 8 shall be compared with figure 9, figure 10 with figure 11 and so on. Especially the two tools' abilities to early in the prioritization process stabilize each requirements value close to the finally achieved priority shall be observed. The results presented in the graphs are analysed in section 7.3.

45 comparisons in Focal Point – criterion 1

	Values after n number of comparisons				
Requirement:	**10**	…	**19**	…	**45**
REQ001	16,74%	…	17,35%	…	17,07%
REQ002	8,25%	…	7,79%	…	7,70%
REQ003	6,48%	…	5,94%	…	5,31%
REQ004	11,93%	…	11,96%	…	12,13%
REQ005	7,42%	…	7,51%	…	7,70%
REQ006	9,47%	…	10,02%	…	10,42%
REQ007	8,26%	…	7,41%	…	7,70%
REQ008	12,23%	…	12,23%	…	12,13%
REQ009	11,59%	…	12,43%	…	12,13%
REQ010	7,62%	…	7,37%	…	7,70%

Table 2 Assigned values by Focal Point for a full prioritization with 45 comparisons and criterion 1

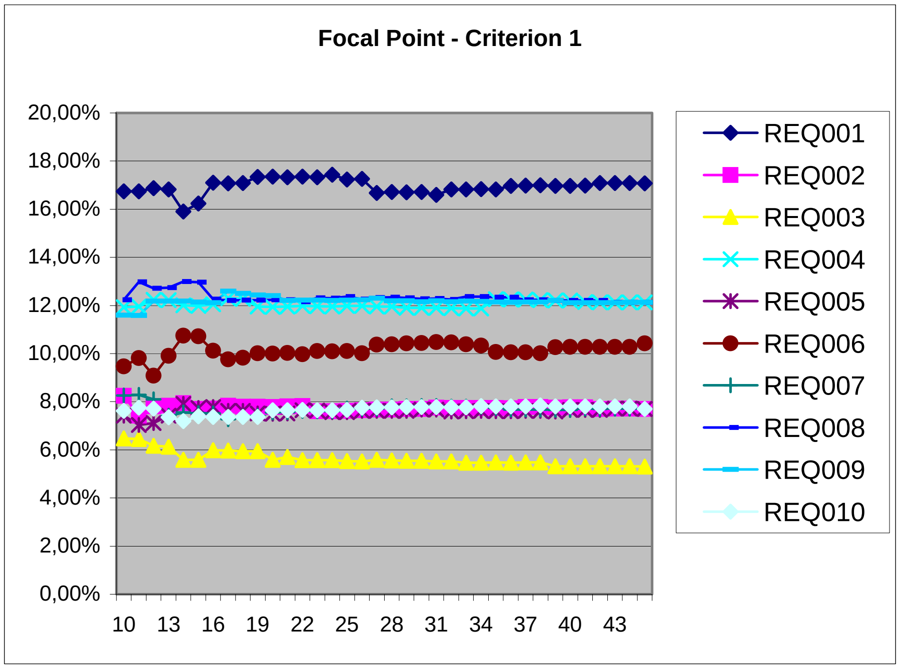


Figure 8 Visualization of assigned values by Focal Point for 45 comparisons and criterion 1

45 comparisons in Rainbowie – criterion 1

	Values after n number of comparisons				
Requirement:	**10**	…	**19**	…	**45**
REQ001	19,78%	…	13,69%	…	27,19%
REQ002	7,86%	…	8,95%	…	4,76%
REQ003	4,58%	…	4,46%	…	2,38%
REQ004	13,52%	…	13,44%	…	13,81%
REQ005	8,40%	…	7,94%	…	4,76%
REQ006	7,42%	…	11,07%	…	9,90%
REQ007	6,64%	…	9,09%	…	4,76%
REQ008	10,80%	…	12,97%	…	13,81%
REQ009	14,18%	…	11,61%	…	13,81%
REQ010	6,77%	…	6,73%	…	4,76%

Table 3 Assigned values by Rainbowie for a full prioritization with 45 comparisons and criterion 1

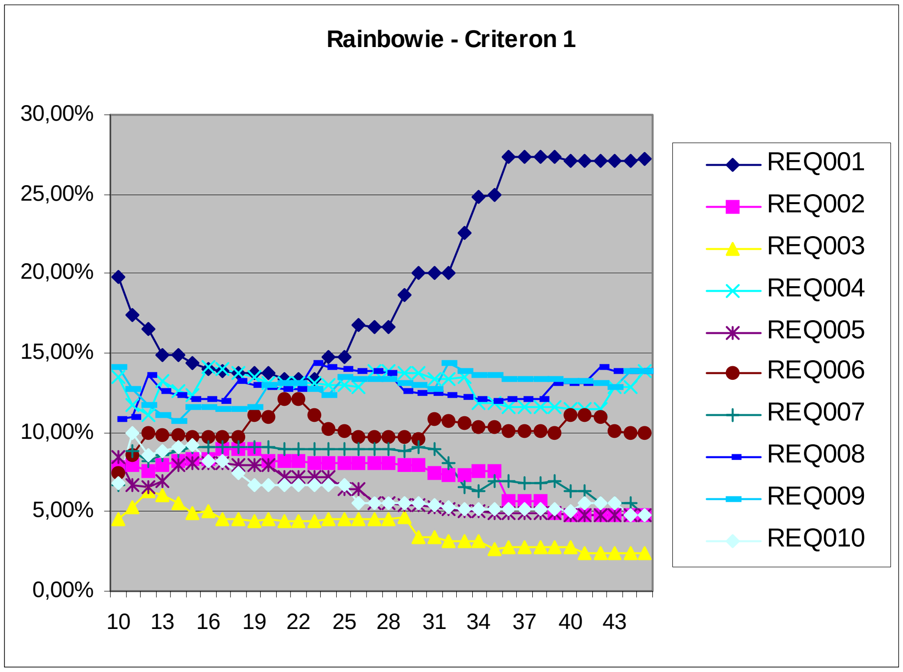


Figure 9 Visualization of assigned values by Rainbowie for 45 comparisons and criterion 1

45 comparisons in Focal Point – criterion 2

	Values after n number of comparisons				
Requirement:	**10**	…	**19**	…	**45**
REQ001	12,81%	…	10,26%	…	9,27%
REQ002	17,85%	…	13,58%	…	13,45%
REQ003	13,95%	…	17,79%	…	15,92%
REQ004	9,25%	…	9,10%	…	9,30%
REQ005	5,66%	…	5,61%	…	5,62%
REQ006	6,60%	…	7,30%	…	7,20%
REQ007	12,77%	…	17,24%	…	16,66%
REQ008	7,97%	…	6,48%	…	6,87%
REQ009	4,06%	…	4,32%	…	5,41%
REQ010	9,09%	…	8,34%	…	10,31%

Table 4 Assigned values by Focal Point for a full prioritization with 45 comparisons and criterion 2

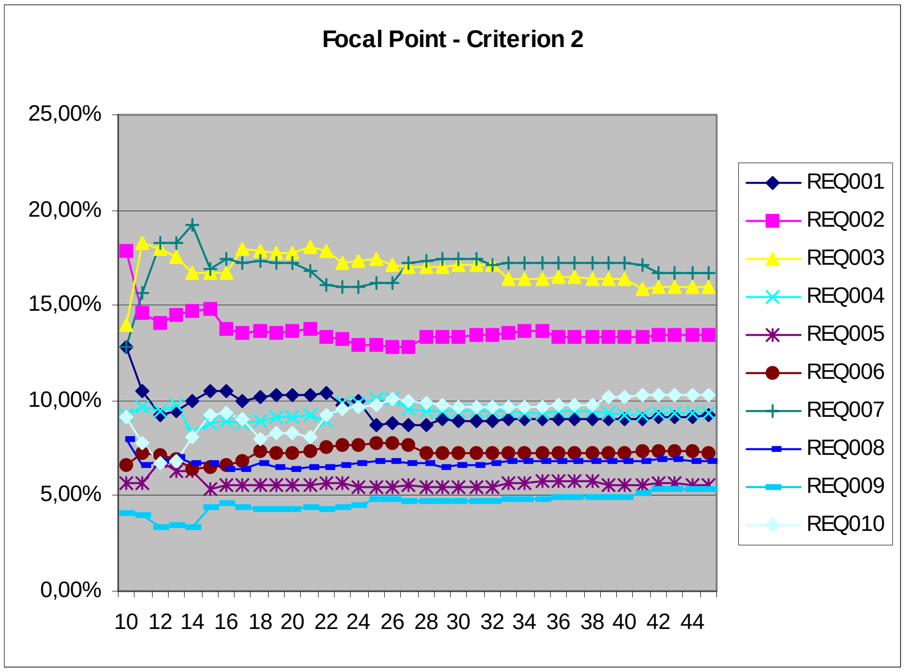


Figure 10 Visualization of assigned values by Focal Point for 45 comparisons and criterion 2

45 comparisons in Rainbowie – criterion 2

	Values after n number of comparisons				
Requirement:	**10**	…	**19**	…	**45**
REQ001	7,66%	…	8,95%	…	8,62%
REQ002	16,79%	…	13,14%	…	15,12%
REQ003	18,67%	…	16,49%	…	21,99%
REQ004	5,94%	…	4,86%	…	7,74%
REQ005	4,58%	…	6,13%	…	2,60%
REQ006	4,38%	…	6,35%	…	4,79%
REQ007	19,32%	…	18,95%	…	23,15%
REQ008	7,33%	…	6,21%	…	4,35%
REQ009	3,97%	…	5,89%	…	2,52%
REQ010	11,32%	…	12,98%	…	9,07%

Table 5 Assigned values by Rainbowie for a full prioritization with 45 comparisons and criterion 2

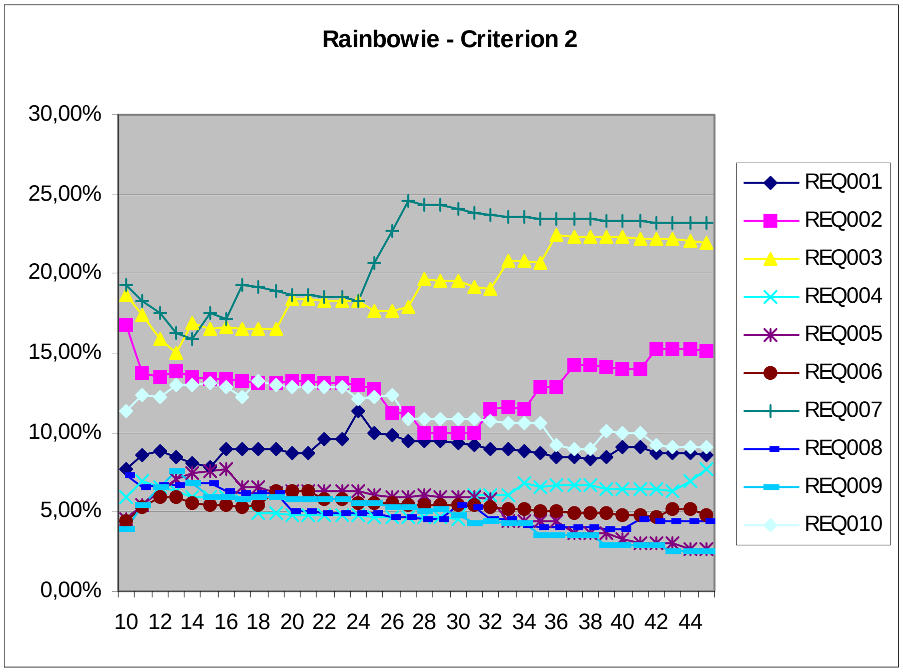


Figure 11 Visualization of assigned values by Focal Point for 45 comparisons and criterion 2

19 equivalent comparisons in Focal Point – criterion 1

	Values after n number of comparisons and internal order at n = 19			
Requirement:	**10**	…	**19**	**Order**
REQ001	14,44%	…	17,35%	1
REQ002	9,16%	…	7,79%	6
REQ003	6,50%	…	5,94%	10
REQ004	13,01%	…	11,96%	4
REQ005	7,44%	…	7,51%	7
REQ006	9,50%	…	10,02%	5
REQ007	8,42%	…	7,41%	8
REQ008	12,13%	…	12,23%	3
REQ009	12,04%	…	12,43%	2
REQ010	7,37%	…	7,37%	9

Table 6 Assigned values by Focal Point for 19 equivalent comparisons and criterion 1

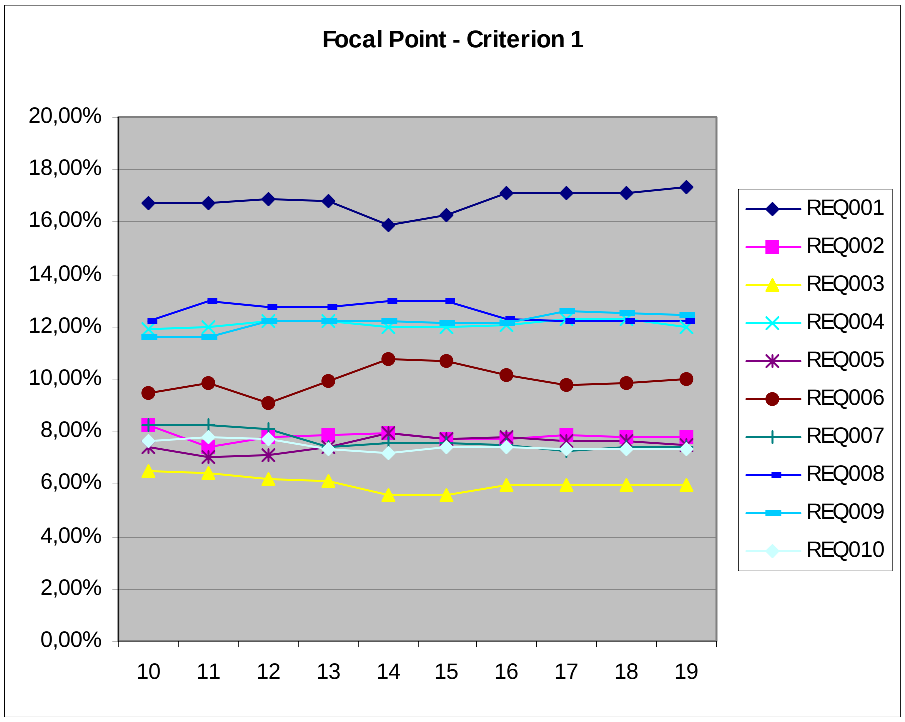


Figure 12 Visualization of assigned values by Focal Point for 19 equivalent comparisons and criterion 1

19 equivalent comparisons in Rainbowie – criterion 1

	Values after n number of comparisons and internal order at n = 19			
Requirement:	**10**	…	**19**	**Order**
REQ001	25,15%	…	17,48%	1
REQ002	7,25%	…	7,91%	7
REQ003	4,32%	…	6,34%	10
REQ004	14,39%	…	11,38%	3
REQ005	2,59%	…	8,56%	6
REQ006	5,87%	…	10,26%	5
REQ007	5,55%	…	7,61%	8
REQ008	14,30%	…	10,83%	4
REQ009	14,71%	…	12,05%	2
REQ010	5,87%	…	7,58%	9

Table 7 Assigned values by Rainbowie for 19 equivalent comparisons and criterion 1

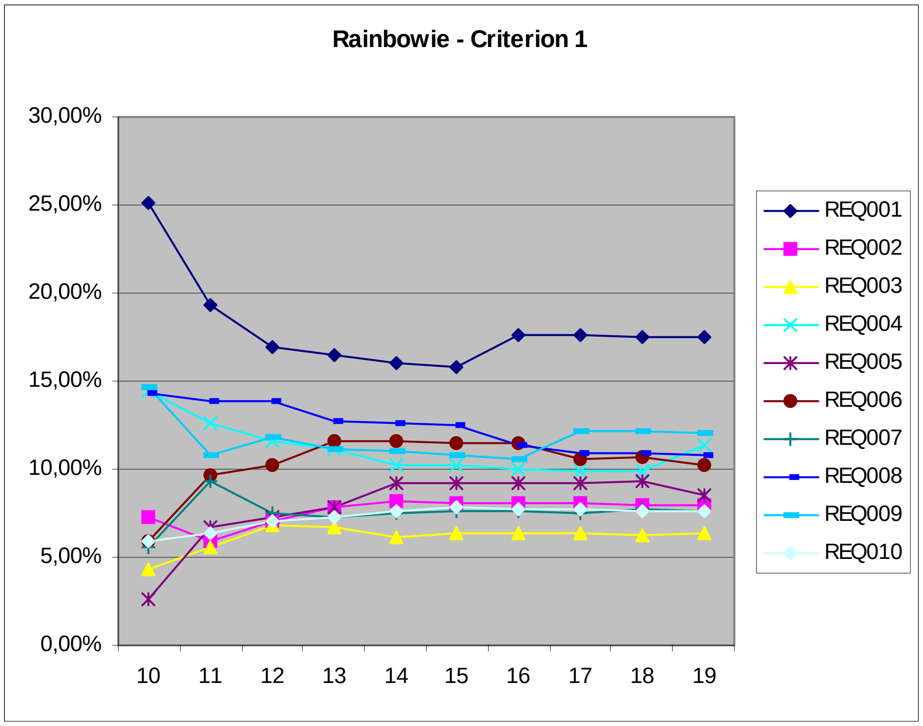


Figure 13 Visualization of assigned values by Rainbowie for 19 equivalent comparisons and criterion 1

19 equivalent comparisons in Focal Point – criterion 2

	Values after n number of comparisons and internal order at n = 19			
Requirement:	**10**	…	**19**	**Order**
REQ001	12,81%	…	10,26%	4
REQ002	17,85%	…	13,58%	3
REQ003	13,95%	…	17,79%	1
REQ004	9,25%	…	9,10%	5
REQ005	5,66%	…	5,61%	9
REQ006	6,60%	…	7,30%	7
REQ007	12,77%	…	17,24%	2
REQ008	7,97%	…	6,48%	8
REQ009	4,06%	…	4,32%	10
REQ010	9,09%	…	8,34%	6

Table 8 Assigned values by Focal Point for 19 equivalent comparisons and criterion 2

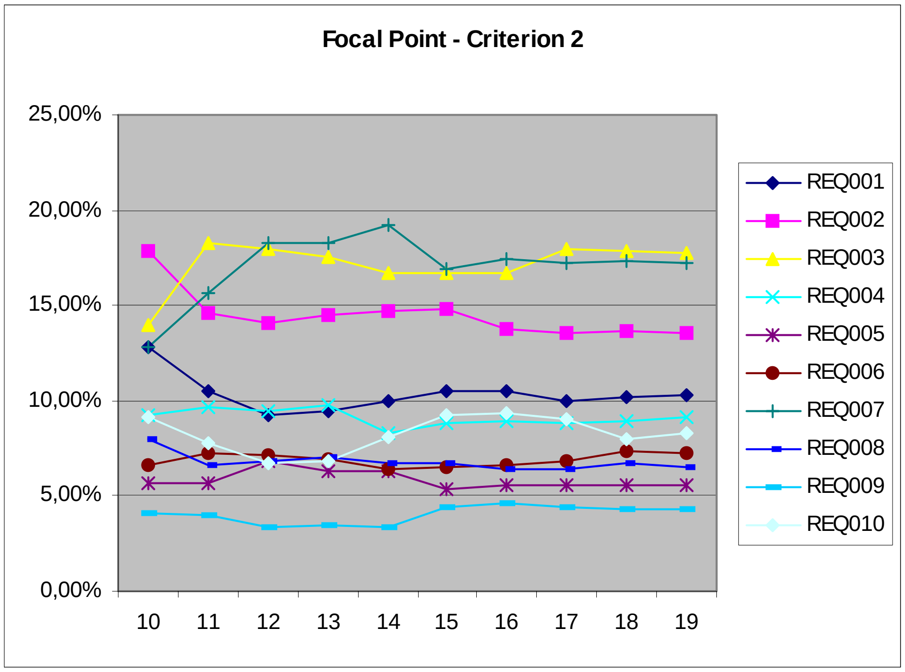


Figure 14 Visualization of assigned values by Focal Point for 19 equivalent comparisons and criterion 2

19 equivalent comparisons in Rainbowie – criterion 2

	Values after n number of comparisons and internal order at n = 19			
Requirement:	**10**	…	**19**	**Order**
REQ001	12,63%	…	7,62%	7
REQ002	24,50%	…	10,26%	5
REQ003	11,57%	…	14,57%	2
REQ004	9,31%	…	12,45%	3
REQ005	4,23%	…	6,80%	8
REQ006	4,44%	…	6,61%	10
REQ007	12,82%	…	14,56%	1
REQ008	9,50%	…	8,17%	6
REQ009	3,42%	…	6,74%	9
REQ010	7,59%	…	12,22%	4

Table 9 Assigned values by Rainbowie for 19 equivalent comparisons and criterion 2

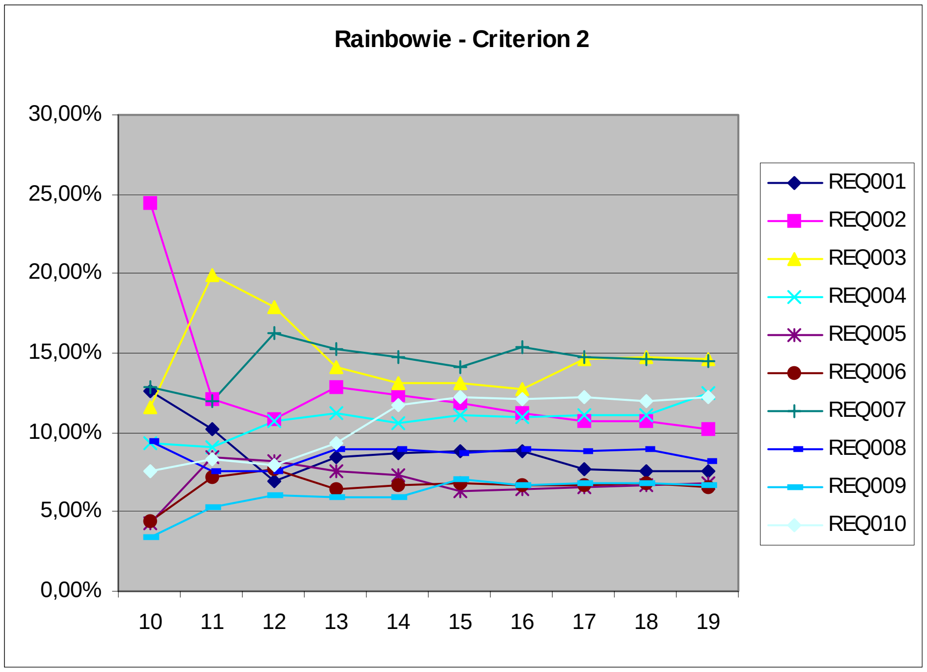


Figure 15 Visualization of assigned values by Rainbowie for 19 equivalent comparisons and criterion 1

19 comparisons in two prioritizations in Focal Point – criteria 1
The “Diff.” column shows the absolute value of the difference between the prioritizations in percentage points.

Requirement:	*Prio. 1*	*Prio. 2*	*Diff.*
REQ001	17,35%	16,23%	1,11 pp.
REQ002	7,79%	7,30%	0,49 pp.
REQ003	5,94%	5,55%	0,39 pp.
REQ004	11,96%	12,37%	0,41 pp.
REQ005	7,51%	7,56%	0,05 pp.
REQ006	10,02%	10,16%	0,14 pp.
REQ007	7,41%	7,72%	0,32 pp.
REQ008	12,23%	13,42%	1,19 pp.
REQ009	12,43%	11,77%	0,66 pp.
REQ010	7,37%	7,90%	0,54 pp.

Table 10 Assigned values for two prioritizations with 19 comparisons in Focal Point and criterion 1

Average difference between the prioritizations: 0.53 percentage points

19 comparisons in two prioritizations in Focal Point – criteria 2

Requirement:	*Prio. 1*	*Prio. 2*	*Diff.*
REQ001	10,26%	9,12%	1,14 pp.
REQ002	13,58%	13,43%	0,14 pp.
REQ003	17,79%	18,11%	0,32 pp.
REQ004	9,10%	7,98%	1,12 pp.
REQ005	5,61%	4,79%	0,82 pp.
REQ006	7,30%	6,83%	0,46 pp.
REQ007	17,24%	16,70%	0,54 pp.
REQ008	6,48%	6,10%	0,37 pp.
REQ009	4,32%	5,88%	1,55 pp.
REQ010	8,34%	11,06%	2,73 pp.

Table 11 Assigned values for two prioritizations with 19 comparisons in Focal Point and criterion 2

Average difference between the prioritizations: 0.92 percentage points.

7 Analysis

The results from the three evaluations have been analysed and compiled in the following sub-sections, together with the conclusions that was made.

7.1 User evaluation of Rainbowie

The outcome of this evaluation was on the whole positive and it was concluded that the program's usability is good enough for users familiar with the PARSEQ method. Further, it was concluded that the program fully, even though not yet always perfectly, supports all the steps in the method.

Some parts of the process, for example the automatically generated graph when the prioritization is done, were found less time-consuming and more flexible than without the tool.

Other parts were considered to be in need of some further program development to achieve an even better support for the method as well as a higher usability. For a list of these improvement suggestions see the list in section 6.2.1. From this list, items 6 and 8 have since been implemented, while the rest will be discussed as suggestions of future development in section 8.2.

7.2 PARSEQ evaluation of the implementation

If the project was to be remade or a new project with a similar process was to be started, the results from the user evaluation would help us to be even more successful.

The three main conclusions drawn is that too little time was spent on discussing and understanding the requirements and the specification, more time should be spent on looking for existing solutions to presented problems and more effort should be put into making earlier prototypes.

The fourth point, concerning requirement's level of description, cannot be influenced by the developers. This is an improvement that must be made by the customer, or the writer of the requirements specification, and is therefore left out as an improvement suggestion for this project.

The main knowledge gained through the PARSEQ evaluation can be put in one sentence: "Spending more time in the beginning of a project to get a picture of what the customer really wants, will help getting the appropriate features in the appropriate release".

7.3 Comparison with Focal Point

As described in section 5.2.3, the comparison between Rainbowie and Focal Point had three main purposes. Each of these will be analyzed in a separate section below.

7.3.1 The pair-wise comparison method as a whole

The first observation made from the results after using the two prioritization tools is the difference in absolute assigned values. This difference is obvious when the values after 45 comparisons are consulted. These values shall be considered to be the true values of the prioritization, i.e. no estimations are done, when the calculation is done by using to the AHP. Since these differ between Focal Point and Rainbowie, a difference must exist in either the AHP calculation or in the translation of relative significance in the comparisons made by user to values used in the calculation. The fact that requirements assigned with equal priority values by Focal Point also are given equal values by Rainbowie indicates the latter. An example of these equal values is that REQ004, REQ008 and REQ009 have all been assigned the value 12,12% for criterion 2 by Focal Point and 13,81% by Rainbowie for the same criterion. This means that Rainbowie probably uses another scale than Focal Point when translating a comparison made by the user to a numerical value for the AHP calculation (for details of this translation, see section 3.5.4).

Another obvious difference is Focal Point's greater ability to in an early stage of the prioritization process stabilize a requirement's priority nearby the final value. In the graphs, this is obvious by the smoother lines in Focal Point's graphs and a more fluctuating evolvement by Rainbowie. The reason for this weakness in Rainbowie is probably a consequence of the modification of the IPC algorithm and will be further described in the next section. As a result of this, Rainbowie will in most cases need more comparisons done by the user than Focal Point to generate a result with the same accuracy. To examine the size of this difference in needed comparisons, a study larger than possible for the scope of time in this thesis must be done and is therefore left out for possible future research.

7.3.2 Effects of the modification of the IPC algorithm

The major modification of the IPC algorithm is the excluded functionality for selecting the comparison with the most potential information to add as the next comparison. The effect of the exclusion of this part, which is presumed to be implemented in Focal Point, is obvious through the bigger fluctuations throughout the prioritization in the values assigned by Rainbowie. When values are stabilized in an early stage of the prioritization in Focal Point, many more comparisons are needed in Rainbowie before a similar behaviour can be observed. In the second part of the comparison between the tools, where the order of 19 comparisons decided by Focal Point is used, a more comparable progress of the assigned values is logically shown. The values calculated by Focal Point seem to stabilize earlier and have a less fluctuating development towards the final values. It must though be pointed out that this evaluation is by far comprehensive enough to draw any definitive conclusions and should rather be seen as an instrument for identifying possible sources of inaccuracy. The lack of functionality for setting up the most efficient order of the list of comparisons in the modified IPC algorithm must however be considered as such a source.

7.3.3 Accepted inaccuracy in industrial applications

To try to get an estimation of the relative extent of the inaccuracy found in the IPC algorithm implemented in Rainbowie, a final examination with Focal Point was done. This examination was in fact a comparison between Focal Point and itself. The intention of this experiment was to examine if there is an accepted degree of inaccuracy in a tool used in industry today. The results from this test show an obvious difference in the results from the two prioritizations with an average divergence of 0,53 respectively 0,92 percentage points. The internal order of the prioritized requirements also differs between the two experiments and the requirements internal ranking is only sustained in eight out of 20 cases. These observations are by no means intended to evaluate the Focal Point tool, instead is their purpose to examine to accuracy of the pair-wise comparison method in general. The conclusions drawn from this part of the evaluation is that there is always an element of inaccuracy when an incomplete form of the pair-wise comparison method is used. This inaccuracy must be taken into mind when doing an incomplete pair-wise comparison, especially when Rainbowie is used since the implemented IPC algorithm is simplified. The values calculated by Rainbowie when an incomplete pair-wise comparison has been made shall not be considered as any truly accurate values. The interpretation of these shall rather be of a guiding nature.

8 Discussion

In this section we discuss our general reflections over the project. Since this is the first tool in its area, developed over only 20 weeks, it is fair to assume that the tool will continue to being developed together with the PARSEQ method itself. Because of this, a range of improvement suggestions is also presented to reflect our opinions of where to focus future development.

8.1 General reflections over the project

Some parts of the project have been more complex, and needed more time, to implement than other parts. Especially complex was the theory and implementation involving AHP and in particular the IPC algorithm. The development of the graphical user interface for the PG method, also belong to the more time-consuming areas of the project. Other parts, such as the implementation of the $100-technique, have in comparison been quite simple and also much less time-consuming.

8.2 Suggestions from the user evaluation

Here the suggestions gotten from the user evaluation is described in more detail together with a discussion of what they would do for the program.

Importing old priorities

The program currently only has support for importing and handling release numbers.

Support for importing and handling earlier priorities means that it instead of comparing the new priorities to which release the requirements were implemented in, it would be possible to directly compare the old priorities to the new ones.

Undo functionality

Throughout the entire program there are several areas that would benefit from an extensive undo support.

The main reason for good undo support is to achieve a higher usability via more forgiving interaction.

Change priorities

Make it possible to change and adjust priorities in an easy way later in the process. If it, for example, in the graph-window is discovered that a requirement has mistakenly been given a wrong priority.

Then, it would save a lot of time if it were possible to correct that mistake, without having to redo the entire prioritization step from scratch.

Merge, group and split cards
The ability to merge, group and split cards in the PG would help if it, during the prioritization, is discovered that requirements are poorly formulated or have strong relations to each other.

Drag cards between piles
In the planning game, it would be easier and faster to move cards between piles if it is possible to directly drag a card from one pile to another, without having to put the card on the desk first.

Sliders in $100-window
A slider in $100-window that can be used to assign the $. When one slider is pulled up, the other sliders should be lowered, to always keep the total amount of used $ to $100. Also, implement functionality to lock desired sliders, so that they will not be changed when another is pulled.

This would mean that the user do not have to think about how many $ that are left to be used.

Improvement suggestion and root-cause mapping
In the root-cause matrix, there currently is only one improvement suggestion column for each root-cause.

The possibility to map several improvements suggestions to one root-cause might make the program more flexible.

Graph-window and root-cause matrix mapping
Visualize in the graph window which requirements that have been added to the root-cause matrix.

This is another usability improvement, which would help the user to keep track of which requirements that have already been added to the root-cause matrix.

Different layout of support lines in the planning game graph
In one view mode in the planning game graph, the area is divided in three equally large areas corresponding to the boxes during reprioritization. This means in practice that the distance between the graph points within each square is dependent on the number of points in that square. This should be changed to have the lines drawn with one third of the requirements in each square, in other words the lines shall no longer correspond to the boxes in the reprioritization.

This change should be made because the scale used in planning game is just ordinal and a difference in distances between the points can mislead the user to believe that the scale is non-ordinal.

8.3 Suggestions from the Focal Point comparison

According to the analysis in section 7.3, the implemented IPC algorithm is one section that could gain improvement by future development. The algorithm needs to be improved if Rainbowie's ability to perform an incomplete pair-wise comparison shall be comparable with industrial tools in the area, as for example Focal Point. If the algorithm were expanded with functionality to calculate the optimal next comparison to make according to the IPC theory in section 3.5.3, the results given by Rainbowie would most likely be more similar to Focal Point's results.

Another suggestion of improvement of the implemented IPC algorithm is to add a stopping rule that also is described theoretically in section 3.5.3. As the algorithm is implemented today, the decision of stopping the prioritization and continuing to the next step in the process is entirely left to the user and no aid is given by the tool. This fact assumes that the user has a considerable knowledge about the prioritization method itself and the possible sources of inaccuracy present to take such a decision. To avoid this demand on the user and create a tool less dependant on user expertise, some sort of stopping rule needs to be implemented.

To summarize the improvement suggestions for the PWC part of the program, this section has significant enhancements to gain from future development. Besides from more usability oriented improvement suggestions listed above, this area should probably be kept in focus if future development of the program would take place. A complete IPC algorithm should probably be the main objective of such a development.

9 Conclusions

This Master Thesis has given two main results.

First of all a software tool has been created that supports all four steps of the PARSEQ method. The functionality of the software has been evaluated and tested by the probable future users. The evaluation showed that the work put into this thesis has proven to give good results and a well functioning software prototype. The resulting software is likely going to be used in future research and the software itself will also probably be developed even further.

Secondly, we have gained knowledge and gotten more familiar with retrospective analysis in general and the PARSEQ method in particular. The knowledge gained concerning retrospective analysis has opened our eyes for an interesting area within the requirements engineering niche.

Further, the thesis has given us more experience and insight into the areas of software- and requirements engineering. During the work of this thesis, we have been given the opportunity to apply the theoretical knowledge obtained during our education in the largest real life project we have been a part of so far.

Through the experience we have gotten in software development in this thesis, there are a number of things that we would do differently if a similar, larger scale, project was to be done: We would have tried to have a more structured and formal requirements engineering process, since this is a key area of a successful project. Also, a tool for configuration management would be necessary. We felt that the way we handled the configuration management in this project was sufficient, however on the edge of growing too large to continue without a tool. Finally, another area that would help save some time and improve the development process is some kind of testing procedures. This could for instance be something like a structured test first policy or automated testing.

10 References

[1] Sommerville, I, *Sofware Engineering, 6th edition*, Addison-Wesley, 2001

[2] Lauesen, S, *Software Requirements, Styles and Techniques*, Addison-Wesley, 2002.

[3] Boehm, B. W, “A Spiral Model of Software Development and Enhancement”, *IEEE Computer*, 21(5), 61-72 (Chs. 3,4), 1988

[4] Karlsson, J., *Marknadsdriven produktledning – från kundbehov till lönsamma produkter,* Focal Point AB

[5] Karlsson, L., T., Berander, P., Regnell, B., Wohlin, C., ”Requirements Prioritisation: An Experiment on Exhaustive Pair-Wise Comparisons versus Planning Game Partitioning”, *Proceedings of the 8th International Conference on Empirical Assessment in Software Engineering,* Edinburgh, Scotland, UK, 2004

[6] Beck, K., *Extreme Programming Explained – Embrace Change*, Addison-Wesley, 2000

[7] Harker, P. T., ”Incomplete Pairwise Comparisons in the Analytical Hierarchy Process”, *Mathematical Modelling*, Vol 9, pp. 837-848, 1987

[8] T.L. Saaty, *The Analytic Hierarchy Process*, McGraw-Hill, New York, 1980

[9] Karlsson, J., Ryan, K., “A Cost-Value Approach for Prioritizing Requirements”, *IEEE Software*, pp. 67-74, September/October, 1997

[10] Ji, P., Jiang, R., ”Scale transivity in th AHP”, *Journal of the Operational Research Society*, Vol. 54, No. 8, pp.896-905, 2003

[11] Leffingwell, D., Widrig, D., *Managing Software Requirements - A Unified Approach*, Addison-Wesley, 2000

[12] Godfrey, A.B., “The Santayana Review”, *Quality Digest*, February, 1999

[13] Rising, L., Derby, E., “Singing the Songs of Project Experience: Patterns and Retrospectives”, *Cutter IT Journal*, Vol. 16, No. 9, pp. 27-31, September, 2003

[14] Nolan, A.J., “Learning from Success”, *IEEE Software*, pp. 97-105, Jan/Feb, 1999

[15] Kerth, N.L., *Project Retrospectives: A Handbook for Team Reviews*, Dorset House Publishing, 2001.

[16] Birk, A., Dingsøyr, T., Stålhane, T., ”Postmortem: Never Leave a Project Whitout it”, *IEEE Software*, pp. 43-45, May/June, 2002

[17] Dingsøyr, T., Moe, N.B., Nytrø, Ø., ”Augmenting Experience Reports with Lightweight Postmortem Reviews”, *3rd International Conference on Product Focused Software Process Improvement*, Kaiserslautern, Germany, September, 2001.

[18] Karlsson, L., Regnell, B., Karlsson, J., Olsson, S., “Post-Release Analysis of Requirements Selection Quality – An Industrial Case Study”, *9th International Workshop on Requirements Engineering: Foundation for Software Quality,* Velden, Austria, 2003

[19] Jeffries, R., Anderson, A., Hendrickson, C., *Extreme Programming Installed*, Addison-Wesley, 2001.

[20] Whitgift, D., *Methods and Tools for Software Configuration Management*, John Wiley & Sons, Inc., 1991.

[21] Preece, J., Rogers, Y., Sharp, H., *Interaction design: Beyond human-computer interaction*, John Wiley & Sons, Inc., 2002.

[22] Pol, M., Teunissen, R., van Veenendahl, E. *Software Testing – A Guide to the TMap Approach*, Addison-Wesley, 2002.

[23] Stevens, P., Pooley, R., *Using UML: Software Engineering with Objects and Components*, Addison-Wesley, 2000

[24] http://ieee.org (visited 2005-05-25)

[25] http://www.ne.se (visited 2005-05-25)

[26] http://www.eclipse.org (visited 2005-05-25)

[27] https://www.cvshome.org/ (visited 2005-05-25)

[28] http://www.cvsnt.com/ (visited 2005-05-25)

[29] http://www.wincvs.org/ (visited 2005-05-25)

[30] http://www.focalpoint.se (visited 2005-05-25)

[31] http://www.andykhan.com/jexcelapi/ (visited 2005-05-25)

11 List of figures and tables

Tables

12 Appendix

Appendix A – Technical terms and abbreviations

AHP	The Analytic Hierarchy Process, designed by T.L. Saaty [8], is a model that includes pair-wise comparisons and is used to aid decision-making.
CI	Consistency Index is an indicator of the accuracy of pair-wise comparisons.
CR	Consistency Ratio defines the accuracy of pair-wise comparisons. It is the ratio between the consistency index and the random indices.
CVS	Concurrent Versions System is a free and well-spread versions control system.
CVSNT	Is just like CVS a free and well-spread versions control system but with many more features.
GUI	Graphical User Interface
IPC	Incomplete Pair-wise Comparisons is a method developed by Harker [7] to reduce the number of necessary comparisons in the normal pair-wise comparisons method.
J2SE	Java 2 Platform, Standard Edition. A Java development environment from Sun Microsystems.
KJ session	A method for collecting and structuring data developed by and named after the Japanese ethnologist Jiro Kawakita.
KK*xxxx*	Quality Requirement number *xxxx* (KvalitetsKrav). Refers to the requirements in the requirements specification.
UK*xxxx*	Development Requirement number *xxxx* (UtvecklingsKrav). Refers to the requirements in the requirements specification.
PARSEQ	Post-release Analysis of Requirements SElection Quality.
PG	Planning Game, the prioritization technique discussed in section 3.5.1.
PK*xxxx*	Product Requirement number *xxxx* (ProduktKrav). Refers to the requirements specification.

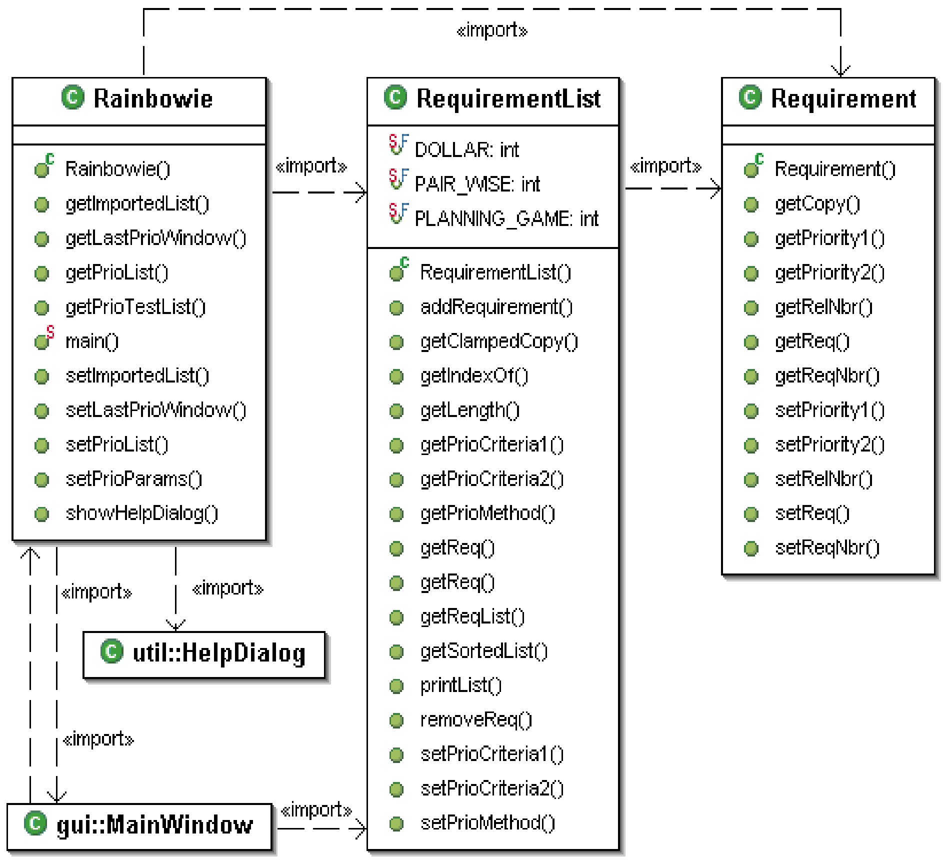
PWC	Pair-Wise Comparisons, the prioritization technique discussed in section 3.5.2.
RI	Random Indices, the consistency indices of randomly generated reciprocal matrices.
SDK	Software Development Kit.
SQL	Structured Query Language is a standard interactive and programming language for getting information from and updating a database.
WIMP	Windows, Icons, Mouse and Pull-down menus, abbreviation used in interaction design.
WINCVS	A graphical user interface for CVS.
XP	eXtreme Programming, is the software development technique discussed in section 5.1.
$100-test	The prioritization technique described in section 3.5.5.

Appendix B – Detailed software architecture

This appendix will present a detailed description of the Rainbowie systems internal software architecture concerning packages, classes and relations. Each package will be presented with a class diagram in UML [23] notation and a description of the internal class structure and functionality. For visibility reasons, the class diagrams are not displayed to their full extent. The intention of presenting these diagrams is to explain the internal package structure rather than to illustrate a complete class description. Related classes in other packages than the currently described are presented with compressed class symbols.

The rainbowie package

In the rainbowie package, classes considered to be universal and frequently used throughout the program are located. These classes are three in total and are structured as follows.



Rainbowie.java
This is the main class in the system. The main method of the program can be found in this class together with the essential lists containing imported and reprioritized requirements. When other parts of the system have completed their tasks, as for example the requirements reprioritization, the affected list in this class is directly updated.

In addition, the main window of the program is initialized from an instance of this class. The class is also responsible for showing the help dialog when the user opens it from any of the windows in the program.

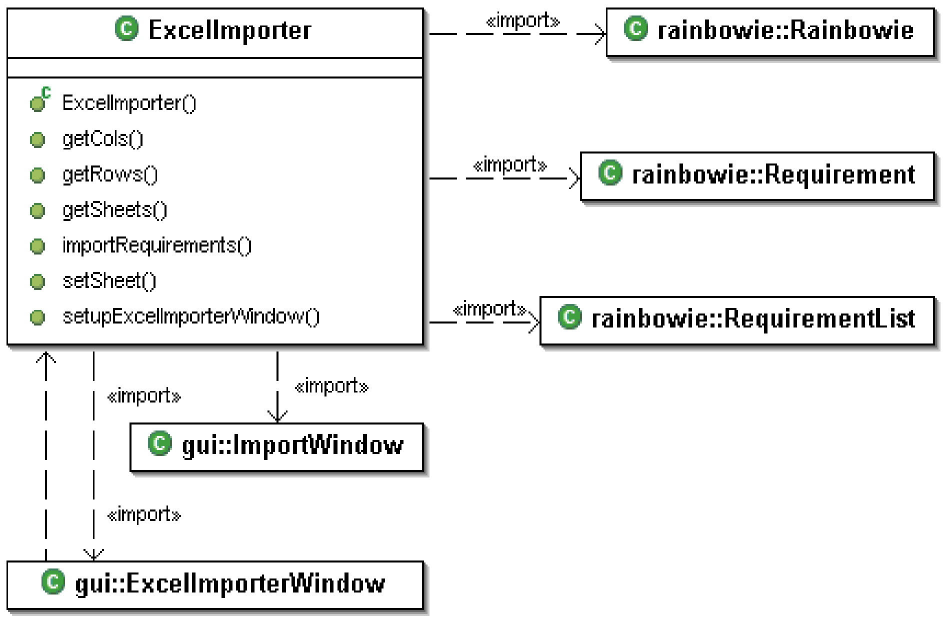
Requirement.java
A requirement in the program is represented by an instance of this class. The class is mainly a holder of the attributes for the represented requirement, such as the requirement number, requirement description and release number. The class also holds attributes for the assigned priorities for the requirement. Corresponding to these attributes, the class also holds a range of get and set methods and a method to create a copy of the requirement.

RequirementList.java
All requirements in the program are collected in requirement lists. These lists are represented by an instance of this class. If the list has been reprioritized, the class stores information of the prioritization technique and criteria used.

Together with the functionality mentioned, this class also contains algorithms needed to create lists sorted by specified parameters.

The importer package

Since the program at this moment is capable of importing requirements from Excel only, this package contains just one class.

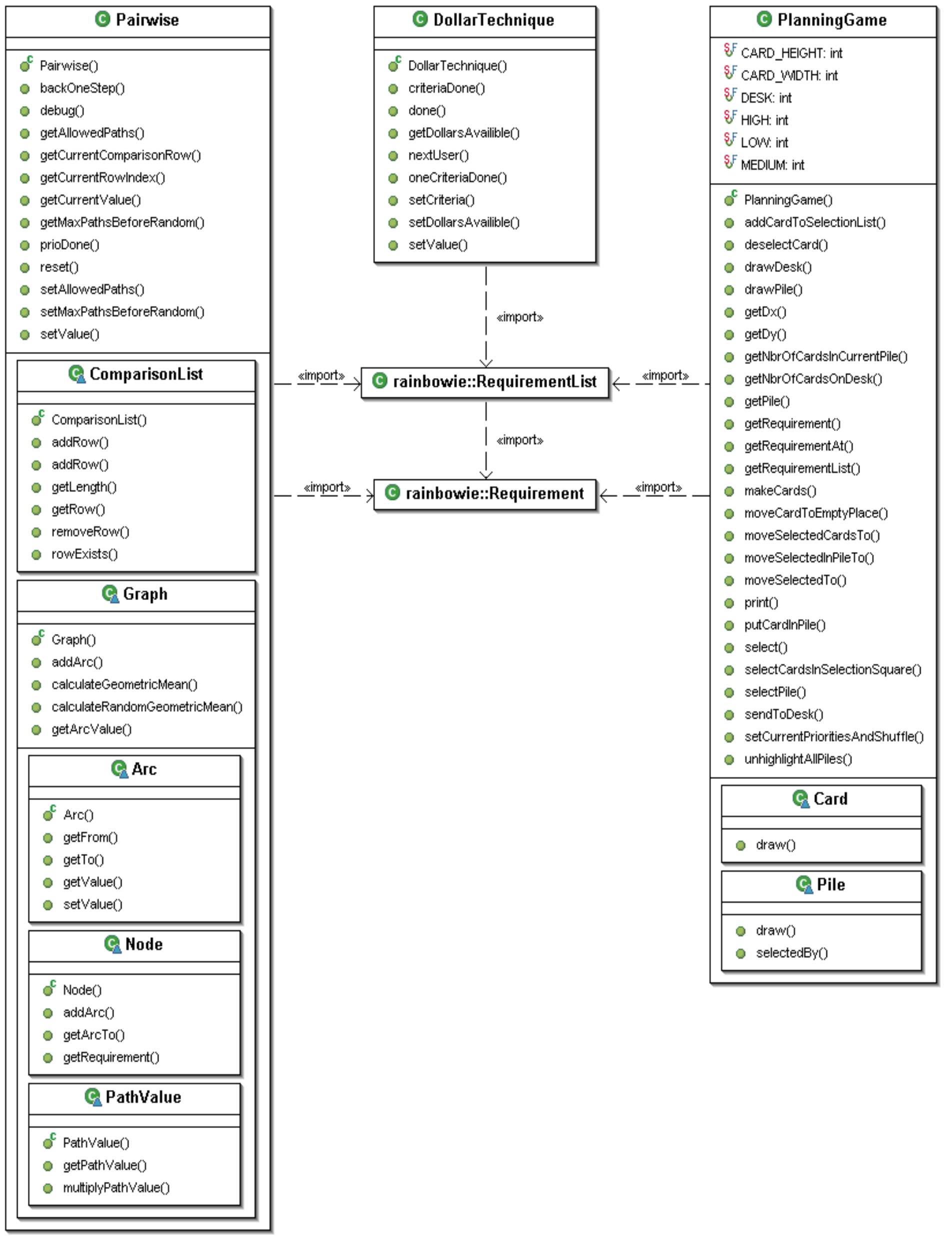


ExcelImporter.java

This class contains functionality to access an Excel spreadsheet. All interaction with Excel files is done by dynamically linking to the open source JExcel API written by Andy Khan [31]. The information collected in the spreadsheet is transferred to the graphical user interface to make the requirements selection possible. When the user has selected the desired requirements, an import is done and the requirement list in the Rainbowie object is updated.

The prioritizer package

All needed calculations and data storage during the reprioritization phase of the PARSEQ process is made in classes in the prioritizer package. Each prioritization technique has a corresponding class in the package for the specific functionality needed. These classes do their calculations and store results in requirements and requirement lists distributed to them.



PlanningGame.java
The class corresponding to the Planning Game technique has the main task to keep track of the cards on the desk and in the piles. Each requirement in the reprioritization corresponds to an instance of the internal Card class. In the PlanningGame class, a linked list of cards to be found on the desk is placed. The three piles located on the desk are represented by three instances of the internal class Pile that also holds a linked list of the cards located in that pile. Each card is then stored in exactly one linked list depending on whether it is in a pile or on the desk.

Other functionality of importance located in this class concerns handling of cards in the window. These methods are for instance responsible for creating the cards and randomly placing them on the desk, detecting whether a card is selected or not by a mouse click in a specified position and handling movement of cards when the user is dragging it with the mouse.

In the internal Card class, a reference to the corresponding requirement is found together with information about the cards' position and how is shall be drawn.

In the Pile class, as mentioned above a linked list containing the cards in the pile is located. Together with this functionality for detecting if a dragged card can be dropped in the pile, if the pile can be selected by a mouse click and drawing instructions can be found.

Since the prioritization values in the Requirement objects are stored as double values, the prioritization order is coded in a special way to handle belongings to different piles. Therefore the stored double value is divided into two parts: The integer part of the value represents the requirements ordinal ranking in the pile, starting by 0 for the first position. The decimal part decides which pile the card is placed in, 0 for "High", 1 for "Medium" and 2 for "Low". To illustrate; the card placed first in the "High" pile will be assigned the priority value 0.0 and the third card in the "Medium" pile will be given the value 2.1.

DollarTechnique.java
The class used for prioritizing with the $100-test has four main parameters to keep track of.

1. The number of dollars used in total during reprioritization. If more than one user do the prioritization, this value must be stored.
2. The number of dollars used so far by the user for the current prioritization criteria.
3. The number of dollars assigned to the *first* priority criteria for each requirement.

4. The number of dollars assigned to the *second* priority criteria for each requirement.

Additionally, the class hold functionality for calculating and for each requirement storing the final priority values when the reprioritization phase is done.

Pairwise.java

In the class for pairwise comparisons the internal structure formed in three main parts. All general functionality is located in the Parwise class itself. This type of functionality handles for example setting and getting values parameters and priority values. Also functionality for setting up the reprioritization and forward and backward iterating through the comparisons can be found here.

An internal class called ComparisonList keeps track of all possible pairwise comparisons in the requirement set. Each pairwise comparison is represented by an Object array referred to as a row in a linked list located in the internal class. The row elements are in the following format:

1. The first requirements’ position in the original requirement list
2. The first requirements’ description
3. The second requirements’ position in the original requirement list
4. The second requirements’ description
5. The assigned priority value for the current comparison

The assigned value is in the format of the relative importance of the first requirement compared to the second requirement, i.e. if the first requirement is of higher significance than the second; the assigned value is greater than one. For details of pair-wise comparison, see section 3.5.2.

When the comparison list is set up, a semi-random algorithm in the Pairwise class is used. A fully random algorithm can not be used since the use of incomplete pair-wise comparisons sets restrictions for the *n* first comparisons, where *n* = the number of requirements in the reprioritization. See section 3.5.3 for details on incomplete pair-wise comparisons. The algorithm therefore works as follows:

1. A copy of the requirement list is created, the copy is hereafter referenced to as *the list*.
2. A first requirement is randomly chosen within the list and then removed from it.

3. A second requirement is randomly chosen and removed from the list.
4. A comparison is set up in the ComparisonList between the two requirements.
5. The requirement previously considered as second is set as first requirement and a jump to step 3 is done if the list still contains any requirements.
6. When the list is empty, a final comparison is set up between the last and the very first requirement removed from the list.
7. When all *n* comparisons have been set up, the next step in the algorithm starts.
8. A set of all remaining comparisons is created and randomly inserted into the ComparisonList after the initial *n* comparisons. All comparison pairs are also internally switched by a probability of 50%, i.e. the first requirement becomes the second and vice versa,. This is done to prevent the user from being influenced by for example one requirement always appearing on the left hand side in the comparison window.

The comparison list is then ready to be iterated during the progress of the users' reprioritization work. For each prioritization criteria, a new setup procedure is done.

The other internal class defines the graph used in the calculations of the incomplete pair-wise comparison method. In this graph another internal class, called Node, represents every requirement. Comparisons between two requirements, arcs between nodes in the graph, are defined by instances of the Arc class. Together, the nodes and the arcs define the graph used to calculate a geometric mean of all possible, or a randomly chosen set of all possible, paths from one requirement to another. This geometric mean is then used as an approximation of the direct comparison between the two requirements. The geometric mean of a set of *n* numbers is defined as follows [7]:

$$G(a_1 ... a_n) \equiv \left(\prod_{i=1}^{n} a_i \right)^{1/n}$$

If the number *n* is unknown until the very end of the calculation, i.e. the total number of possible paths is never known until all possibilities have been examined, the product within the parenthesis might grow enormously and cause an memory overflow problem if ordinary multiplication is used. Because of this problem, another internal class called PathValue is created to handle this calculation.

The graph has the ability to calculate the geometric mean of paths in two ways. The first way to solve the problem is to choose to examine all possible paths in the graph between two nodes by a depth-first algorithm. Since the number of paths grows exponentially when arcs are added, see section 3.5.3, this method is preferably used when few comparisons have been completed. The other way the graph can calculate the geometric mean is to use a randomly chosen set of paths. The algorithm to find a random path between a specified start and end node works as follows:

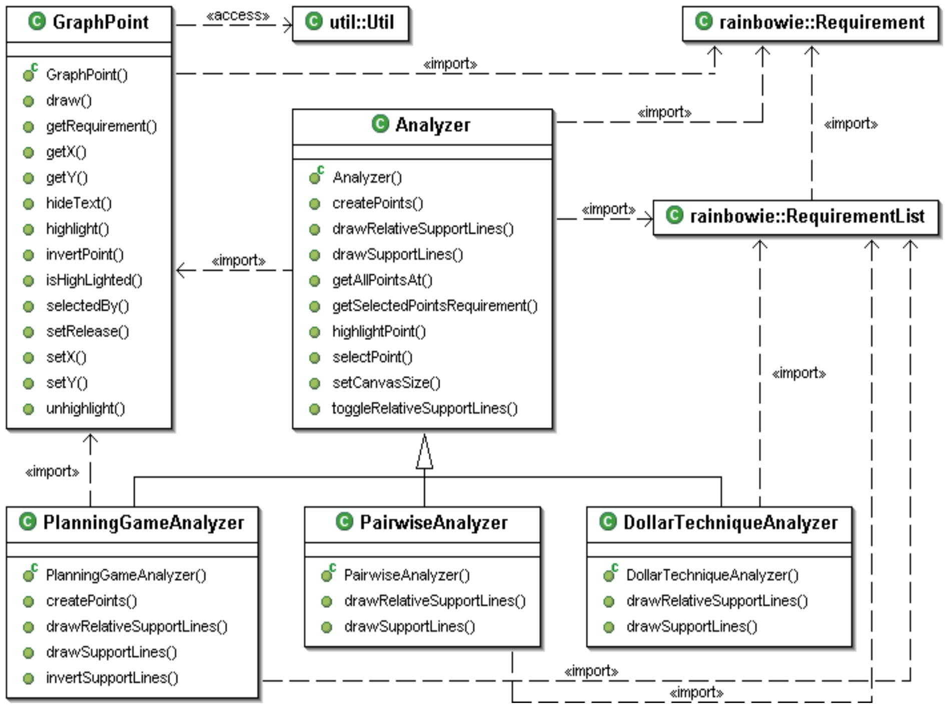
1. Set the specified start node as the current node and mark the node as visited.
2. Among all outgoing arcs to unvisited nodes from the current node, randomly select one.
3. Set the node at the end of the selected as the current node and mark the new node as visited.
4. If the new node is the specified end node, exit the algorithm and add the path to the set of randomly chosen paths, if not jump back to step 2. If the current node has no outgoing arcs to unvisited nodes, a dead end is reached and the algorithm starts over at step 1 again.

In both approaches to find a set of paths, recursive algorithms are used for graph traversing. The threshold values used for deciding when to switch to random mode and how many random paths to use are stored as modifiable parameters in the Pairwise class.

After each comparison made by the user, the priority values and consistency ratio are calculated by using the theory behind AHP, see section 3.5.4. To make this possible, a couple of matrix calculation methods are implemented.

The analyzer package

When the requirements are reprioritized, the assigned priority values shall be translated to positions in the graph used in the root-cause analysis. For each prioritization method this translation is done in a distinctive way. Each class responsible for the translation of requirement priorities due to a corresponding prioritization method is a subclass of the Analyzer class.



GraphPoint.java

The physical representation of a requirement in the graph used in the root-cause analysis is an instance of the GraphPoint class. This class holds information of the points' position, a reference to the matching requirement, whether or not the point is highlighted and information of how to draw the icon. The GraphPoint also contains a method called to relocate the point when the axes in the graph have been inverted. Another method in this class is responsible to detect if a point in the graph is selected by a mouse click in a specified position.

Analyzer.java

In the superclass of all analyzer classes, general methods used for priority value translations can be found. These methods are mainly responsible for:

1. Finding the largest assigned priority value. This value is then used to adjust the scales on the axes to ensure that the whole graph area is used.
2. Finding the number of present release numbers and assigning each a unique index number. This is done to make each release be drawn with a specific icon.
3. Creation and positioning all GraphPoint objects needed. By default, GraphPoint objects are positioned by doing a straight transformation of the assigned priority value to a corresponding position in the graph, e.g. considering the assigned values as a numerical value of their significance due to the currently used prioritization criteria. If another use of the assigned priority values shall be done, this part of the class must be overridden by the subclass.
4. Drawing the support lines shown in the graph. The default way to represent these is to draw two upward sloping lines, *y = 2x* and *y = 0.5x*, starting in origin. The lines can also be adjusted to instead divide the set of points in the graph in three equal groups by making the lines more or less steep. The look of these lines can differ in various ways between the prioritization methods and therefore is also this part sometimes overridden by a subclass.

Besides this functionality, methods for detecting selection of one or more points in the graph by mouse clicks can be found in this class.

PlanningGameAnalyzer.java
The result from a reprioritization made by the Planning game technique differs particularly in two ways from the case where a default GraphPoint translation in Analyzer can be used. First, the special coding of the priority values, described in the PlanningGame.java section, must be taken care of. The second major difference is that the graph's support lines in this case represent the three piles used during reprioritization in two dimensions, one for each criteria. The relation between the piles and the graph points positions can be generated in two ways. Either the piles, illustrated by the graphs' support lines, are equally wide and high, meaning that the distances between graph points is relative to the number of requirements in each pile. The other way to generate graph points and support lines is to place the graph points with equal distances and instead adjust the widths and heights of the piles.

Because of this prioritization technique specific behaviour, both the method for creating graph points and the methods for drawing support lines in Analyzer are overridden by this class.

PairwiseAnalyzer.java
When graph points are generated after a pairwise comparison reprioritization, the default method in Analyzer is used. The only technique specific functionality in this class is instead the creation of labels for the axes. The highest assigned priority value found is translated to a value in percentage notation and written out at the end of both axes. In all other calculations, the default methods in Analyzer are used.

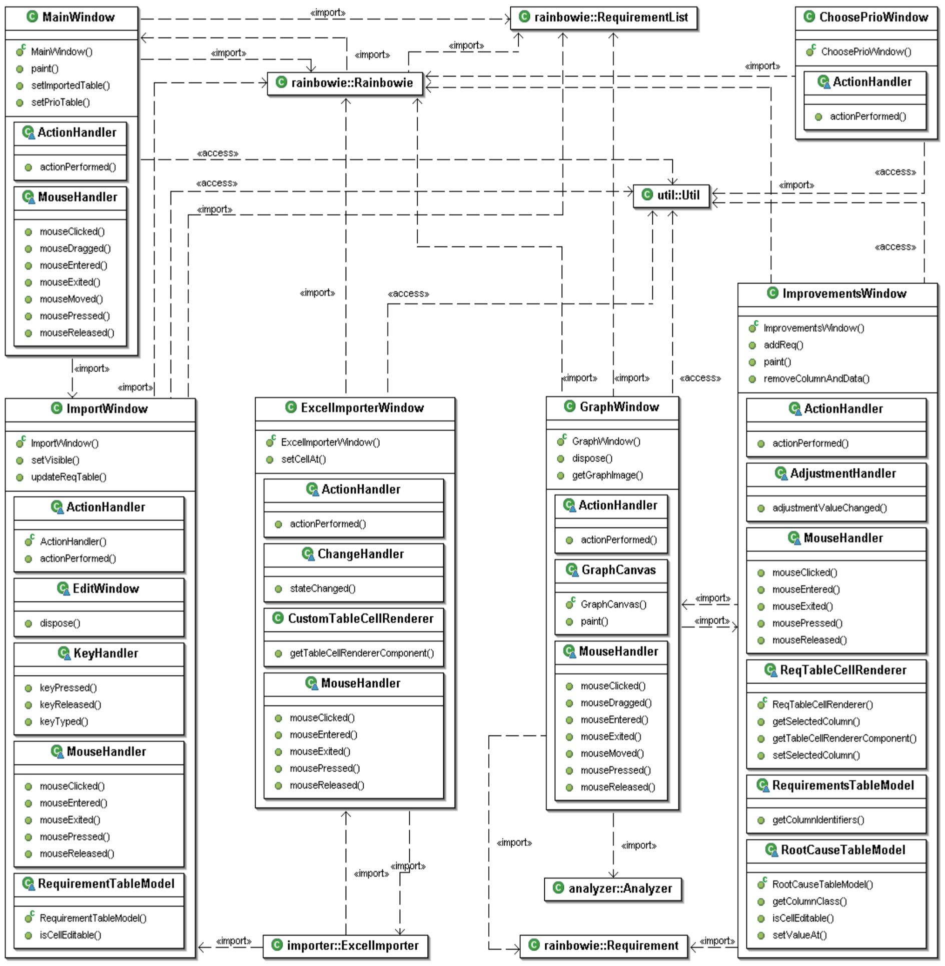
DollarTechniqueAnalyzer.java
This method is identical with PairwiseAnalyzer in its functionality except that the label created is written as a $ value instead of a percentage value.

The gui and gui.prio packages

The classes responsible for all windows and their direct interaction with the user are divided into two packages. Windows involved in the requirements reprioritization phase are found in a package called prio, located as a sub package to the gui package. In the gui package, all windows not related to reprioritization activities are located.

Common for all window classes in the program is that they are subclasses of the class JFrame in the package javax.swing.

In this section, all classes in the gui package will be described first followed by the ones in the gui.prio package.



MainWindow.java

The first window the user meets is the main window of the program. In this window two tables, a menu bar and a set of buttons is located. Via the buttons or the menu bar, the user can initialize the next step in the PARSEQ process and via the tables the progress can be tracked. The updating of the tables throughout the process is done from the programs Rainbowie object. The action taken from the user is in this class, as well as all other window classes, is registered and handled by internal classes implementing the ActionListener and MouseListener interfaces.

ImportWindow.java
During the import step of the process, a listing of the so far chosen requirements is displayed in a table in this window. The last column of this table is directly editable while editing of any of the other two demands an instance of a special edit window. The EditWindow class is an internal class of the ImportWindow and is visualised when the user either double-clicks on a desired row in the table or presses a specified button. To create a faster but more complex way of editing requirement rows for advanced users, a class implementing the KeyListener interface is created to allow keyboard shortcuts. When the user is satisfied with the set of requirements to import, a press on the accept button will update the table of imported requirements in the main window via the Rainbowie object and the import window will be closed.

ExcelImporterWindow.java
When the user selects to import requirements from an Excel spreadsheet, an instance of this class is created and displayed on the screen. In this window a table corresponding to the selected spreadsheet is displayed. The class is responsible for taking care of user interaction made for selecting which cells to import requirement numbers, requirement descriptions and release numbers from. This information is then passed on to the ExcelImporter class, described above, which imports the selected requirements to the program.

ChoosePriowindow.java
When the reprioritization phase of the process is about to start, the user must select which prioritization method to use and which criteria to prioritize after. These actions are taken in the ChoosePrioWindow. The window consists mainly of three method buttons, one for each prioritization technique, and five checkboxes for the criteria, three pre-defined and two own definition possibilities. When the user has selected one technique and two criteria a press on the Ok button will open the appropriate prioritization window with the selected criteria. The main responsibility of this window is to register the choices made by the user and then pass them on to the Rainbowie object.

GraphWindow.java
The graph windows main task is to display the graph points representing the reprioritized requirements in a plot on the screen. The graph can be redrawn in several ways depending on a variety of parameters set by the user. These parameters can for example be different kinds of support lines as mentioned above, graph points with or without the requirement description and with the axes of the graph inverted. All drawing of

graph points and support lines is made on the graphics of the internal class GraphCanvas.

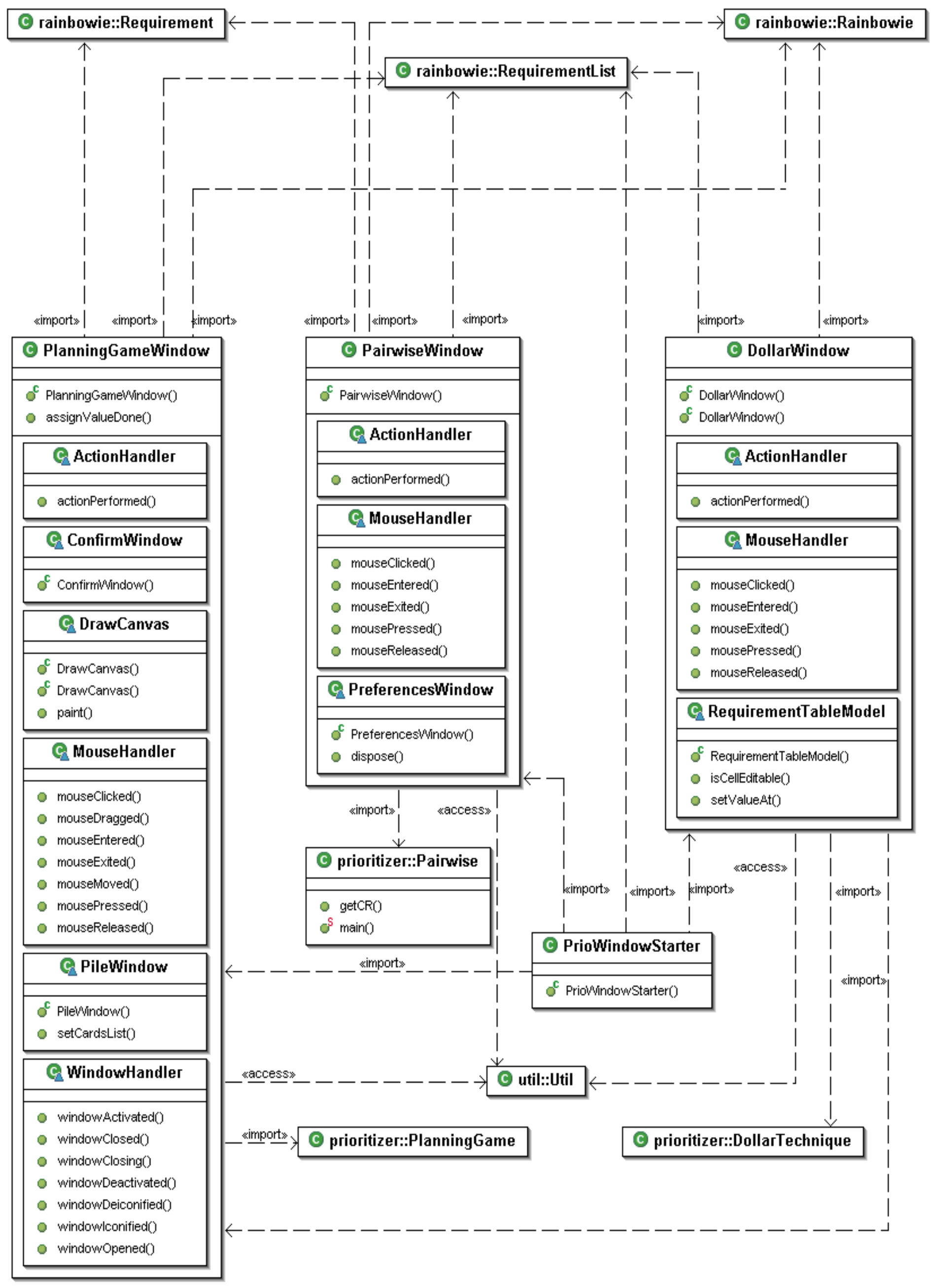
The window class can also detect user interaction taken to select requirements for further analysis in the root cause matrix. Instantly when a GraphWindow object is created, a corresponding ImprovementsWindow, holding the root cause matrix, is instantiated. A double click on a graph point, or a press on the Analyze button when a graph point is selected, will add the corresponding requirement to the root cause matrix by direct interaction between the classes. Other user interaction that is detected by the class is if the user drags the mouse on the graph to create a dependency line.

In addition to the functionality mentioned above, the class can also save the graph image as a png file with a name and at a location selected by the user.

ImprovementsWindow.java

The root-cause matrix used for root cause analysis and improvements elicitation can be found in the class ImprovementsWindow. As mentioned above, this window is opened from an instance of the GraphWindow class. The root-cause matrix is in fact two linked tables, one for the root-causes and the improvements and one for the requirements. Functionality for adding and removing rows and columns is implemented to support working with the root-cause matrix. The information entered in the cells of the tables can be exported to an Excel spreadsheet via the ExcelWriter class in the util package.

Since this is the last step in the PARSEQ process, information is not distributed anywhere else except for the Excel export possibility.



PrioWindowStarter.java
When the user has selected a prioritization technique and two criteria, an instance of this class is responsible for starting the right window. This is the only functionality found in this class. The only reason of this class' existence is to make a future program extension of additional prioritization techniques easier.

PlanningGameWindow.java
During a planning game reprioritization, an instance of the PlanningGame class is responsible for displaying the desk with the cards

on the screen. Functionality for dragging and dropping one or several cards with the mouse is implemented to make the prioritization possible.

The class have three internal classes of significant interest. First a class called DrawCanvas that is responsible for all drawing during the reprioritization. The second defines the window that represents an opened pile. This window is used for internal sorting in one pile and is therefore called PileWindow. Finally, a minor window used for letting the user see the complete ordinal prioritized list of requirements before the reprioritization can be accepted is located as an internal class. This internal class, called ConfirmWindow, is also responsible for staring a DollarWindow if a combined planning game - $100-test reprioritization shall be done. This type of reprioritization is described more in detail in section 6.1.1.

PairwiseWindow.java
The window used for reprioritization with the pairwise comparison technique is mainly responsible for forwarding the user input to the Pairwise class where all calculations are done.

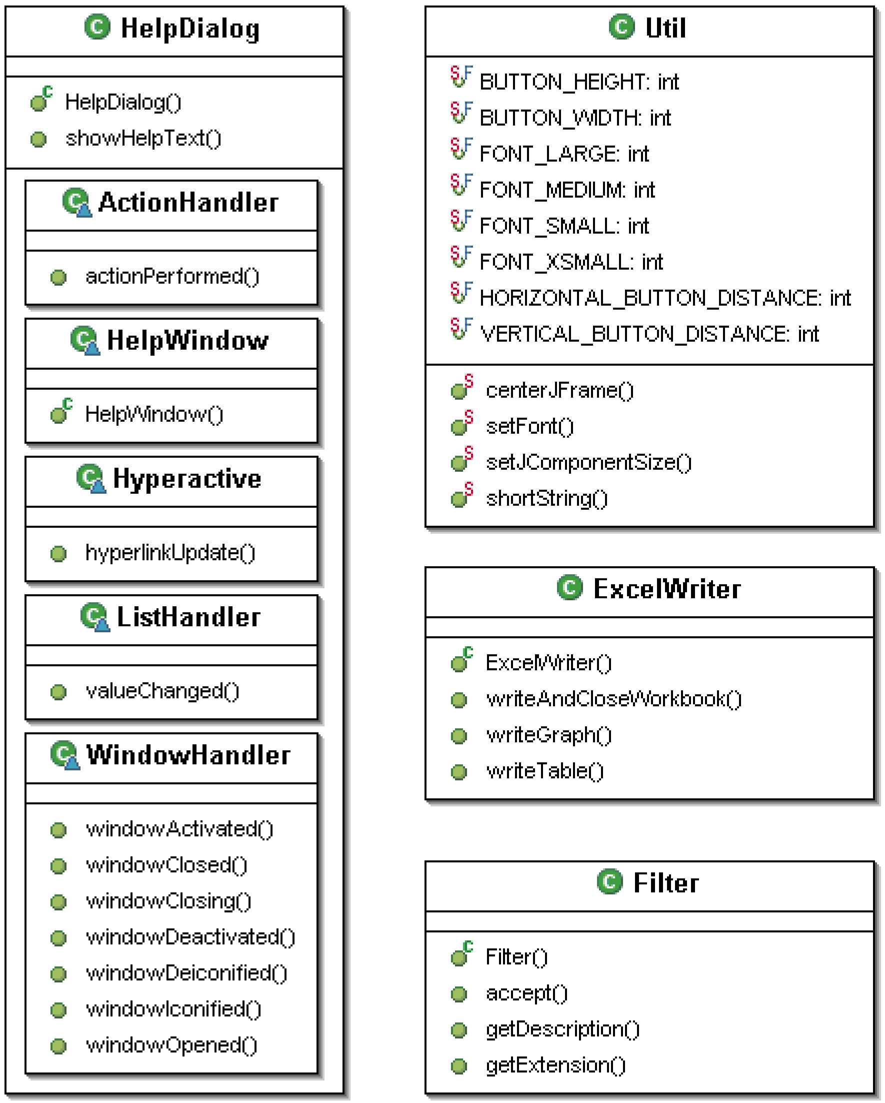
To handle parameter settings by the user, an internal class called PreferencesWindow is implemented.

DollarWindow.java
The functionality of the DollarWindow class is similar to the PairwiseWindow; the main task is to pass information to the corresponding class in the prioritizer package. In this case, the information is sent to the DollarTechnique class.

Besides the functionality mentioned, this class is also responsible for displaying and handling the table with a single editable column used for dollar value assignments.

The util package

In the util package, utility classes used throughout the program are placed.



Util.java
The Util class is a collection of utility methods used by window classes primarily. These methods handle layout issues needed to create a consistent look for all windows in the program. The methods set for example font sizes, fixed sizes for window component and locations of windows.

A method for dividing a long text string into a specified number of rows with a desired number of letters on each row is also located in this class.

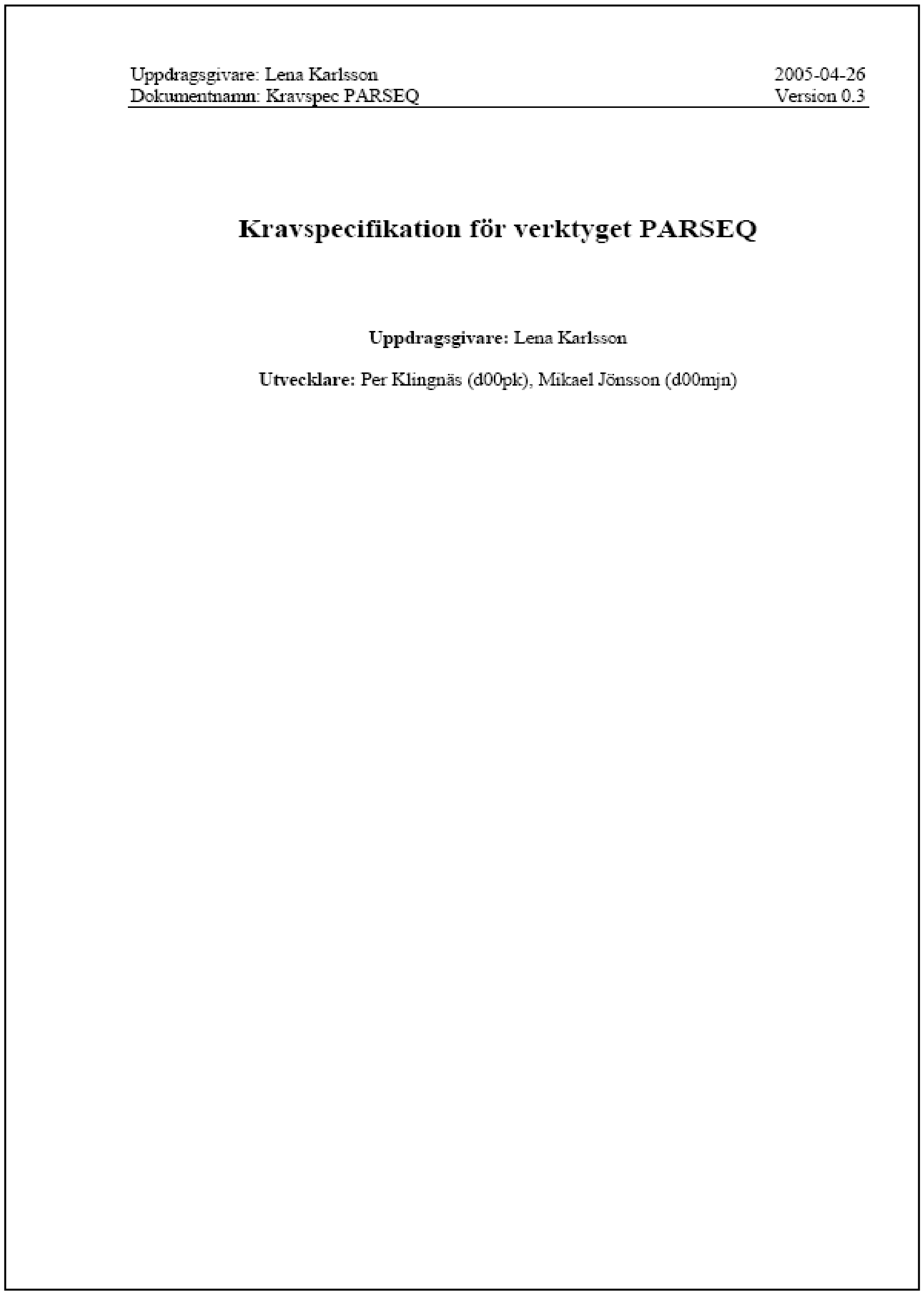
ExcelWriter.java
When the result from the root cause matrix, together with or without the graph image, shall be exported to an Excel spreadsheet this class is used. This class uses, in analogy with the requirement importing from Excel with the ExcelImporter, the JExcelApi library.

Filter.java
Since the program only can handle Excel files for importing and exporting, files with an xls extension, and png files for image saving, a file filter must be used. This class filters the contents of a directory to only display the selected file-types in open- and save-file dialogs.

HelpDialog.java
When the user of the program needs to consult the help files, the HelpDialog class is responsible for displaying the help window with the desired help file. The HelpDialog has an internal class called HelpWindow that represents the window where the help text is displayed. Help files are stored as html files and therefore functionality for reading and displaying these kinds of files is also implemented in this class.

Appendix C – Requirements specification

In this appendix we list the latest requirements specification as received by the customer. *Note: The entire specification is in Swedish.*

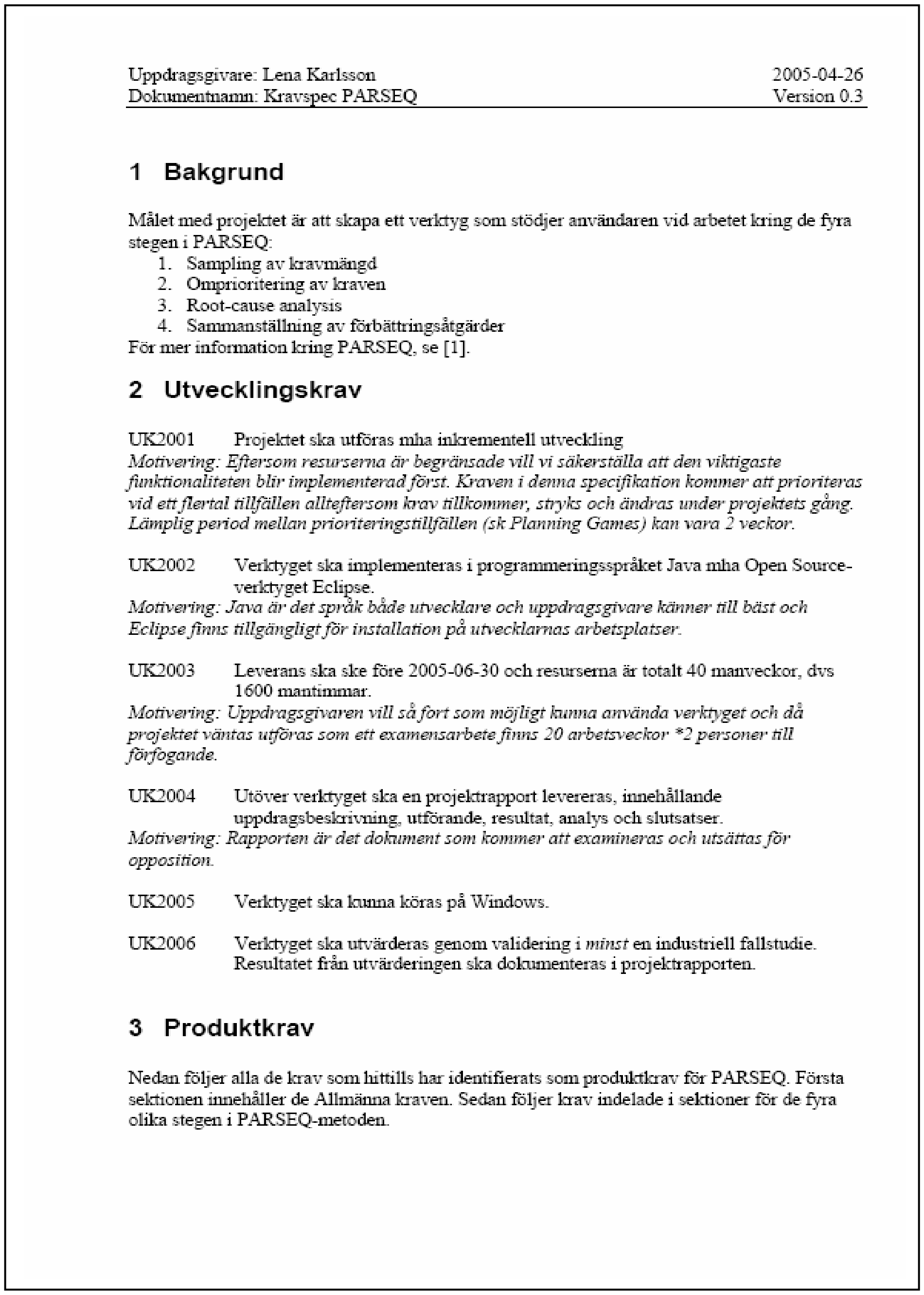


Uppdragsgivare: Lena Karlsson 2005-04-26
Dokumentnamn: Kravspec PARSEQ Version 0.3

Kravspecifikation för verktyget PARSEQ

Uppdragsgivare: Lena Karlsson

Utvecklare: Per Klingnäs (d00pk), Mikael Jönsson (d00mjn)



Uppdragsgivare: Lena Karlsson 2005-04-26
Dokumentnamn: Kravspec PARSEQ Version 0.3

1 Bakgrund

Målet med projektet är att skapa ett verktyg som stödjer användaren vid arbetet kring de fyra stegen i PARSEQ:

1. Sampling av kravmängd
2. Omprioritering av kraven
3. Root-cause analysis
4. Sammanställning av förbättringsåtgärder

För mer information kring PARSEQ, se [1].

2 Utvecklingskrav

UK2001 Projektet ska utföras mha inkrementell utveckling
Motivering: Eftersom resurserna är begränsade vill vi säkerställa att den viktigaste funktionaliteten blir implementerad först. Kraven i denna specifikation kommer att prioriteras vid ett flertal tillfällen allteftersom krav tillkommer, stryks och ändras under projektets gång. Lämplig period mellan prioriteringstillfällen (sk Planning Games) kan vara 2 veckor.

UK2002 Verktyget ska implementeras i programmeringsspråket Java mha Open Source-verktyget Eclipse.
Motivering: Java är det språk både utvecklare och uppdragsgivare känner till bäst och Eclipse finns tillgängligt för installation på utvecklarnas arbetsplatser.

UK2003 Leverans ska ske före 2005-06-30 och resurserna är totalt 40 manveckor, dvs 1600 mantimmar.
*Motivering: Uppdragsgivaren vill så fort som möjligt kunna använda verktyget och då projektet väntas utföras som ett examensarbete finns 20 arbetsveckor *2 personer till förfogande.*

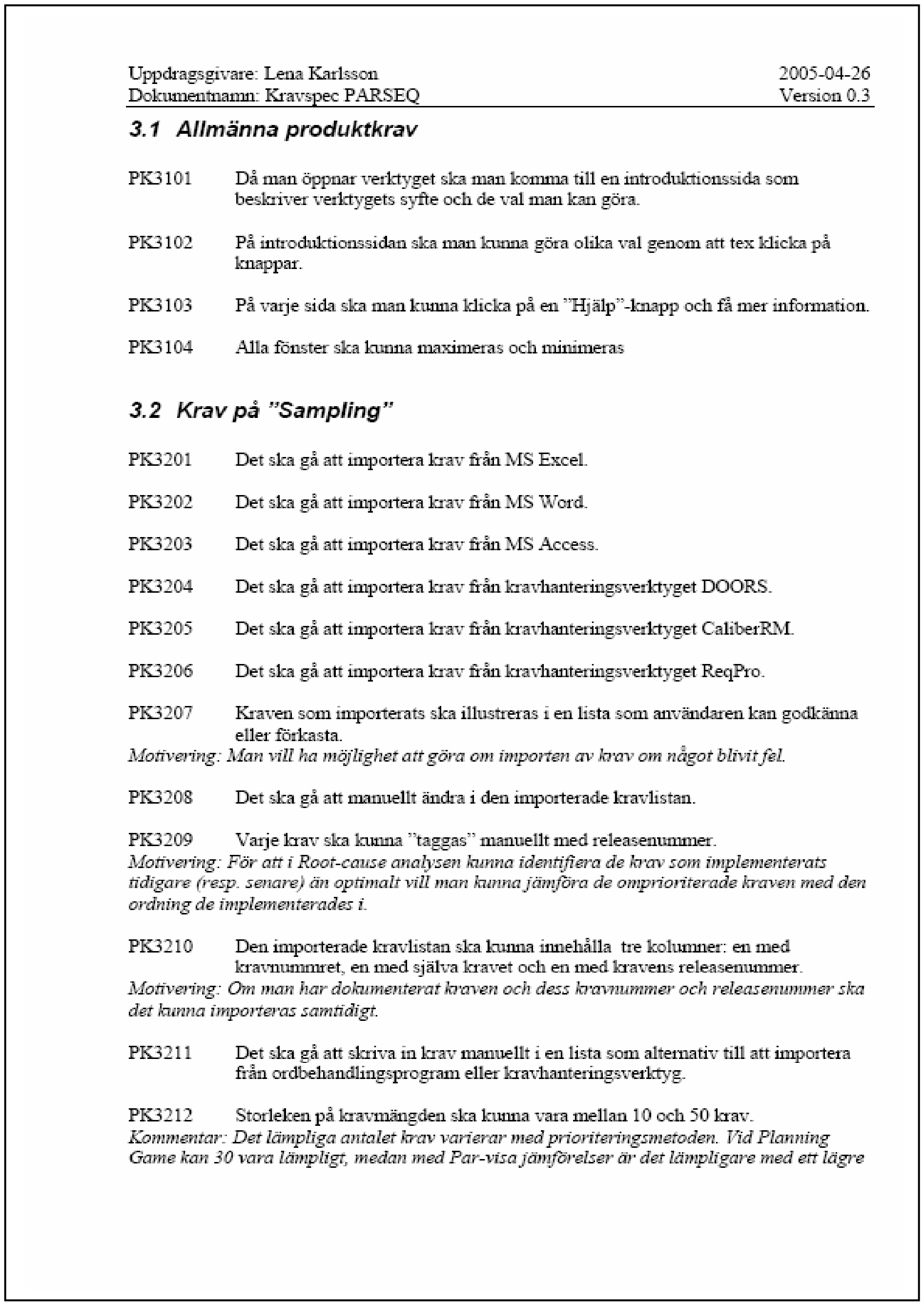
UK2004 Utöver verktyget ska en projektrapport levereras, innehållande uppdragsbeskrivning, utförande, resultat, analys och slutsatser.
Motivering: Rapporten är det dokument som kommer att examineras och utsättas för opposition.

UK2005 Verktyget ska kunna köras på Windows.

UK2006 Verktyget ska utvärderas genom validering i *minst* en industriell fallstudie. Resultatet från utvärderingen ska dokumenteras i projektrapporten.

3 Produktkrav

Nedan följer alla de krav som hittills har identifierats som produktkrav för PARSEQ. Första sektionen innehåller de Allmänna kraven. Sedan följer krav indelade i sektioner för de fyra olika stegen i PARSEQ-metoden.



Uppdragsgivare: Lena Karlsson 2005-04-26
Dokumentnamn: Kravspec PARSEQ Version 0.3

3.1 Allmänna produktkrav

PK3101 Då man öppnar verktyget ska man komma till en introduktionssida som beskriver verktygets syfte och de val man kan göra.

PK3102 På introduktionssidan ska man kunna göra olika val genom att tex klicka på knappar.

PK3103 På varje sida ska man kunna klicka på en "Hjälp"-knapp och få mer information.

PK3104 Alla fönster ska kunna maximeras och minimeras

3.2 Krav på "Sampling"

PK3201 Det ska gå att importera krav från MS Excel.

PK3202 Det ska gå att importera krav från MS Word.

PK3203 Det ska gå att importera krav från MS Access.

PK3204 Det ska gå att importera krav från kravhanteringsverktyget DOORS.

PK3205 Det ska gå att importera krav från kravhanteringsverktyget CaliberRM.

PK3206 Det ska gå att importera krav från kravhanteringsverktyget ReqPro.

PK3207 Kraven som importerats ska illustreras i en lista som användaren kan godkänna eller förkasta.
Motivering: Man vill ha möjlighet att göra om importen av krav om något blivit fel.

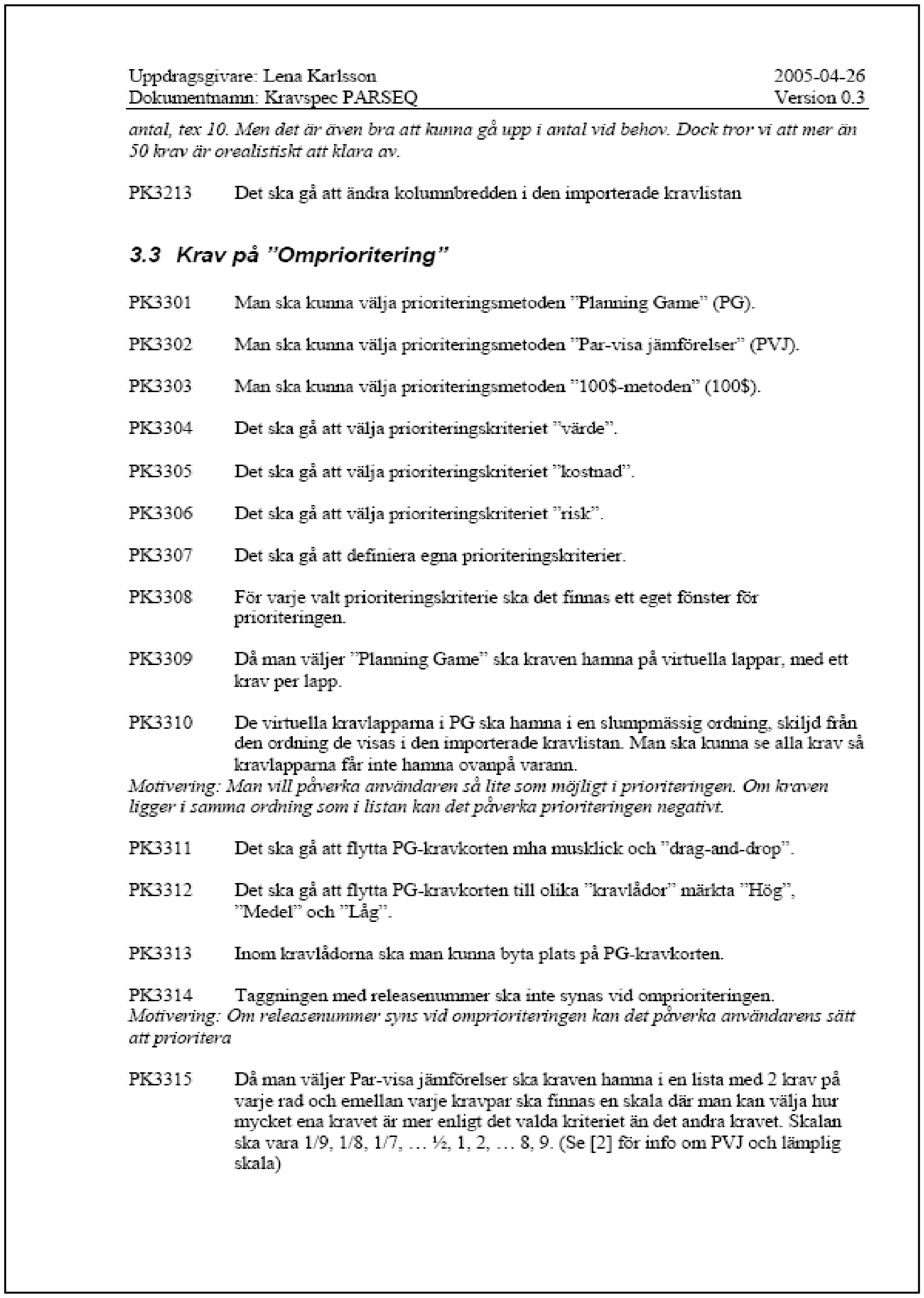
PK3208 Det ska gå att manuellt ändra i den importerade kravlistan.

PK3209 Varje krav ska kunna "taggas" manuellt med releasenummer.
Motivering: För att i Root-cause analysen kunna identifiera de krav som implementerats tidigare (resp. senare) än optimalt vill man kunna jämföra de omprioriterade kraven med den ordning de implementerades i.

PK3210 Den importerade kravlistan ska kunna innehålla tre kolumner: en med kravnummret, en med själva kravet och en med kravens releasenummer.
Motivering: Om man har dokumenterat kraven och dess kravnummer och releasenummer ska det kunna importeras samtidigt.

PK3211 Det ska gå att skriva in krav manuellt i en lista som alternativ till att importera från ordbehandlingsprogram eller kravhanteringsverktyg.

PK3212 Storleken på kravmängden ska kunna vara mellan 10 och 50 krav.
Kommentar: Det lämpliga antalet krav varierar med prioriteringsmetoden. Vid Planning Game kan 30 vara lämpligt, medan med Par-visa jämförelser är det lämpligare med ett lägre



Uppdragsgivare: Lena Karlsson 2005-04-26
Dokumentnamn: Kravspec PARSEQ Version 0.3

antal, tex 10. Men det är även bra att kunna gå upp i antal vid behov. Dock tror vi att mer än 50 krav är orealistiskt att klara av.

PK3213 Det ska gå att ändra kolumnbredden i den importerade kravlistan

3.3 Krav på "Omprioritering"

PK3301 Man ska kunna välja prioriteringsmetoden "Planning Game" (PG).

PK3302 Man ska kunna välja prioriteringsmetoden "Par-visa jämförelser" (PVJ).

PK3303 Man ska kunna välja prioriteringsmetoden "100$-metoden" (100$).

PK3304 Det ska gå att välja prioriteringskriteriet "värde".

PK3305 Det ska gå att välja prioriteringskriteriet "kostnad".

PK3306 Det ska gå att välja prioriteringskriteriet "risk".

PK3307 Det ska gå att definiera egna prioriteringskriterier.

PK3308 För varje valt prioriteringskriterie ska det finnas ett eget fönster för prioriteringen.

PK3309 Då man väljer "Planning Game" ska kraven hamna på virtuella lappar, med ett krav per lapp.

PK3310 De virtuella kravlapparna i PG ska hamna i en slumpmässig ordning, skiljd från den ordning de visas i den importerade kravlistan. Man ska kunna se alla krav så kravlapparna får inte hamna ovanpå varann.
Motivering: Man vill påverka användaren så lite som möjligt i prioriteringen. Om kraven ligger i samma ordning som i listan kan det påverka prioriteringen negativt.

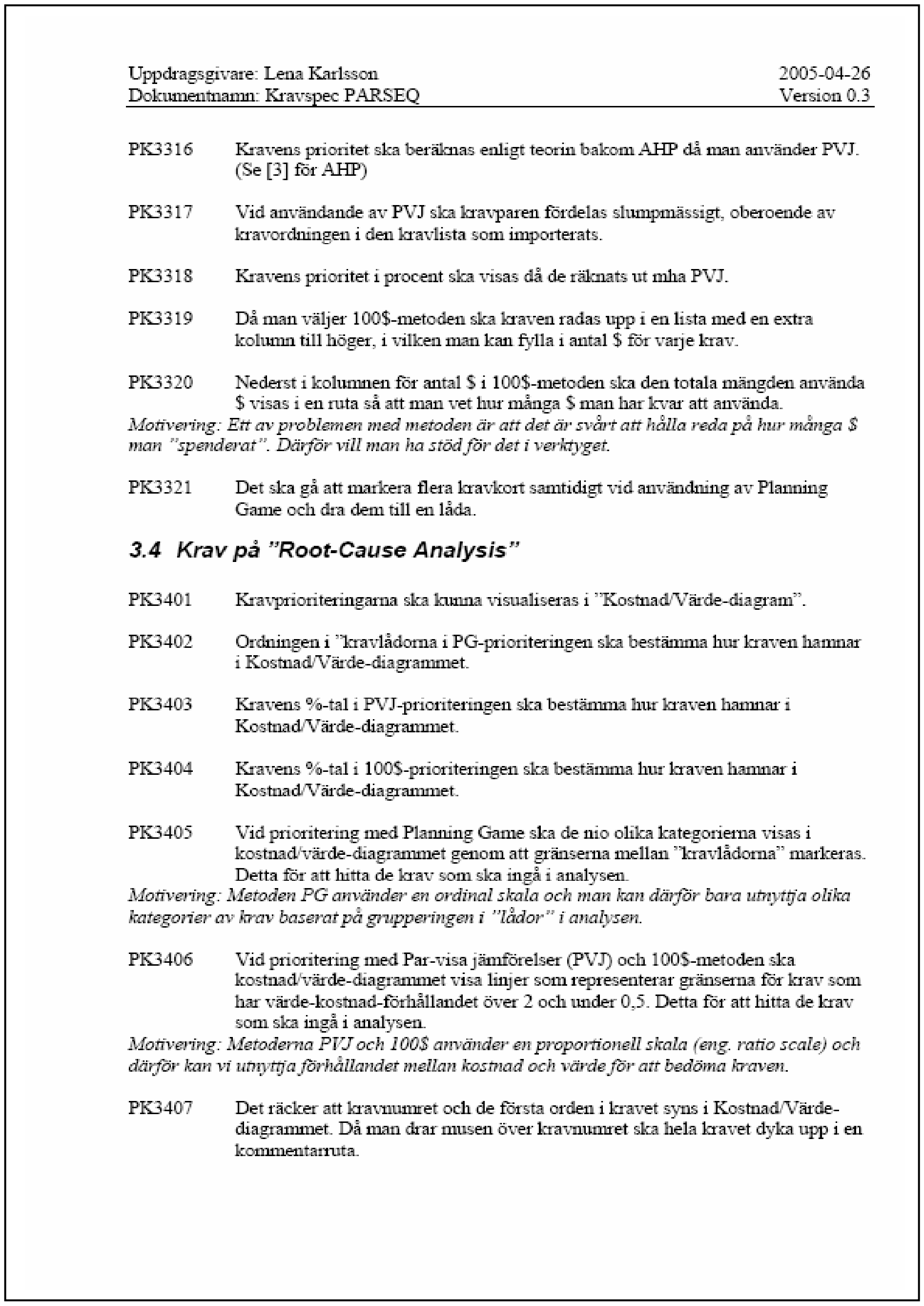
PK3311 Det ska gå att flytta PG-kravkorten mha musklick och "drag-and-drop".

PK3312 Det ska gå att flytta PG-kravkorten till olika "kravlådor" märkta "Hög", "Medel" och "Låg".

PK3313 Inom kravlådorna ska man kunna byta plats på PG-kravkorten.

PK3314 Taggningen med releasenummer ska inte synas vid omprioriteringen.
Motivering: Om releasenummer syns vid omprioriteringen kan det påverka användarens sätt att prioritera

PK3315 Då man väljer Par-visa jämförelser ska kraven hamna i en lista med 2 krav på varje rad och emellan varje kravpar ska finnas en skala där man kan välja hur mycket ena kravet är mer enligt det valda kriteriet än det andra kravet. Skalan ska vara 1/9, 1/8, 1/7, … ½, 1, 2, … 8, 9. (Se [2] för info om PVJ och lämplig skala)



PK3316 Kravens prioritet ska beräknas enligt teorin bakom AHP då man använder PVJ. (Se [3] för AHP)

PK3317 Vid användande av PVJ ska kravparen fördelas slumpmässigt, oberoende av kravordningen i den kravlista som importerats.

PK3318 Kravens prioritet i procent ska visas då de räknats ut mha PVJ.

PK3319 Då man väljer 100$-metoden ska kraven radas upp i en lista med en extra kolumn till höger, i vilken man kan fylla i antal $ för varje krav.

PK3320 Nederst i kolumnen för antal $ i 100$-metoden ska den totala mängden använda $ visas i en ruta så att man vet hur många $ man har kvar att använda.
Motivering: Ett av problemen med metoden är att det är svårt att hålla reda på hur många $ man "spenderat". Därför vill man ha stöd för det i verktyget.

PK3321 Det ska gå att markera flera kravkort samtidigt vid användning av Planning Game och dra dem till en låda.

3.4 Krav på "Root-Cause Analysis"

PK3401 Kravprioriteringarna ska kunna visualiseras i "Kostnad/Värde-diagram".

PK3402 Ordningen i "kravlådorna i PG-prioriteringen ska bestämma hur kraven hamnar i Kostnad/Värde-diagrammet.

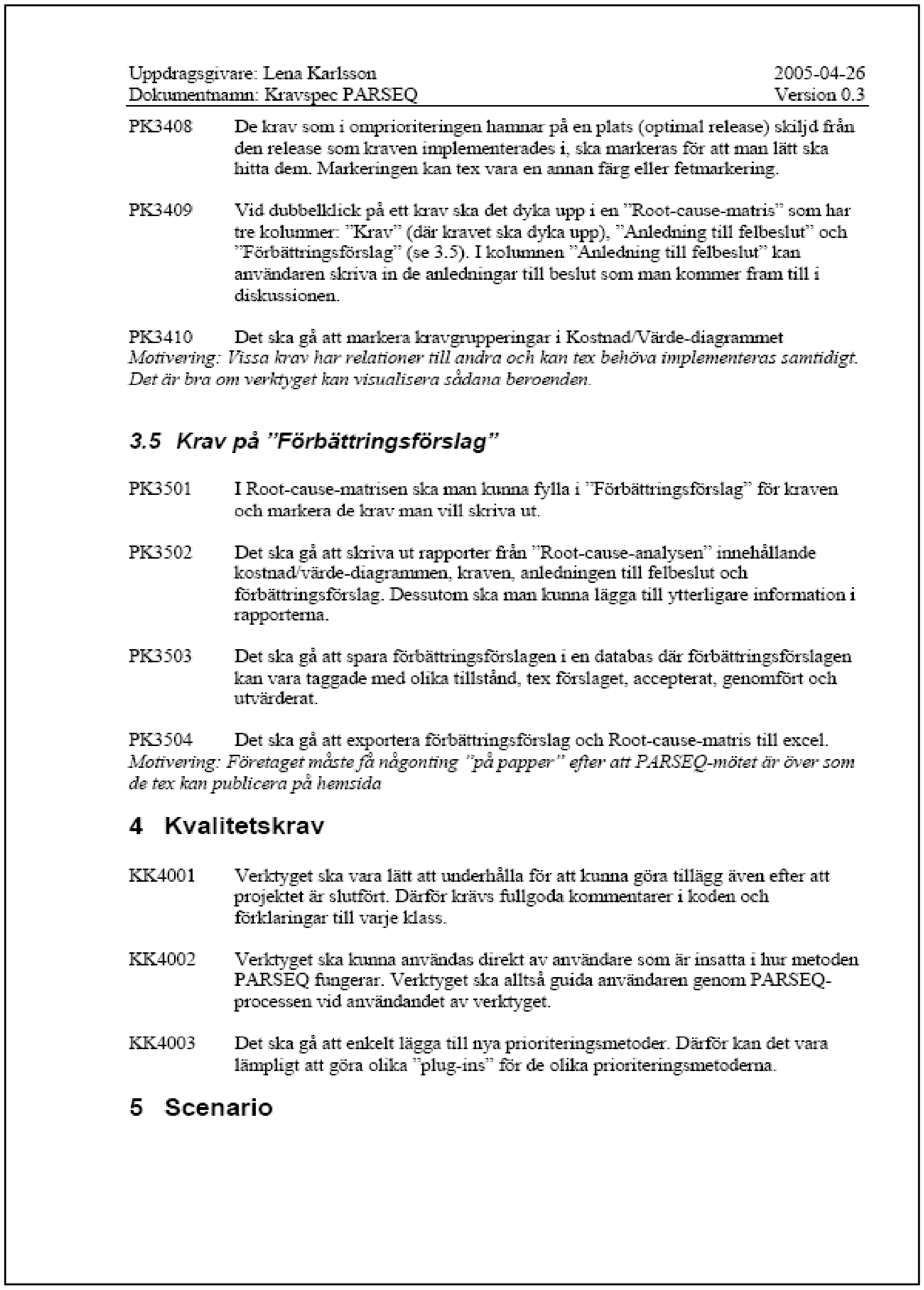
PK3403 Kravens %-tal i PVJ-prioriteringen ska bestämma hur kraven hamnar i Kostnad/Värde-diagrammet.

PK3404 Kravens %-tal i 100$-prioriteringen ska bestämma hur kraven hamnar i Kostnad/Värde-diagrammet.

PK3405 Vid prioritering med Planning Game ska de nio olika kategorierna visas i kostnad/värde-diagrammet genom att gränserna mellan "kravlådorna" markeras. Detta för att hitta de krav som ska ingå i analysen.
Motivering: Metoden PG använder en ordinal skala och man kan därför bara utnyttja olika kategorier av krav baserat på grupperingen i "lådor" i analysen.

PK3406 Vid prioritering med Par-visa jämförelser (PVJ) och 100$-metoden ska kostnad/värde-diagrammet visa linjer som representerar gränserna för krav som har värde-kostnad-förhållandet över 2 och under 0,5. Detta för att hitta de krav som ska ingå i analysen.
Motivering: Metoderna PVJ och 100$ använder en proportionell skala (eng. ratio scale) och därför kan vi utnyttja förhållandet mellan kostnad och värde för att bedöma kraven.

PK3407 Det räcker att kravnumret och de första orden i kravet syns i Kostnad/Värde-diagrammet. Då man drar musen över kravnumret ska hela kravet dyka upp i en kommentarruta.



Uppdragsgivare: Lena Karlsson 2005-04-26
Dokumentnamn: Kravspec PARSEQ Version 0.3

PK3408 De krav som i omprioriteringen hamnar på en plats (optimal release) skiljd från den release som kraven implementerades i, ska markeras för att man lätt ska hitta dem. Markeringen kan tex vara en annan färg eller fetmarkering.

PK3409 Vid dubbelklick på ett krav ska det dyka upp i en "Root-cause-matris" som har tre kolumner: "Krav" (där kravet ska dyka upp), "Anledning till felbeslut" och "Förbättringsförslag" (se 3.5). I kolumnen "Anledning till felbeslut" kan användaren skriva in de anledningar till beslut som man kommer fram till i diskussionen.

PK3410 Det ska gå att markera kravgrupperingar i Kostnad/Värde-diagrammet
Motivering: Vissa krav har relationer till andra och kan tex behöva implementeras samtidigt. Det är bra om verktyget kan visualisera sådana beroenden.

3.5 Krav på "Förbättringsförslag"

PK3501 I Root-cause-matrisen ska man kunna fylla i "Förbättringsförslag" för kraven och markera de krav man vill skriva ut.

PK3502 Det ska gå att skriva ut rapporter från "Root-cause-analysen" innehållande kostnad/värde-diagrammen, kraven, anledningen till felbeslut och förbättringsförslag. Dessutom ska man kunna lägga till ytterligare information i rapporterna.

PK3503 Det ska gå att spara förbättringsförslagen i en databas där förbättringsförslagen kan vara taggade med olika tillstånd, tex förslaget, accepterat, genomfört och utvärderat.

PK3504 Det ska gå att exportera förbättringsförslag och Root-cause-matris till excel.
Motivering: Företaget måste få någonting "på papper" efter att PARSEQ-mötet är över som de tex kan publicera på hemsida

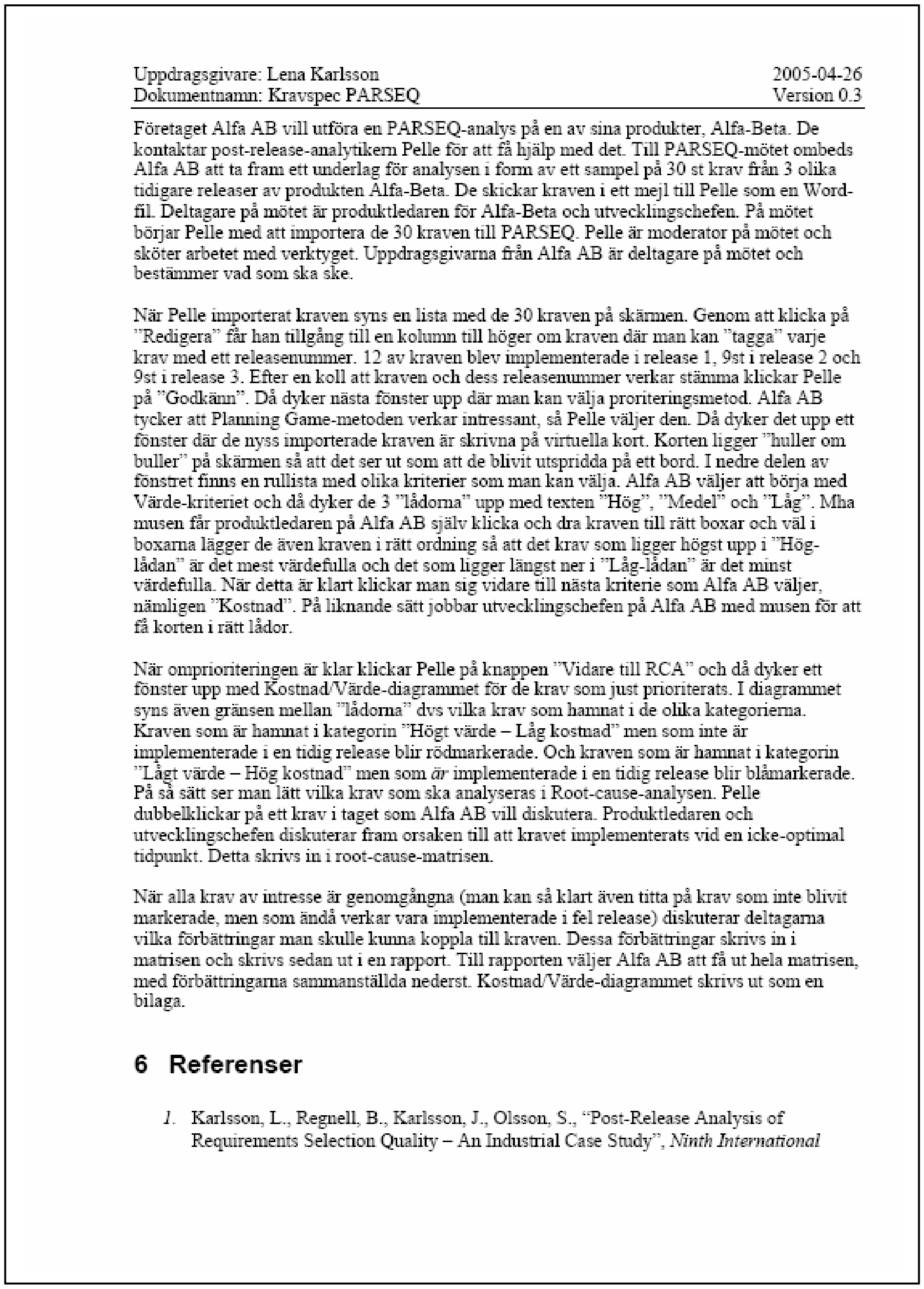
4 Kvalitetskrav

KK4001 Verktyget ska vara lätt att underhålla för att kunna göra tillägg även efter att projektet är slutfört. Därför krävs fullgoda kommentarer i koden och förklaringar till varje klass.

KK4002 Verktyget ska kunna användas direkt av användare som är insatta i hur metoden PARSEQ fungerar. Verktyget ska alltså guida användaren genom PARSEQ-processen vid användandet av verktyget.

KK4003 Det ska gå att enkelt lägga till nya prioriteringsmetoder. Därför kan det vara lämpligt att göra olika "plug-ins" för de olika prioriteringsmetoderna.

5 Scenario



Uppdragsgivare: Lena Karlsson 2005-04-26
Dokumentnamn: Kravspec PARSEQ Version 0.3

Företaget Alfa AB vill utföra en PARSEQ-analys på en av sina produkter, Alfa-Beta. De kontaktar post-release-analytikern Pelle för att få hjälp med det. Till PARSEQ-mötet ombeds Alfa AB att ta fram ett underlag för analysen i form av ett sampel på 30 st krav från 3 olika tidigare releaser av produkten Alfa-Beta. De skickar kraven i ett mejl till Pelle som en Word-fil. Deltagare på mötet är produktledaren för Alfa-Beta och utvecklingschefen. På mötet börjar Pelle med att importera de 30 kraven till PARSEQ. Pelle är moderator på mötet och sköter arbetet med verktyget. Uppdragsgivarna från Alfa AB är deltagare på mötet och bestämmer vad som ska ske.

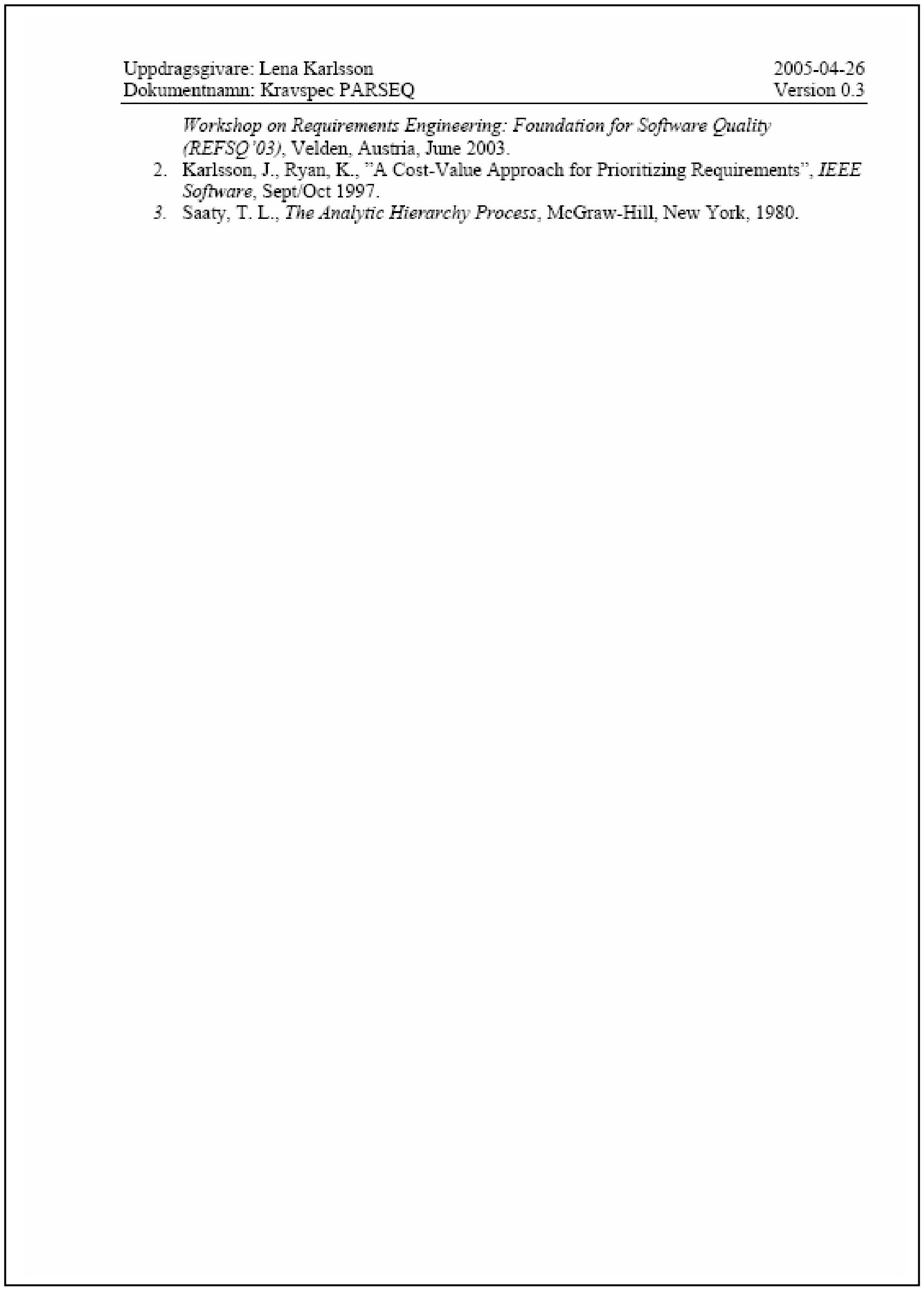
När Pelle importerat kraven syns en lista med de 30 kraven på skärmen. Genom att klicka på ”Redigera” får han tillgång till en kolumn till höger om kraven där man kan ”tagga” varje krav med ett releasenummer. 12 av kraven blev implementerade i release 1, 9st i release 2 och 9st i release 3. Efter en koll att kraven och dess releasenummer verkar stämma klickar Pelle på ”Godkänn”. Då dyker nästa fönster upp där man kan välja proriteringsmetod. Alfa AB tycker att Planning Game-metoden verkar intressant, så Pelle väljer den. Då dyker det upp ett fönster där de nyss importerade kraven är skrivna på virtuella kort. Korten ligger ”huller om buller” på skärmen så att det ser ut som att de blivit utspridda på ett bord. I nedre delen av fönstret finns en rullista med olika kriterier som man kan välja. Alfa AB väljer att börja med Värde-kriteriet och då dyker de 3 ”lådorna” upp med texten ”Hög”, ”Medel” och ”Låg”. Mha musen får produktledaren på Alfa AB själv klicka och dra kraven till rätt boxar och väl i boxarna lägger de även kraven i rätt ordning så att det krav som ligger högst upp i ”Hög-lådan” är det mest värdefulla och det som ligger längst ner i ”Låg-lådan” är det minst värdefulla. När detta är klart klickar man sig vidare till nästa kriterie som Alfa AB väljer, nämligen ”Kostnad”. På liknande sätt jobbar utvecklingschefen på Alfa AB med musen för att få korten i rätt lådor.

När omprioriteringen är klar klickar Pelle på knappen ”Vidare till RCA” och då dyker ett fönster upp med Kostnad/Värde-diagrammet för de krav som just prioriterats. I diagrammet syns även gränsen mellan ”lådorna” dvs vilka krav som hamnat i de olika kategorierna. Kraven som är hamnat i kategorin ”Högt värde – Låg kostnad” men som inte är implementerade i en tidig release blir rödmarkerade. Och kraven som är hamnat i kategorin ”Lågt värde – Hög kostnad” men som *är* implementerade i en tidig release blir blåmarkerade. På så sätt ser man lätt vilka krav som ska analyseras i Root-cause-analysen. Pelle dubbelklickar på ett krav i taget som Alfa AB vill diskutera. Produktledaren och utvecklingschefen diskuterar fram orsaken till att kravet implementerats vid en icke-optimal tidpunkt. Detta skrivs in i root-cause-matrisen.

När alla krav av intresse är genomgångna (man kan så klart även titta på krav som inte blivit markerade, men som ändå verkar vara implementerade i fel release) diskuterar deltagarna vilka förbättringar man skulle kunna koppla till kraven. Dessa förbättringar skrivs in i matrisen och skrivs sedan ut i en rapport. Till rapporten väljer Alfa AB att få ut hela matrisen, med förbättringarna sammanställda nederst. Kostnad/Värde-diagrammet skrivs ut som en bilaga.

6 Referenser

1. Karlsson, L., Regnell, B., Karlsson, J., Olsson, S., “Post-Release Analysis of Requirements Selection Quality – An Industrial Case Study”, *Ninth International*

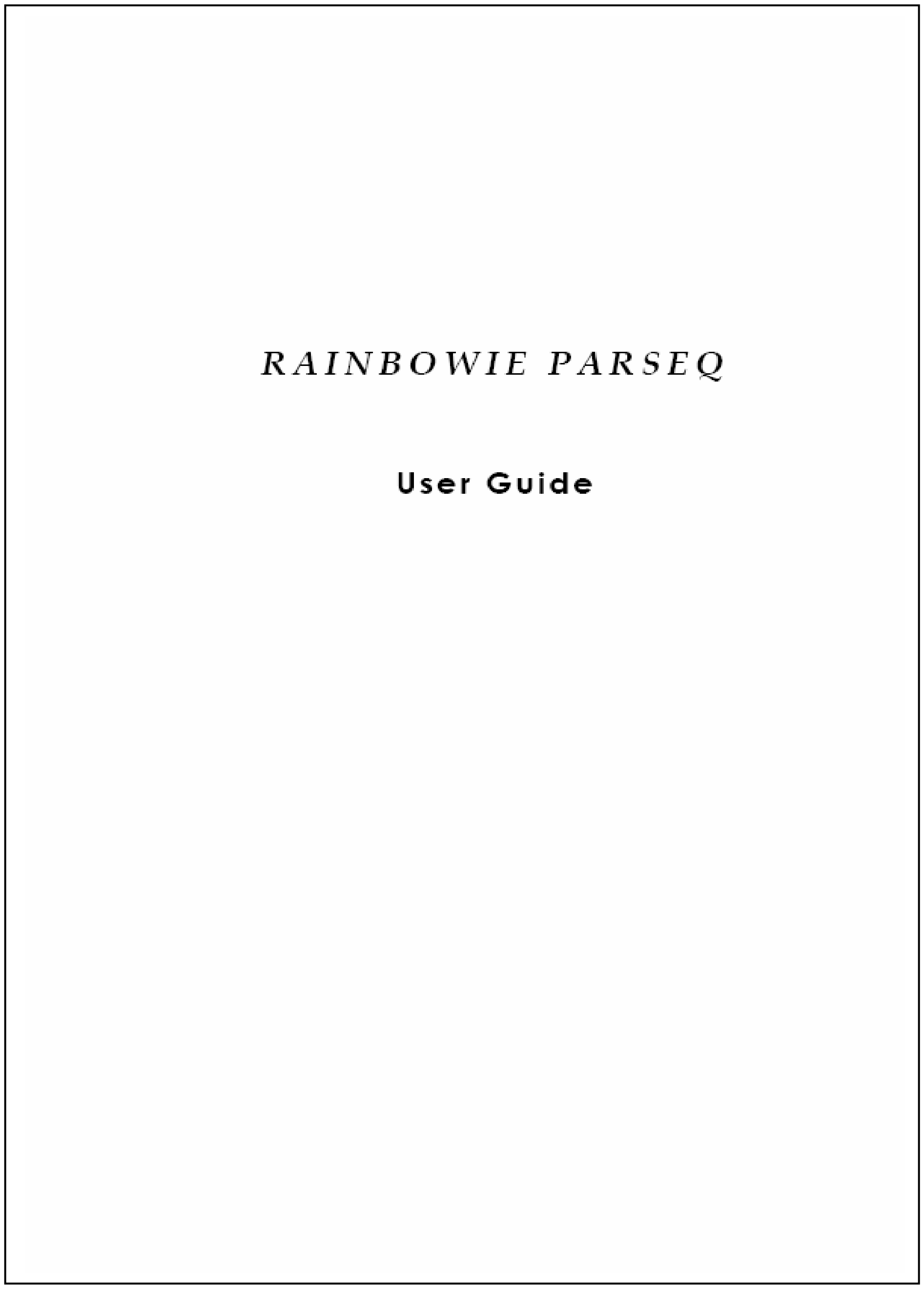


Uppdragsgivare: Lena Karlsson 2005-04-26
Dokumentnamn: Kravspec PARSEQ Version 0.3

Workshop on Requirements Engineering: Foundation for Software Quality (REFSQ'03), Velden, Austria, June 2003.
2. Karlsson, J., Ryan, K., "A Cost-Value Approach for Prioritizing Requirements", *IEEE Software*, Sept/Oct 1997.
3. Saaty, T. L., *The Analytic Hierarchy Process*, McGraw-Hill, New York, 1980.

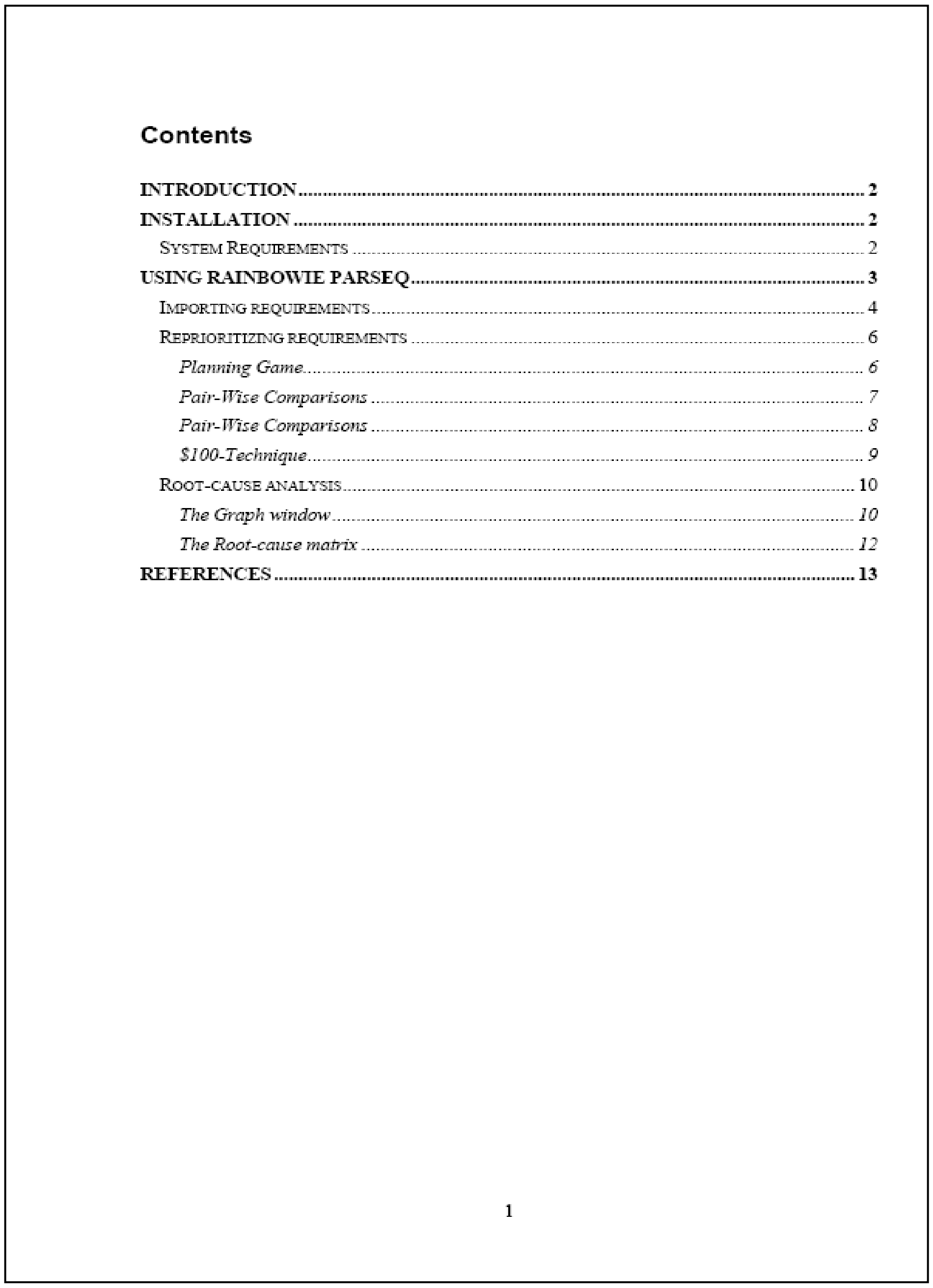
Appendix D – User guide

In addition to this report a user guide for Rainbowie was written.



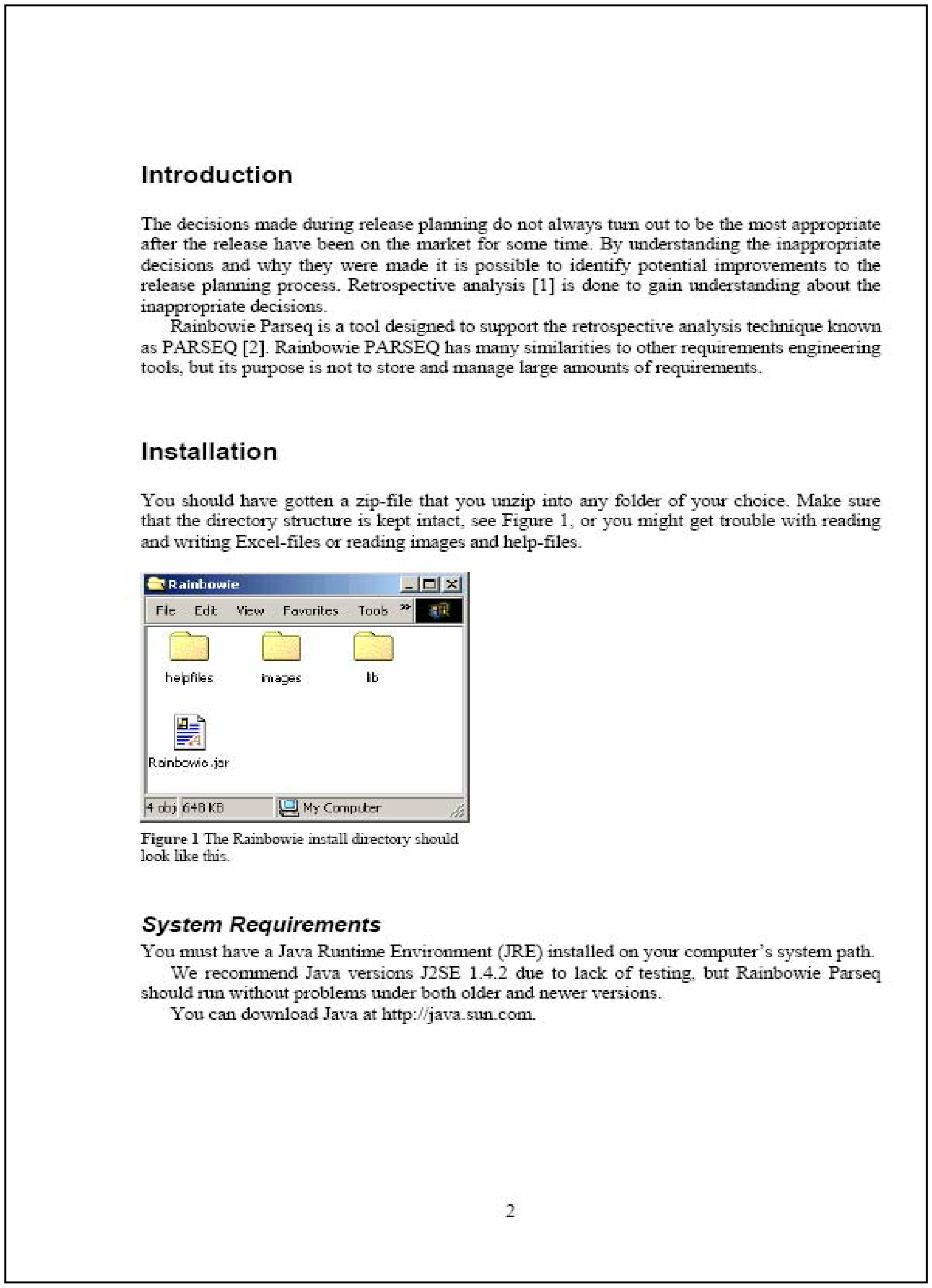
RAINBOWIE PARSEQ

User Guide



Contents

1



Introduction

The decisions made during release planning do not always turn out to be the most appropriate after the release have been on the market for some time. By understanding the inappropriate decisions and why they were made it is possible to identify potential improvements to the release planning process. Retrospective analysis [1] is done to gain understanding about the inappropriate decisions.

Rainbowie Parseq is a tool designed to support the retrospective analysis technique known as PARSEQ [2]. Rainbowie PARSEQ has many similarities to other requirements engineering tools, but its purpose is not to store and manage large amounts of requirements.

Installation

You should have gotten a zip-file that you unzip into any folder of your choice. Make sure that the directory structure is kept intact, see Figure 1, or you might get trouble with reading and writing Excel-files or reading images and help-files.

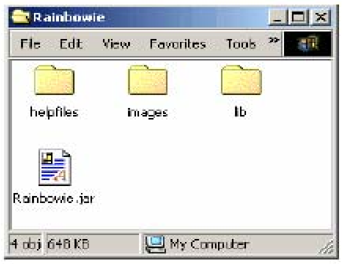


Figure 1 The Rainbowie install directory should look like this.

System Requirements

You must have a Java Runtime Environment (JRE) installed on your computer's system path.

We recommend Java versions J2SE 1.4.2 due to lack of testing, but Rainbowie Parseq should run without problems under both older and newer versions.

You can download Java at http://java.sun.com.

2

Using Rainbowie Parseq

Start Rainbowie Parseq by,

- In Windows or similar: double-clicking the Rainbowie icon in the folder that Rainbowie was installed
- In Unix, Dos or similar: typing `java -jar Rainbowie.jar` in the directory that Rainbowie was installed

Rainbowie Parseq is now running and the main window, shown in Figure 2, is displayed.

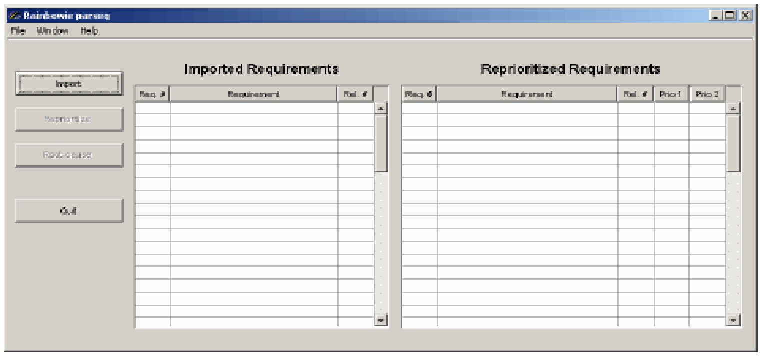


Figure 2 The main window of Rainbowie Parseq.

It is from this window that you choose when to import, reprioritize and do a root-cause analysis of the requirements, given that the previous step has been completed.

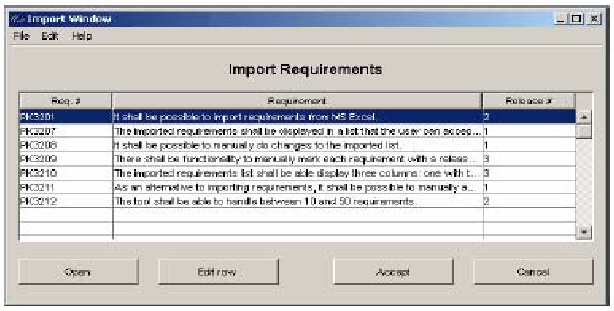
Feed-back about how far you have come in the process is given by buttons that are enabled and the tables that are filled once the appropriate step has been completed.

The menu bar

The menu bar is present in almost every window; there you can get help on how to use the program at any given time. Most of the functionality available through the buttons is also available through the menu. Therefore, you will in this guide, only find information about the menu when it contains functionality that is not covered by the buttons.

3

Importing requirements



Req. #	Requirement	Release #
PK3201	It shall be possible to import requirements from MS Excel.	2
PK3207	The imported requirements shall be displayed in a list that the user can accep...	1
PK3208	It shall be possible to manually do changes to the imported list.	1
PK3209	There shall be functionality to manually mark each requirement with a releas...	3
PK3210	The imported requirements list shall be able display three columns: one with t...	3
PK3211	As an alternative to importing requirements, it shall be possible to manually a...	1
PK3212	The tool shall be able to handle between 10 and 50 requirements.	2

Figure 3 The Import Window where the requirements to process are entered.

There are two ways to enter requirements into the program; both are done through the Import Window, see Figure 3, that you open by pressing the *Import*-button in the main window. Either you enter the requirements you want manually or you import the requirements from an Excel sheet. If you wish to reset and clear the table, go to the *Edit*-menu and choose *Clear table*.

Only the requirements' numbers are necessary, the numbers must be unique and there must be at least two requirements before you can continue.

When the table contains all the requirements you want to import, you press the *Accept*-button. This will return you to the main window, where the left table now contains the imported requirements.

Enter requirements manually
By double-clicking the row in the table where you want the requirement to be entered will let you enter a requirement manually. You can also edit a row by selecting it and pressing the Edit Row-button or by pressing the Enter-key. This will open the Edit Window, shown in Figure 4, where you enter the information you want. Then, you press the Accept-button in the Edit Window and the information you entered will be added to the table in the Import Window.
Only the last, Release #, column can be edited directly in the Import Window. This is to prevent users from changing a requirement by mistake.

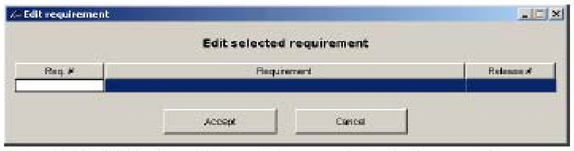
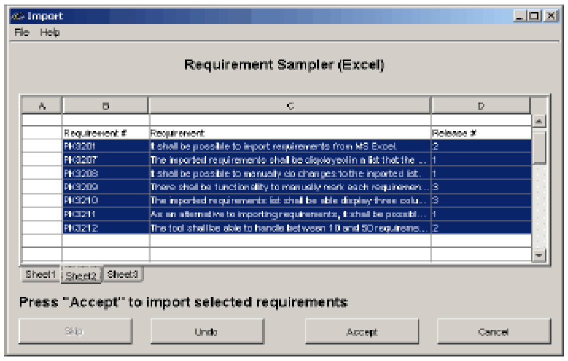


Figure 4 The Edit Window where requirements can be edited or entered.

4

Import requirements from Excel
The Excel-sheet you want to import from must be formatted in the correct way. That is, the requirements' numbers, descriptions and release numbers must each be in their own column and there must only be one requirement on each row. It is, however, only necessary to have and import the requirements' numbers. The requirements' descriptions and release numbers are optional.

To import from an Excel-sheet, press the *Open*-button in Import Window. This will open a file chooser dialog, where you select the Excel-file to open. The opened Excel-file will be read into the Excel importer window, Figure 5, this is a read-only window that will not and cannot edit the Excel-file.



A	B	C	D
	Requirement #	Requirement	Release #
	RK3201	It shall be possible to import requirements from MS Excel	2
	RK3207	The imported requirements shall be displayed in a list that the ...	1
	RK3208	It shall be possible to manually do changes to the imported list.	1
	RK3209	There shall be functionality to manually mark each requiremen...	3
	RK3210	The imported requirements list shall be able display three colu...	3
	RK3211	As an alternative to importing requirements, it shall be possibl...	1
	RK3212	The tool shall be able to handle between 10 and 50 requireme...	2

Press "Accept" to import selected requirements

Figure 5 Excel importer window.

In the Excel importer window, follow these steps:

1. Select the sheet with the requirements you want to import
2. Double-click the cell containing the first requirement to import
3. Double-click the cell containing the last requirement to import. This will select all the requirements between the first and last requirement.
4. Double-click anywhere in the column containing the requirements' descriptions to select them.
5. Double-click anywhere in the column containing the requirements' release numbers to select them.
6. Press the *Accept*-button

Step 4 and 5 are optional. You can skip step 4 by pressing the *Skip*-button. You do not have to press the *Skip*-button if you want to skip step 5 or both step 4 and 5, pressing *Accept* will automatically skip the remaining steps and close the window.

By pressing the *Accept*-button, you are returned to the Import Window.

5

Reprioritizing requirements

When you have imported the requirements, it is time to reprioritize them. Pressing the *Reprioritize*-button in the main window will open a dialog, Figure 6, where you choose the appropriate method and criteria to prioritize by.

There are three predefined, commonly used, criteria: Value, Cost and Risk. There are also two text fields if you wish to define your own criteria.

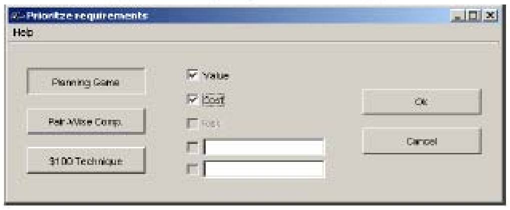
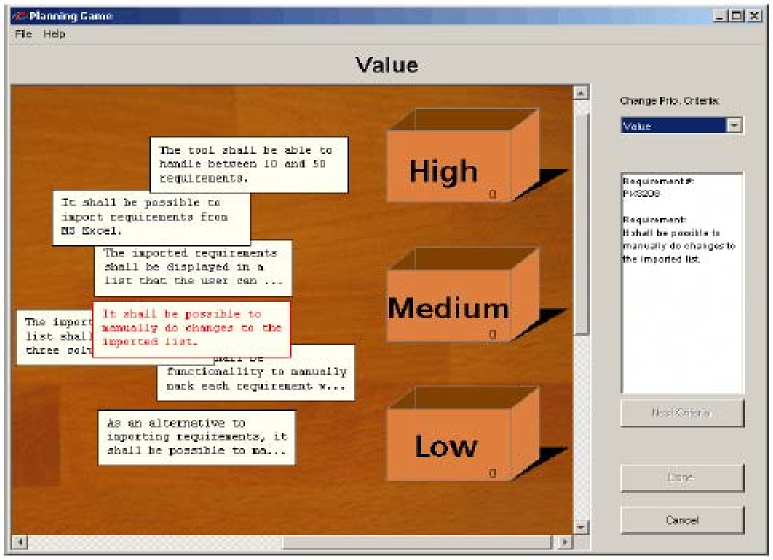


Figure 6 The window where prioritizing method and criteria are selected.

Planning Game

In the Planning Game the cards on the desk are sorted into the three boxes, where each box corresponds to a level of importance to the criteria. Within each box the cards are sorted by the user, where the card most relevant to the criteria is put on the top.



The tool shall be able to handle between 10 and 50 requirements.

It shall be possible to import requirements from MS Excel.

The imported requirements shall be displayed in a list that the user can ...

It shall be possible to manually do changes to the imported list.

functionallity to manually mark each requirement w...

As an alternative to importing requirements, it shall be possible to ma...

Requirement#:
PKS208

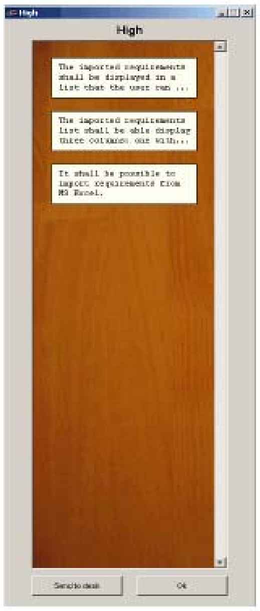
Requirement:
It shall be possible to manually do changes to the imported list.

Figure 7 The Planning Game window.

6

First choose which criteria to start prioritizing after by using the combobox in the upper right corner. This can only be done before the first card has been put in a box.

Then, for each card, click & drag it to one of the boxes. When a box is highlighted you can drop the card in that box. More than one card can be selected by dragging a selection square or by holding down the Ctrl-key while clicking on the cards you want to select. The selected cards can then be moved and dropped in a box together. *Dragging and dropping multiple cards do not guarantee the order the cards will have in the box.*



The imported requirements shall be displayed in a list that the user can ...

The imported requirements list shall be able display three columns: one with...

It shall be possible to import requirements from MS Excel.

Figure 8 Window representing a box

To sort the cards in a box, open it by double-clicking on that box, then click & drag the cards. If you notice that a card is in the wrong box, you can highlight that card and press the *Send to desk*-button, the card sent to the desk can now be put in another box. This is the only way to move the cards between the boxes or between a box and the desk.

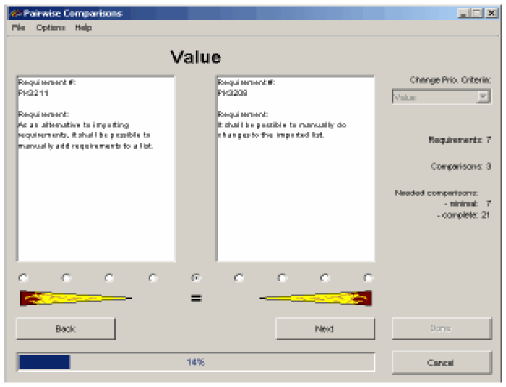
When you are finished with the prioritization for the first criteria, press the *Next Criteria*-button and a window will appear with an overview of the prioritization. In this window you can choose if you want to confirm the prioritization, you can also choose to assign relative values to the requirements.

If you press *Assign values*, a window for assigning relative values using a modified $100-technique will be shown. If you choose to assign, or not to assign, values for the first criteria, you must do the same for the last criteria as well. The same procedure is then used when you are finished with the prioritization for the last criteria.

7

Pair-Wise Comparisons

In the Pair-wise Comparisons technique the priorities are decided by comparing the requirements two and two. To get the correct result, all possible pairs should be compared. However, this means that in practice there can be a huge number of needed comparisons. Therefore, approximate priorities can be calculated using an incomplete pair-wise comparisons technique, this is what is done in this program if you choose to stop before all possible pairs have been compared. Although, you are encouraged to keep in mind that this may result in a less accurate priority list.



Requirement #:
PH3211

Requirement:
As an alternative to importing requirements, it shall be possible to manually add requirements to a list.

Requirement #:
PH3209

Requirement:
It shall be possible to manually do changes to the imported list.

Requirements: 7

Comparisons: 3

Needed comparisons:
- minimal: 7
- complete: 21

Figure 9 The Pair-Wise Comparisons window.

When the window is open, the current comparison is shown in the text fields and their relative importance to each other is set using the radio buttons below.

First choose which criteria to start prioritizing after by using the combobox in the upper right corner. This can only be done before the first comparison have been done.

To set the relative significance, either double-click a radio button or select a radio button and press the *Next*-button. This will bring up the next pair to compare. The more significant one requirement is, than the other, a radio button further to that side should be selected. If you hold the mouse over a radio button, an explanation will be shown.

You can go back to look at and change previously compared pairs by pressing the *Back*-button.

Through the progress bar at the bottom of the window and the information to the right in the window, you can follow your progress and see how many pairs you have left, throughout the prioritization.

You must complete at least the same number of comparisons as the amount of requirements to be able to get any result at all.

Under the *Options*-menu, you can choose whether or not to show radio buttons for intermediate values. These intermediate values are used when a compromise between two adjacent judgments must be done.

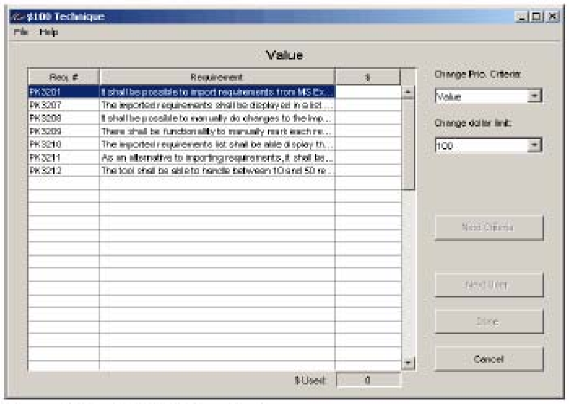
Another feature that is found under the *Options*-menu is the possibility to set the number of paths in the graph to use when calculating priorities with the incomplete pair-wise comparisons (IPC) algorithm [3]. If you are unfamiliar with the IPC algorithm it is recommended that you do not change these values.

$100-Technique

In the $100-technique the priorities are decided by giving each requirement a share of a total budget of $100. This means in practice that each requirement is given a rate in percent of significance according to the currently used criteria.

If a large number of requirements are being prioritized, the $100 limit can be extended to $1000 to make it easier to divide the money.

If this window is opened after an initial prioritization with the Planning Game technique, i.e. it was chosen to assign relative values from the Planning Game's Confirm window; the order of the requirements in the table must be sustained when assigning priorities. This means that you cannot assign a higher value to a requirement positioned in a lower row.



Req. #	Requirement	$
PK3201	It shall be possible to import requirements from MS Ex...	
PK3207	The imported requirements shall be displayed in a list ...	
PK3208	It shall be possible to manually do changes to the imp...	
PK3209	There shall be functionality to manually mark each re...	
PK3210	The imported requirements list shall be able display th...	
PK3211	As an alternative to importing requirements, it shall be...	
PK3212	The tool shall be able to handle between 10 and 50 re...	

Figure 10 The $100 Technique window.

First choose the prioritization criteria to start with by using the combobox in the upper right corner of the window. This combobox will be disabled when the first value has been assigned to a requirement. Then choose if $100 or $1000 shall be used as a limit for the prioritization. This parameter can be changed anytime until the *Next Criteria*-button is pressed for the first time.

You then prioritize by assigning values to the requirements by clicking the $-field and entering the desired amount for each requirement. If you leave a requirement unassigned, it will receive $0.

When all $100/$1000 has been assigned for the first criteria, press *Next Criteria* to prioritize after the last criteria.

When the prioritization is done after an initial Planning Game prioritization, the *Done*-button must be pressed instead of the *Next Criteria*-button to return to the Planning Game window.

When both criteria have been completed, you can choose to finish the prioritization or let another user repeat the prioritization. When the latter is done, the average value for each requirement will be calculated and used as the priority.

9

Root-cause analysis

The root-cause analysis is the phase of the process where requirements in need of further analysis are identified. When the identification is done, a Root-cause analysis and an elicitation of improvement suggestions are done. The support provided by the tool for this part is divided into two windows, the Graph window and the Root-cause matrix.

The Graph window

In this window, the results from the reprioritization are displayed in a graph.

Each requirement's position is shown with an icon. This icon can either be a "+" or one or more circles, to tell releases apart. The requirements marked with a "+" have no release number assigned. The number of circles are decided in alphabetical/numerical order, for instance if we have releases 1, 2 and 3 they would have one, two and three circles respectively.

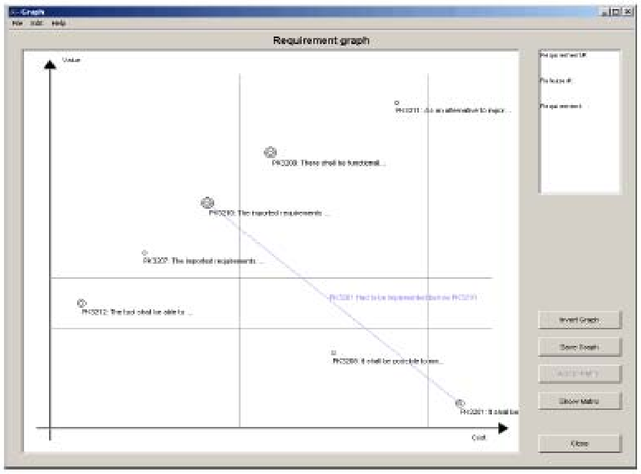


Figure 11 An example of the graph after the planning game method.

The Graph window has a number of functionalities. You can:

- Show the full requirement description by selecting the requirement to show the information about it in the text field in the upper right corner of the window. Another way to see the full requirement description is to hold the mouse pointer over the requirement to see the description as a tooltip.
- Select a requirement by clicking on the icon in the graph.
- Add a requirement to the Root-cause matrix, either by double-clicking on the graph icon or by first selecting the requirements and then pressing the *Add to Matrix* button/selecting the *Add to Matrix* menu item.
- Visualize dependencies between requirements in the graph by holding down the shift key and pressing a mouse button and dragging a line between two requirements.

10

When the mouse button is released over the second requirement a dialog will be shown where you can enter a description for the dependency. The description area can be left empty if not description is wanted.

- Save the graph as a png-file by pressing the *Save Graph*-button.
- Invert the axes of the graph by pressing the *Invert Graph*-button.
- Choose to hide the dependencies and the text for dependencies and requirements by selecting the hide options from the *Edit*-menu.
- Open the Root-cause matrix window if it is minimized or closed by pressing the *Show Matrix*-button.
- Switch between two different modes of showing the support lines by choosing Change view in the *Edit*-menu. Either fixed values can be used to draw the support lines or values relative based on the requirements' positions in the graph.

There are two main types of Graph windows as shown in Figure 11 and Figure 12.

The graph with the horizontal and vertical support lines in Figure 11 is shown when the planning game was used for prioritization, where each area in the graph represents a box-combination from the planning game. By default the support lines are drawn as equally sized boxes. But by switching viewing mode the boxes' size will be adjusted to the number of requirements in them.

The other prioritization methods will result in a graph like the one in Figure 12. There are two support lines in this graph, that by default have the equations *2x* and *x/2*. By switching viewing mode the lines will be drawn with an equal number of requirements in each area. If the number of requirements is not equally dividable by three, either the middle area or the two outer areas will contain an extra point in the graph to sustain symmetry in the graph.

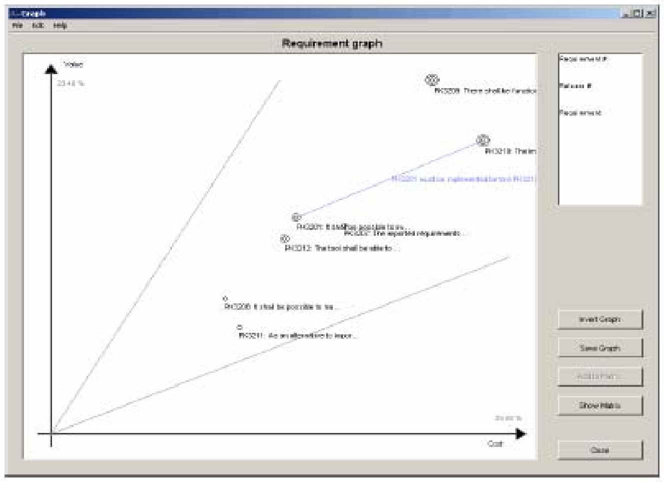
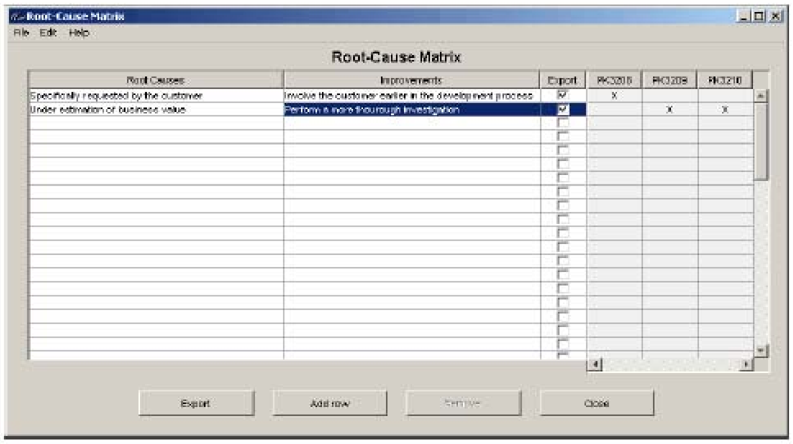


Figure 12 An example of the graph after the pair-wise comparisons method.

11

The Root-cause matrix

This is the window where you can define root-causes and improvement suggestions for the analyzed requirements. Root-causes and improvement suggestions are entered in the left part of the table. In the right part of the table each requirement added from the graph window is represented by a column. The results of this analysis can then be exported to an Excel-spreadsheet.



Root Causes	Improvements	Export	RK3208	RK3209	RK3210
Specifically requested by the customer	Involve the customer earlier in the development process	✓	X		
Under estimation of business value	Perform a more thourough investigation	✓		X	X

Figure 13 The root-cause matrix.

You link each requirement to one or more root-causes/improvement suggestions by double-clicking the cell on the same row as the root-causes/improvement suggestions you want to link it with. (Double-click again to unlink)

To remove a requirement from the root-cause matrix, click on the column header to select and highlight that column, and then press the *Remove*-button.

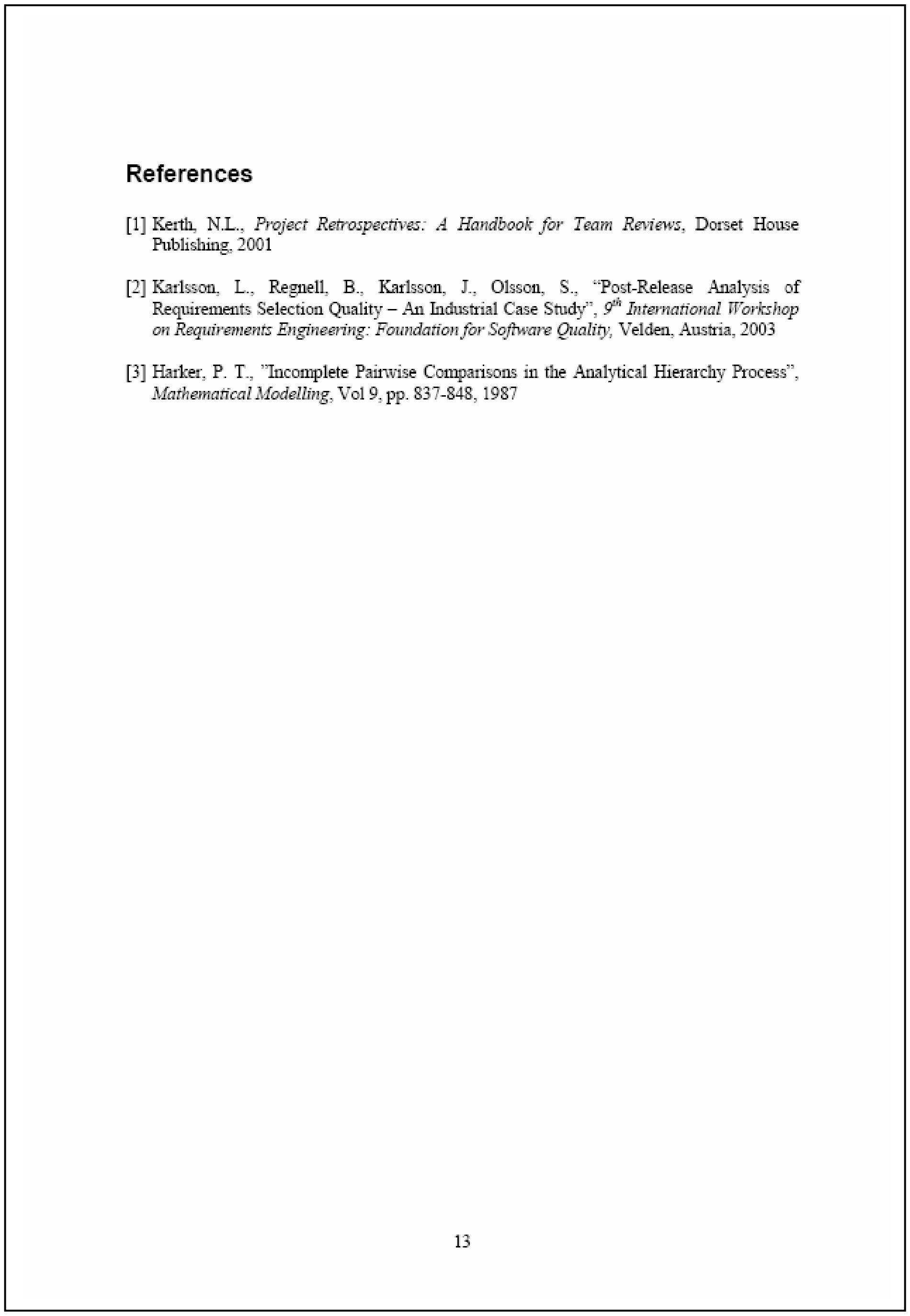
If you need more rows, pressing the *Add row*-button will add an additional row at the bottom of the table.

Exporting and saving results

From the root-cause matrix you can export the results to an Excel-file by pressing the *Export*-button. This will open a save-file dialog where you can choose where to save the Excel-file. *If you choose to save it to an already existing file, that file will be overwritten.* All the columns are always exported but only the rows that have been marked by checking the checkbox in the *Export* column.

You will also be prompted if you want to include the graph in the Excel-file or not. If you choose to include the graph it will be exported to the second sheet while the matrix always is imported to the first sheet.

12



References

[1] Kerth, N.L., *Project Retrospectives: A Handbook for Team Reviews*, Dorset House Publishing, 2001

[2] Karlsson, L., Regnell, B., Karlsson, J., Olsson, S., "Post-Release Analysis of Requirements Selection Quality – An Industrial Case Study", *9th International Workshop on Requirements Engineering: Foundation for Software Quality*, Velden, Austria, 2003

[3] Harker, P. T., "Incomplete Pairwise Comparisons in the Analytical Hierarchy Process", *Mathematical Modelling*, Vol 9, pp. 837-848, 1987

13