Requirements Eng (2002) 7:139–151

Requirements Engineering

Release Planning in Market-Driven Software Product Development: Provoking an Understanding

Pär Carlshamre
Department of Computer and Information Science, Linköping University, Linköping, Sweden

In market-driven software development, release planning is one of the most critical tasks. Selecting a subset of requirements for realisation in a certain release is as complex as it is important for the success of a software product. Despite this, the literature provides little information on how release planning is done in practice. We designed, implemented and evaluated a support tool for release planning as a means for provoking a rich understanding of the task of release planning. The tool utilises a selection algorithm which, based on value, resource estimate and interdependencies, presents a number of valid and good release suggestions. The initial attempt at supporting release planning proved to be based on an overly simplistic and structuralistic view. The results provide ample evidence that the task could be characterised as a wicked problem, which in turn has several implications for the support needed. Although the provotype could indeed support the planner, in its current version it has several serious shortcomings related to the degree of interactivity, underlying models, presentation of information and general appearance. A rich description of the task of release planning is provided. Based on these findings, a list of design implications is proposed, which is intended to guide the future design of a support tool for release planning.

Keywords: Pragmatic algorithm; Provotype; Release planning; Requirements coupling; Requirements interdependencies; Wicked problem

Correspondence and offprint requests to: P. Carlshamre, Ericsson Radio Systems AB, PO Box 1248, SE-58112 Linköping, Sweden. Email: par.carlshamre@era.ericsson.se

1. Introduction

At the highest level of abstraction, release planning could be described as selecting an optimal subset of requirements for realisation in a certain release. To the extent that release planning has received any attention in the literature, this is also the level of description that is usually found.

Release planning is where requirements engineering for market-driven software product development meets the market perspective. It determines which customer gets what features and quality at what point in time. Hence it is a major determinant of the success of the software product.

The selection task is normally preceded by requirements prioritisation, which has received increasing interest in recent years [1,2], and resource estimation, which has been an issue within software engineering since its origins. Once this is done it may seem to be a fairly straightforward task to select requirements from the priority list until the total estimate equals the available resources. However, at the time of prioritisation it is difficult to be fully aware of the context and circumstances present at release planning and hence further judgement is needed. Furthermore, in a previous study [3] it was found that close to 80% of the requirements had interdependencies pertinent to release planning. Thus, a top-priority requirement may require that a low-priority requirement be implemented first, or it may be that one requirement dramatically affects the customer value of another (this should be seen in contrast to stakeholder conflict resolution (e.g. [4]), which is here assumed to be an integral part of the prioritisation). This radically increases the complexity of the selection task.

Despite this importance and complexity of release planning, credible accounts are hard to find in the literature. In order to gain an in-depth understanding of release planning, we designed and evalated a 'provotype' in the spirit of Mogensen [5]. The design was based on an initial understanding of the task, derived from a longitudinal study of one company, and interviews with four other companies. The provotype was evaluated using cooperative evaluation [6] with three release planners at different companies. The aim was to provoke a dialogue with release planners, to study the context in which release planning occurs, what judgements are made by release planners, and also how the problem with interdependencies could be diminished. Specifically, the research question was: What is the nature of release planning, and what implications does this have for the design of a supportive tool?

It should be noted that our primary aim is not to present and evaluate a tool, but to convey an undertanding of a task in its context. However, since the release planning provotype is a part of the instrumentation a brief description is provided, including some of the difficulties tackled during the design of the provotype. Thus, in the next section we address the provotype, the three cases, the procedures employed and some limitations. Then follows a section on the characteristics of release planning which is based on the results of the evaluation. This leads to a number of normative conclusions regarding the design of a support tool for release planning. The paper ends with some reflections on the approach used, some concluding remarks and suggestions for further studies.

2. Methodology Issues

Understanding and describing tasks is a major issue in human–computer interaction as well as requirements engineering, and a multitude of techniques and notations have been proposed over the last couple of decades (see e.g. [7] for a brief overview). However, while traditional task analysis techniques are well suited for capturing flows of activities with fairly static structure, they are less effective for describing highly dynamic and creative tasks. Such tasks tend to be represented by a single rectangle with a tag like 'Compose the picture' or 'Write the document'. From our previous studies we saw release planning as belonging to this category of dynamic problems, and instead we found inspiration in the work by Mogensen [5], who argues that to devise qualitatively new systems while ensuring their usability in the given practice we need to 'ask the practice' itself. This is achieved through provocation, in the sense of *invoking* or *stimulating to action*. Mogensen coins the term 'provotyping' to denote the provocation of current practice, to be distinguished from the literal meaning of prototyping, namely to guess ('proto') a possible solution. The borderline between traditional prototyping and provotyping is not a distinct one. The difference lies in a shift of perspective from studying a tool in use to understanding the task; from creating a future to understanding current practice. As Mogensen states:

> In the strategy of successive prototypes lies a danger of blindness … Once the process … has started, the danger arises that one is led to elaborate the details of the current prototype instead of questioning its underlying premises.

In this spirit, an application was designed and implemented, in the following referred to as the RPP (release planner provotype), or just the provotype. The 'provocation' and data collection were done in the form of an empirical qualitative evaluation of the provotype together with experts.

In the next few sections, we describe the provotype and the procedures.

2.1. The Release Planner Provotype

The RPP is a Java application that parses an XML-like input file containing information about requirements and interdependencies. The design was based on a few assumptions derived from a longitudinal study of one company, and interviews with four other companies:

- The tool should have a graphical interface, and provide good overview of requirements and release suggestions.
- The tool should implement a fast selection algoritm, where the selection is based on the trade-off between requirement value and cost.
- Interdependencies between requirements should be considered by the algorithm, and broken/non-broken dependencies should be clearly presented by the user interface.
- The coupling factors (see Section 2.1.2) for the requirements as well as for the release suggestions should be calculated and presented, in order for the planner to consider these as additional goodness parameters.
- It should be possible to force the inclusion and exclusion of particular requirements, in order to override the automatic selection.
- The algorithm should present more than one suggestion to a release, in order for the planner to make relative judgments.

In terms of visual appearance, it has two main regions: one for the requirements and one for the releases (see Fig. 1).

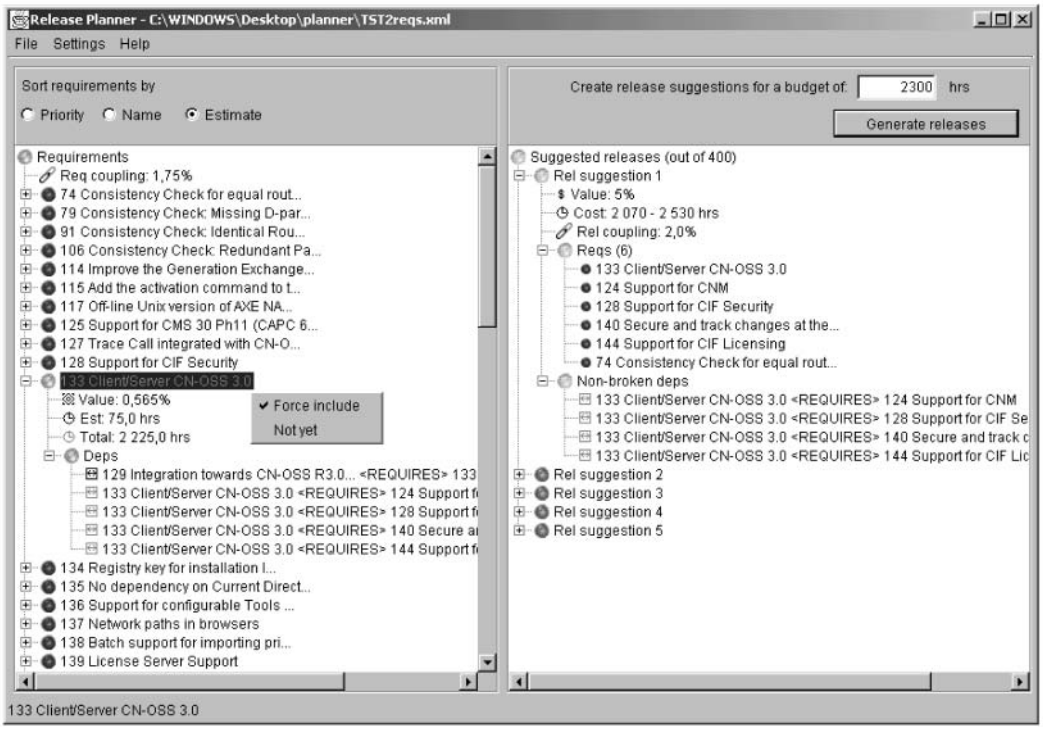

Fig. 1. Screenshot of the release planner provotype.

In the most basic use case, the planner enters a total budget in an input field, clicks a button and the provotype presents a number of good release suggestions in the right main region. There are a few more options, which may best be illustrated by a usage scenario.

Scenario. The requirements manager (planner) is responsible for preparing a basis for the decision-making regarding the next release. He will present the suggestion to the change control board this afternoon.

The planner loads the current requirements database into RPP, sorts the requirements in the order of value (priority), and takes a quick look at the list of requirements tree. By clicking on some of them he can refresh his memory with regard to priorities and estimates. He can also see that some of the requirements that were estimated to be fairly easy to implement (cheap) in isolation in fact are much more expensive, since they require the implementation of several other requirements. He enters the total amount of hours available for design, implementation and basic test, since that is what is included in the estimates. He then clicks the 'Generate releases' button.

RPP now presents five releases in a tree structure. They are ordered by total value. The planner expands the first suggestion, and notes that they will be able to cover 38% of the total value of the current requirements database with the next release. He compares this to the fifth (worst) release suggestion and sees that there are only a few percent that differ. The planner then expands the 'Reqs' tree for the first release suggestion to have a look at the actual requirements included. Several of the top-priority requirements are included, as expected, but there are a few things that need to be corrected. The first requirement in the list is indeed a top-priority one, but should not be implemented in this release since the third-party product that it is supposed to support will not be available in another six months. Similarly, requirement number 417 can only be implemented by Lisa, and she will be on maternity leave this fall. There are a number of such cases, and for each case the planner marks the requirements 'Not yet' in the list to the left.

He then expands the tree of dependencies listed under 'Broken deps' and notices that requirement 164 affects the implementation cost of requirement 27, but 27 was not included for some reason – probably because its overall value/cost relationship was too low. He remembers that the chief architect said something about major cost savings, and decides to include 27 too, by marking 27 and 164 as 'Force include' in the requirements list. Then he clicks the 'Generate' button again.

Now, the first suggestion has a somewhat lower total value, only 32%, but on the other hand the list of included requirements looks very reasonable. The

planner notes that four of the suggested releases have about the same total value, but some of them have less disregarded dependencies, as indicated by the 'Release coupling factor'. Thus, they are more independent of the rest of the requirements, and should be easier to consider. He decides to print out[1] three of the suggestions for the meeting in the afternoon.

The next step is to get an idea of how the release after this one will look. He marks the first suggestion 'Keep this release', in order to make the application regard those requirements as already implemented. He then enters a new budget and considers the suggested releases. Finally, he prints out the suggestions, and as he makes some comments about his own reasoning, he also decides to bring his laptop to the meeting and actually present his reasoning while showing the RPP on the widescreen. This way the members of the change control board will have a chance to give immediate feedback.

Except for the features mentioned in the scenario, it is also possible to set the number of suggested releases and the search depth (explained below). Furthermore, it is possible to set the accuracy level of the resource estimates. Setting this to 10%, for example, will not affect the algorithm itself, but only the presented total cost for the suggested release, so that a 10% interval about the nominal value is presented.

2.1.1. The Pragmatic Planning Algorithm

Selecting an optimal subset of objects from a larger set, where each object has a cost and a profit contribution, and the selection is bounded by an upper limit, is known within operations research as the 'knapsack problem'. Specifically, the release planning problem, in its simplest form, is a binary knapsack problem since requirements cannot be partially selected for realisation. More formally the problem is stated as:

$$\text{maximise} \sum_{j=1}^{n} v_j x_j$$

$$\text{subject to} \sum_{j=1}^{n} r_j x_j \leq b; x_j = 0 \text{ or } 1;\ j = 1, ...,\ n$$

when n is the total number of requirements; v and r represent the value and the estimated resource demand, respectively; and b is the total available resources (sometimes referred to as 'budget').

[1]Printouts were not available in the evaluated version of the provotype.

Although the knapsack problem is NP-hard, there are several approximate algorithms for finding one optimal solution in polynomial time (e.g. [8]). Jung [9] applied one of these techniques to the release planning problem with $n = 14$, and found that the 'solution time in a PC was negligible', but our own studies indicate that the computation time increases dramatically with real-world-sized requirement sets, and is further slowed down when dependencies are considered by the algorithm.

In practice, there are a number of aspects to consider when comparing the release planning problem to the more refined knapsack problem. First, computation speed has to be kept to an absolute minimum, in order to allow for immediate interaction, i.e. direct manipulation. Second, our preliminary studies indicated that the release planner would not be interested in one 'optimal' solution, as computed by some magical algorithm. Instead, the nature of the problem requests that a *number* of *good* suggestions are presented for the planner to consider and modify based on other aspects than just resource demands and relative values. Third, requirements have dependencies which must be accounted for.

For these reasons a modified version of the so-called 'greedy algorithm' (e.g. [8]) was implemented, where the search is performed on an ordered set of requirements. In our case, the search depth is limited to a specific but adjustable value. For a certain depth d, the algorithm produces d^2 good suggestions, each guaranteed to include at least one of the requirements 1..d in descending order of value, and then selects the five (or 10 or 15, as set by the user) best of these. Goodness is defined simply as the sum of the individual values.

Thus, this 'pragmatic algorithm' is not guaranteed to find an optimal solution, but to find a number of reasonably good ones including top-priority requirements, very quickly.

2.1.2. Considering Interdependencies

As mentioned previously, one of the parameters that the release planner has to consider is the interdependencies between requirements. A previous study [3] of five independent requirements repositories indicated that only about 20% of the requirements were singular, or independent of each other. The most common types of interdependencies found were 'And', 'Requires', 'Affects customer value of' (CVALUE) and 'Affects implementation cost of' (ICOST). CVALUE dependencies are always related to the use of the product, and can thus be seen as work-related, whereas ICOST dependencies are usually system-oriented. A few examples may clarify the meaning of these types.

Example 1: AND. A printer requires a driver to function, and the driver requires a printer to function.
Example 2: REQUIRES. Sending an email requires a network connection, but not the opposite.
Example 3: CVALUE. A detailed online manual may decrease the customer value of a printed manual.
Example 4: ICOST. A requirement stating that 'no response time should be longer than 1 second' will typically increase the cost of implementing many other requirements.

The algorithm considers requirements interdependencies to the extent that is possible from an algorithmic point of view. Interdependencies of type AND and REQUIRES are always obeyed, so that if R_1 requires R_2, R_1 will not be included in the release unless there are sufficient resources enough to include R_2 too (although R_2 may be included without considering R_1). Similarly, in the case of an AND, either both requirements or none will be included.

In the case of a weaker interdependency, the algorithm cannot make any reasonable decision; instead the judgement must be made by the planner. For example, if R_1 increases the customer value of R_2, no general conclusion could be drawn by the application about the sequence of these, or whether to implement them in the same release. Instead, the planner will have to make the decision, based on their individual values.

However, in order for the planner to make these decisions, the information must be readily present, and hence the application indicates clearly what interdependencies have been disregarded (Fig. 1).

In the case of the ICOST dependency, i.e., if R_1 affects the implementation cost of R_2, it is often considered better to implement R_1 first or in parallel with R_2 in order to minimise risk (if R_1 is expected to increase the cost of R_2), or to maximise internal savings (if R_1 is expected to reduce the implementation cost of R_2). But there may be exceptions to this rule. As an example, R_1 may represent the migration to a new version of the operating system, which is expected to simplify certain operations (R_2) in a networked environment. On the other hand, there is a risk that the release of the new operating system will be delayed, so it is wise to disregard this interdependency in order not to risk any delay on one's own part.

A set of requirements can be more or less interconnected by dependencies. If there are no dependencies at all, the selection of requirements for a particular release is trivial once the requirements are prioritised, but the complexity of the selection problem increases with the number of dependencies. In [3] we proposed a measure of how tightly interconnected a set of requirements is. This *requirement coupling* factor was defined for an arbitrary set of requirements as the ratio between the number of actual dependencies and the number of possible dependencies.

Another coupling measure was defined for a release, i.e. a particular subset of requirements. Paritioning the full set of requirements to create a release will potentially break some of the dependencies that exist within the full set. It was suggested that minimising the number of broken dependencies may simplify further development and increase stability. The *release coupling* factor was thus defined as the ratio between the number of broken dependencies and the number of existing dependencies within the full set of requirements. Both these coupling factors were calculated and presented by the provotype to provide additional information for the planner to consider (or not).

Again, for a full discussion of the dependencies and the measures, the reader is referred to Carlshamre et al. [3].

2.2. The Three Cases

Three different requirements managers, henceforth referred to as planners, at different companies were involved in the evaluation. Each of them was responsible for a mature software product, i.e. it has been available on the market for several years. The products are not currently aimed at mass markets with completely anonymous customers, but the development situation is market-driven and evolutionary, with customers in several different countries.

- *Case 1: A product unit within a large telecommunications company.* The product is a tool for analysis of large data tables in telephone exchanges. It is marketed as a stand-alone product, but is also delivered as a module in a large operations and maintenance system, to which some releases must be aligned. The database contained 74 selectable requirements.
- *Case 2: A small company with one product and services associated with it.* The product is a support tool for requirements management with special attention on prioritisation. The requirements database contained close to 200 requirements. At the time of evaluation, 47 of these had been analysed sufficiently to be considered candidates for the next release.
- *Case 3: A medium-sized company with three different product areas.* The product involved in our study is a system for storage and management of X-ray images for medical services. The product has several different branches based on the same core. These branches have partly different markets and partly different user

groups. The requirements database contained close to 1100 requirements, of which 172 were candidates for the next release.

In all three cases, the respondents are experienced requirements managers with explicit interest in requirements engineering methodology. They are knowledgeable about architectural and implementation issues as well as market demands, and are able to comfortably and swiftly shift between these perspectives.

2.3. Procedure

The sessions with the planning experts were planned and designed as cooperative evaluation sessions. Cooperative evaluation is a technique developed by Monk et al. [6] to improve a user interface specification by detecting usability problems in an early prototype. Although it is inexpensive and straightforward, it has been proven to be effective for finding usability problems, and it is well suited for the situation where a designer wishes to evaluate a design and gain an understanding about a task in its context with the help of a domain expert (actual user).

In short, representative users work through representative tasks chosen by the designer, and as they work they explain to the designer what they are doing and ask questions. The designer allows the user to make mistakes and uses the user's questions to elicit further information about a potential problem. Unexpected behaviour and comments about the interface are viewed as symptoms of potential usability problems.

In each of the three cases, the users worked through a series of eight predefined tasks. Each session took about 2 hours and was recorded on videotape, backed up by note-taking. At the end of each session, the requirements managers filled out a short questionnaire. The videotapes were then analysed[2] with regard to salient issues (positive comments or problems). In total, 86 such issues were found. These were then clustered under 15 captions, which in turn were grouped into four themes in an iterative interpretative process.

In all cases, the data used in the provotype consisted of excerpts from the companies' actual requirements databases, which had been coded into an XML-like format. In order to capture the interdependencies in each requirements set, the evaluation sessions were preceded by preparatory 'interdependencies sessions', carried out a few days in advance. During these preparatory sessions, dependencies between requirements were identified by means of pairwise assessments (for a description of this procedure, see [3]). Notes were taken during these sessions as well.

As a complement to the provotype, a graph model had been prepared and was provided on paper, with all the requirements illustrated as nodes and the dependencies as edges. This graph was just laid on the table beside the computer, with little explanation. The aim was to see if the planners preferred turning to the graph in some situations, which in turn would be an indicator of its usefulness.

The analysis is based on the combined data from both the preparatory and the evaluative sessions.

2.4. Limitations and Validity

The evaluation setting reflects a situation where a single user works with the support application. Although this is an important use situation it is not the only conceivable one. As suggested in the previous usage scenario, the application may be used during a meeting of several people. It is unclear how the results from this study would apply to a group usage scenario.

A related limitation is the size of the products in our study. In all cases the products are small enough for one person to have a fair grasp of the requirements, the market demands, architectural issues etc. However, our conjecture is that the larger the product, the more pressing is the need for visual tools. Some of the results, e.g. the need for a graphical representation of the requirements set, may even be more valid in cases where several people with different perspectives should come to an agreement.

The companies involved in this study all have a fairly good picture of the customers, which may not be the case when shrink-wrapped products are aimed at mass markets. The relationship with the customers plays an important role in release planning for the companies studied, and continuously affects the decision-making. In a mass-market situation with anonymous customers, the decision-making may appear to be easier in that there are fewer known variables to consider, and the planner's role can be more of an absolute ruler's. On the other hand, the decisions are necessarily less well informed.

In terms of general validity of the findings, it is clear that all results are organisation-dependent to some extent. However, our striving to understand and describe the underlying mechanisms of release planning, rather than shallow phenomena, is based in a firm belief that the results should be transferable to many other contexts.

[2]Thorough analysis of video or tape recordings is not suggested as a part of the cooperative evaluation technique in industrial settings, due to the time pressure ([6], p. 35).

3. The Nature of Release Planning

The following attempts to illustrate the nature of release planning. Again, the provotype evaluation was used as a means of learning about the task in its context, rather than to establish the level of utility of the tool. Therefore, the results of the evaluation is presented as a part of the task description. Quotations from the records have been included at several points to clarify and/or emphasise a particular issue.

3.1. Overview

The basic planning problems differ significantly between traditional project-oriented systems development and market-driven product-oriented development. Simplistically put, the planning task in the project-oriented case is one of estimating the resource demands and delivery date for a more or less fixed set of requirements. By contrast, in the product-oriented case there is a fixed delivery date, a more or less fixed set of available resources, and the task is one of selecting an optical subset of requirements for implementation. Release planning also differs from traditional product planning as found in manufacturing industry (e.g. [10]). Here, considerations are more similar to what a software product developer needs to make before the product is developed and marketed in the first place, such as the company profile, available competence, production capacity and financing.

The two most important attributes pertinent to release planning are value and cost. Each of these may by itself constitute a complex relationship of different aspects. In the case of 'value', for example, this is often an amalgamation of strategic business value for the vendor, long-term and short-term value for the customer (which in turn may be a long list of various customers, with various importance, in different markets, see [11]), compliance with laws and regulations as well as new computing platforms, and internal cost savings (e.g. 'cleaning up the code'). Two of the respondents in our study also commented that there are even considerations such as 'fun or boring to implement' to make about the requirements.

In the case of 'estimated resource demands', the planners also have to consider specific skills required for certain requirements, e.g. UI design, Java expertise, operating systems expertise. This in turn involves verifying a suggested release plan against individual employees' workload and vacation schedules, recruitment plans, etc.

One of the companies (Case 3) had several products based on the same software platform, and thus the planner had to consider synergy effects among products. Another company (Case 1) had one product supporting several computing platforms, hence the planner had to take this into consideration. Furthermore, all these aspects are evaluated from various strategies, in order to maximise overall value for all stakeholders while minimising the risk of delayed delivery. Indeed, the release planners have a difficult task to solve.

3.2. Characteristics of the Task

None of the respondents consider release planning to be an activity isolated to only one occasion per release. Instead, release planning *includes* prioritising the requirements, estimating their resource demands, and selecting requirements for a certain release. These activities are usually performed continuously but with an emphasis at the end of a previous release, as would be expected, in order to allow for an up-to-date understanding of the market situation, as well as recently captured requirements. Release planning is also part of the overall strategic product planning, which is a continuous work of meeting with customers and listening to the market changes and demands. When asked about how much time is spent on release planning the answers varies between 120 (Case 2) and 1800 (Case 3) person-hours per year, which may indicate a discrepancy in the definition of scope of the task, rather than in ambition. Between 50% and 75% of this time is spent on prioritising and estimating resource needs. The typical number of releases per year varied among the respondents between two and four major releases, plus minor releases such as bug fixes and minor updates.

In all three cases there was one person (the respondent) responsible for working out a suggestion for a new release. All respondents have a degree in computer science, but their respective roles and perspectives range from an emphasis on implementation issues to a considerable predominance of market issues.

The suggestion produced is then discussed and approved by a more or less formal meeting involving people with various perspectives and responsibilities. In one case, which could serve as a model, this group of people is a formal board with representatives from marketing, systems development, training and executive management.

All respondents claim that they have a fairly good picture of what should be included in the next release, but that there is always discussion about the various parameters (priority, estimates, dependencies, custo-

mers, etc.) during these meetings. These discussions tend sometimes to be frustrating, because of the many variables and considerations.

3.3. General Opinions about the Provotype

On a general level, the planners were quite positive towards the RPP. One planner even stated that 'This could be useful to us ... right now, in its current version.'[3] The need for tool support was also emphasised at several points: 'With lots of requirements and dependencies, you couldn't do this without tool support.' The most important role for the tool was regarded as supporting reasoning, to find arguments for and against various solutions, and to be used in preparation for meetings with the product committees.

As for the design of RPP, opinions were fairly positive too. The planners were all quite comfortable with the tree structures, and they found the presentation to be good, with a few exceptions, such as the need for horizontal scrolling, and the difficulty in comparing release suggestions against each other. This positive attitude towards the tool is important to note although it was fairly obvious to an observer that the provotype was flawed in many respects, and the video analysis made this even clearer.

3.4. Release Planning is a Wicked Problem

The whole approach of having an algorithm which, based on a few attributes, can find 'optimal' releases (although in a relaxed sense) suggests that there is a strong structure and rationality to the release planning task. This turned out to be an overly simplistic view. The data shows ample evidence that release planning is what Rittel and Webber [12] characterise as a wicked problem, in contrast to 'tame' problems. Tame problems are not trivial, but can be understood sufficiently to be analysed by established methods and it is clear when a solution has been reached. They may also be amenable to automated analysis [13]. Wicked problems, on the other hand, have a number of properties that distinguishes them from tame problems, and thus the procedures needed to solve them. For example, Rittel and Webber suggest that:

- Wicked problems have no stopping rule. The solving stops when the solver runs out of time, money or patience, and thinks 'this is good enough'.
- Wicked problems have better or worse solutions, but no optimal one.
- Every wicked problem is essentially unique. Despite long lists of similarities with a previous problem, there is always an additional distinguishing property that is of overriding importance.
- Wicked problems have no objective measure of success, and there is no immediate test of a solution.
- Wicked problems require iteration – every trial counts.
- Wicked problems have no given alternative solutions – these must be discovered and compared.
- There is no definitive formulation of a wicked problem. To define the problem is the same as defining the solution.

Wicked problems are found in many areas and situations: in politics, design, city planning and teaching, to give just a few examples. Next, we provide some evidence that this description fits quite well to the release planning task.

3.4.1. The Value of a Release is Hard to Define

Each requirement in the RPP has a value assigned to it. These values were given by the respondents as attributes of the requirements. The basis for these values differed among the cases. In Case 1 the value was based on an overall judgement. In Case 2 it was based on two parameters – strategic and operative value – and in Case 3 the value was based on strategic and operative value and health risk (since the product was used in health care). The value for each requirement was converted into a percentage of the total value for all requirements, so that each requirement presented its relative contribution to a new release.

The total value of a release suggestion was then calculated by the RPP as the sum of the individual values of the included requirements, so that the user could conclude, for instance, that 'for 3500 h we can get 43% of the total value represented in the requirements database'. Although such arithmetic calculations are questionable, it was generally agreed that this is a straightforward and sufficiently precise way of dealing with values. The suggestions were also sorted by this total value, making it easy to compare the values of the suggestions.

However, although the planners had put some effort into assigning values to the requirements, the total value of a release was treated as only a minor determinant, and in all three cases there was discussion about what really makes a 'good' release.

One planner thought that the percentage of top-priority requirements included in the release should count: '... like, if you have X requirements in the release suggestion ... how many of these are represented

[3]Quotations are translated from Swedish and mildly polished for readability.

among the X highest-priority requirements, for example'. This was based more on a feeling than rational grounds. As the algorithm sometimes suggested releases that did not include very many of the top-priority requirements, this didn't feel right, even though the planner could understand this mechanism perfectly well.

Another suggestion was that the number of included requirements should also affect the total value, so that more requirements gave a higher value. This was also based on a feeling that 'more is better'. This kind of reasoning always ended in the conclusion that the only rational criterion was the sum of individual values, since any other judgement should already have been accounted for in the prioritisation process. Nevertheless, these discussions underline the wicked nature of the problem.

3.4.2. Criteria Cannot be Defined in Advance

It became clear that the two parameters, cost and value, were far from sufficient in determining the goodness of a release suggestion. On several occasions it was pointed out that more information about the requirements was needed, some of which was not even present in the original database. Suggestions included 'specific stakeholders', 'specific markets', 'theme for the release', 'interval improvements or visual to customer'. One planner said: 'There are so many parameters that you consider … that are never really quantified.' Also, in cases where the value of a requirement was an amalgamation of more than one value attribute (Cases 2 and 3), the planners would like to see them all to be able to optimise on different attributes for comparison.

The requirements were identified by unique numbers and slogans, which was sufficient for most purposes according to the planners. But on several occasions a richer description was requested to judge the value of a requirement. The planners were unable to specify what exact information was needed, but claimed that they needed to get a better 'feel for what it really meant'.

3.4.3. Judgements Are Always Relative

The ability to compare requirements against each other and, even more importantly, release suggestions turned out to be very important, which emphasises that judgements are predominantly relative in release planning. The RPP did not provide any tools to assist such comparisons, which was also noted as one of the major three drawbacks of the provotype. Comparing requirements, or viewing them simultaneously, was important primarily in two cases. First, where there is a dependency between two requirements it is important to be able to see these two at the same time to recall the nature of the dependency. Second, when the algorithm suggests something that is inconsistent with the planner's intuition, he usually wants to investigate the reasoning behind this, which in turn involves comparing, for example, included and excluded requirements.

Comparing releases is a continuous issue. The provotype only presents a number of suggestions which, for natural reasons, tend to be fairly similar in terms of total value and/or the specific requirements selected. The difference in total value between two suggestions may be less than 1%, and the number of distinguishing requirements between two suggestions may be quite small. This emphasises the need for supporting comparison, e.g., by highlighting discrepancies and/or similarities.

Another related issue was the importance of judging what requirements had been *excluded* from a release, which turned out to be as important as seeing what requirements had been selected. The excluded requirements were an important determinant of the value of a release suggestion, since it indicated which customers or markets might be disappointed: 'All requirements that have made it this far are fairly important … there is always someone out there who is waiting for them to be implemented … so you have to … you have to call them and tell them that, unfortunately this didn't quite make it this time.' Again there was a relative judgement of the positive implications for certain markets at the expense of others. This also underlined the need for managing several consecutive releases (see below) to even out added value among stakeholders, markets and releases.

3.4.4. Planners Discover Properties as They Plan

At several points the planners felt that they needed to add information to the requirements descriptions as they were planning. Examples include comments about relative values, resource estimates and dependencies. On some occasions the planners even wanted to change the value of some attributes based on new insights. For example, one planner said that '… if requirement [X] takes 120 hours, which is probably true, then requirement [Y] cannot possibly require only 100 hours. We must have missed something.'

Furthermore, it was evident that release suggestions have properties on their own, which are not easily seen as a function of the individual requirements. In one case, the planner contemplated a release suggestion which he thought was good but far from expected, and stated: '… when you see the different suggestions … you see different themes … even if I know that [the provotype] doesn't know anything about themes. Interesting!' Other examples of holistic characteristics include the balance between strategic and operative value, and also the balance between invisible (e.g. architectural) improve-

ments and visible features (which the customer would be more willing to upgrade for). Still other adjectives that were used to describe release suggestions (beyond 'good' and 'bad') included 'interesting', 'risky', 'expensive' (because it would demand a large administration overhead) and 'boring'. This indicates that there are holistic traits to releases which are not apparent to the planners until a suggestion is presented. This would in turn imply a serious impediment to any analytical and algorithmic approach to release planning.

Finally, while studying and reasoning about dependencies between requirements, and especially the possibilities for working around them, the planners also considered the architecture of the respective products, which gave new implications for what requirements should be considered in the same release.

3.4.5. Need Support for Alternative Models

A task with the characteristics as described above calls for quick shifts between different perspectives. This, in turn, puts demands on the support tool to be able to present the same information in different ways, according to different models. This was one of the most salient drawbacks of the provotype, and although the planners did not complain about this explicitly to a great extent, it was apparent to the observer.

The provotype implied that the requirements and the release suggestions should be seen as lists, or rather hierarchies, as in the now ubiquitous MS Windows Explorer metaphor. There are some advantages to this model; it is well known by most computer users, and it is well supported in terms of ready-made class libraries. However, from a task or cognitive perspective there is nothing that supports the idea that it is a good way of representing a set of requirements, or a set of releases suggestions. On the contrary, the planners frequently talked in terms of time, and when they drew sketches to explain something the predominant model was either that of a time line or a set of nodes and edges. It should be noted, though, that the planners were very obliging towards the hierarchical model: '... it might be possible, you know, to represent time in the tree structure too ...'.

The temporal aspect of requirements and interdependencies was not represented in any way in the provotype. But even if only a few requirements were associated with specific points in time (or specific releases during the year), it was important for the planners to be able to specify this during planning as a way of reducing the complexity: 'If I could say ... that these requirements should not be considered until the December release ... then I could forget about them now.'

The paper-based graph model (nodes and edges) was efficient in discussions about interdependencies. Here, the time line perspective could not contribute to the reasoning. Likewise, the hierarchical structure was clearly insufficient in providing both overview and details on demand. The graph model provided a good overview of the dependencies, and the planners were able to identify clusters of requirements, as well as 'problematic' requirements that needed special attention.

3.5. One Release is not Enough

The predominant view suggested by the provotype is that release planning is done for one release only, namely the next one. Although a rudimentary mechanism was provided to control whether a specific requirement should be eligible or not, by marking it 'Not yet', this was far from sufficient. For example, as noted previously, the requirements that were not selected by the algorithm were as important as those selected, because they indicated what customers or markets might be disappointed. Thus, not only would these customers have to be compensated in the next release, but the sales personnel had to be able to *say* that those requirements were due in the next release. This is an important effect of fixed release dates, which is beneficial for the customer–vendor relationship.

Moving around requirements between consecutive releases at a planning level stood out as very important, and the unwieldy mechanism for this purpose did not support the speedy and easy interaction that the task and the planners' cognitive processes demanded.

3.6. New Insights about Interdependencies

Based on previous interviews and discussions, our understanding was that the selection algorithm could and should only obey functional dependencies, and just present value-related dependencies for the planner to consider or not. While working with the provotype, this turned out to be an incorrect presumption, because value-related dependencies can be as strong or imperative as functional dependencies. In several cases the planner realised that a CVALUE dependency could not be broken, because there was a strong relationship between the involved requirements on a task level. For example, if requirements R_1 and R_2 support different steps of a certain task, excluding R_2 would upset the customers more than excluding both R_1 and R_2. Thus, there is an imperative value dependency between R_1 and R_2, even if there is no relationship from a functional

standpoint. Hence, this division of functional (as more important) and value-related (as less important) dependencies proved to be erroneous on several occasions.

3.7. Credibility Issues

Introducing a tool that claims to solve parts of a difficult problem by use of a more or less obscure algorithm is necessarily associated with credibility issues, and the fact that the prospective users are experts in their area does not mitigate such problems. Trust and credibility issues are very important aspects of decision support systems, which the provotype could be characterised as. The data offers several examples of such issues.

For example, at the beginning of the evaluation sessions, the planners tended to 'test' the algorithm under great suspicion: 'If I have the feeling that the algorithm isn't right, I will have to verify everything manually anyway.' Each time the algorithm suggested something that was inconsistent with the planner's intuition, he immediately requested an explanation of its behaviour (cf. [14]). In all three cases, much time was spent on explaining the selection algorithm, on (implicit or explicit) request by the planners. After a while, however, the planners gained confidence, and also adapted their expectations so as to consider the release suggestions with a grain of salt. This phenomenon is known as calibration within decision support research.

However, the fact that all information was presented with such apparent accuracy seemed to increase and prolong this calibration. The coupling factors were presented with three decimals (for no good reason), requirements values and most importantly the release suggestions' individual values were presented with one decimal (because sometimes they only differed in the first decimal). This gave a false impression of accuracy and exactness (the digital watch syndrome). This was further emphasised by the fact that the release suggestions were ordered by total value, even if the difference in value was very small. On many occasions, the planner thought that, for example, the third suggestion was much better than the first–regardless of the calculated value. He thus lost a bit of confidence in the algorithm–not because any of the suggestions were bad, but because they were in the wrong order: 'If the application cannot pick the best one of these two correctly, how could it possibly pick a good one at all?'

For credibility reasons, it was also important to the planners that some of the top-priority requirements were selected for all releases. Even though they perfectly well understood that if the top-priority requirements are 'large', either by themselves or via imperative dependencies, and the budget is small, none of them may be selected. It was as if, even though the top-priority requirements didn't quite make it, the planners wanted to know that the algorithm had at least considered these requirements. However, again, this effect tended to fade away with increased use. Eventually it was concluded by all planners that the suggestions were in fact quite close to what they would have come up with manually.

In general, the approach of suggesting ready-made solutions to wicked problems is problematic. Muir [15] makes a distinction between prosthetic systems, which provide solutions, and instrumental systems, which represent one of several resources that the decision maker may consider. In the latter case, the human is explicitly recognised as having the authority and responsibility for decision making. The provotype could be said to represent a mix between these two classes, in that it presents several, but ready-made, solutions for the planner to consider. However, our understanding points in the direction of a more clean-cut instrumental approach.

4. Design Implications for a Support Tool

The results from the cooperative evaluation sessions provide numerous important implications and limitations for the design of a supportive tool for release planning, some of which are described in brevity here:

- On a general level, the tool needs to be much *more interactive* than the provotype in order to support the creative and wicked nature of the task. It needs to *support exploration*. The notion of a 'generate–evaluate–adjust–regenerate' cycle as suggested by the provotype is not sufficient. Instead, a *direct manipulative* interface, where calculations are performed in the background and *information is displayed passively* as a result of the planner's acts, is more suited for the task.

  An important consequence of this is that a selection algorithm may not even be necessary. Instead the planner could drag and drop requirements onto a release, getting instant feedback on the total value and total resources needed, until he was satisfied.

  On the other hand, it is reasonable to believe that *an algorithm that can be used on demand* will be considered convenient. For example, it could be of assistance in situations where the planner has selected a few requirements for a particular release, and then wants the application to suggest how to fill up the release to the budget limit.
- The tool needs to be able to present information according to *several different models.* The Explorer metaphor is well suited for finding and inspecting

particular requirements, but is insufficient for most other parts of the task. A *time line* of some sort is necessary for the planner to be able to specify where in time particular requirements should be considered and vice versa. A *graph model* is needed for reasoning about dependencies between requirements. A *set model*, providing the ability to group requirements arbitrarily, i.e. by semantics unknown to the application, is needed to be able to compare characteristics of sets of requirements.

- *Comparison* of requirements, release suggestions and arbitrary groups of requirements, respectively, need to be supported so that differences and/or similarities are highlighted.
- The tool must support planning for *several consecutive releases*. A related issue is the need for highlighting at least high-priority requirements that have been excluded by the algorithm.
- The selection algorithm needs to *consider the strength of interdependencies, rather than the type*. There may be imperative value-related dependencies as well as fairly negligible functional dependencies.
- It is probably *not worth the effort* for the algorithm to handle more parameters than value, estimate and dependencies. The number of parameters considered by the planners is very large, and modelling them all for all requirements is neither possible nor necessary. It is impossible because they are not all known in advance, and it is unnecessary because the planner only uses them in special cases.
- It is important that the support tool 'keeps a *low profile*' or 'acts humbly', in order to gain credibility among planners. By this we mean, for example:
  - that it does not pretend to be exact unless it is needed and adequate;
  - that it does not discriminate between entities, e.g. the total value of releases, unless the discrepancy is significant;
  - that the planner has full control over the interaction.

Thus, in terms of design trade-offs, comprehensibility is more important than exactness, presentation is more important than calculation, and an instrumental approach is preferable over a prosthetic one.

It should be pointed out that, given the wicked nature of the task, it can only be partly supported with computer-based tools.

5. Reflections on the Provotyping Approach

Space does not permit a full account of the findings from the interviews and the evaluation sessions. Yet, the richness of the data should be apparent. We believe that this kind of knowledge could not be captured by means of traditional task analysis techniques. The provotype represented a fair attempt at a design based not only on several interviews but also a longitudinal study that has gone on for over a year. Still, the design was flawed on several points, due to an insufficient understanding of the task. It is fair to say that not even the planners thought of the release planning problem as having the complexity and the wicked nature that we discovered together. One planner stated, before we started the session: 'It is impossible to pick out an *optimal* release manually' (my emphasis), indicating the position that there really are optimal releases – one only needs help to find them.

To give a couple of examples, the importance of excluded requirements or the emphasis on comparisons would be very difficult to capture during regular interviews or passive observation of someone doing release planning without a tool. These are also examples of how the practical use of a fully functional tool, using the planners' own data, uncovered aspects that may not be important without the tool. The possibilities offered by the provotype encouraged the planners to reflect on their own expertise.

Furthermore, it is with some consternation that we realise that many of the fundamental design deficiencies of the provotype would be hard to identify even with traditional usability testing. Since the planners were fairly positive, it is our contention that an ordinary commercial situation would have misled us to accept the fundamentals, request a few changes and some added features, and then proceed to implementation. This also emphasises the difference between this approach and traditional prototyping. The focus is on understanding the task, not fixing the tool.

6. Conclusions and Future Research

Release planning is an important and complex task in a market-driven software product development situation, but little information on this task is available in the literature. We designed a prototype to provoke a rich understanding of the problem, and also to provoke normative design knowledge for a computer-based supportive tool.

Our results show that release planning could be characterised as a wicked problem, which in turn has several implications for the support needed. Most importantly, it does not easily lend itself to analytical solving approaches. Our initial attempt at supporting release planning proved to be based on an overly simplistic and structuralistic view. Although the provotype could indeed be helpful to the planner in its current

version it has several serious shortcomings related to the degree of interactivity, underlying models, the presentation of information and the general appearance. Based on these findings, a list of design implications was proposed, which is intended to guide a future design of a support tool for release planning, and to avoid some of the most serious pitfalls.

We also conclude that our approach to studying the release planning task was indeed a fruitful one, generating a level and detail of knowledge and understanding which would probably not have been possible by means of conventional interviews and task analysis techniques.

The obvious question for future studies is to what extent the design implications presented here are valid. Each implication could be viewed as a hypothesis for further research.

Acknowledgements. The author would like to thank the anonymous participants of the cooperative evaluation sessions, and Dr Stefan Engevall at the Department of Mathematics, Linköping Univervity, for insightful comments on the pragmatic algorithm. Thanks also to Dr Jonas Löwgren, Dr Kristian Sandahl, Stefan Holmlid, Åsa Granlund and the anonymous reviewers for valuable comments on earlier drafts, and Pelle Eriksson and Tobias Wirén for initial help with the Java code.

References

1. Karlsson J, Ryan K. A cost-value approach for prioritizing requirements. IEEE Software 1997;14(5):67–74
2. Karlsson J, Ryan K. Improved practical support for large-scale requirements prioritising. Requirements Eng 1997;2(1):51–60
3. Carlshamre P, Sandahl K, Lindvall M, Regnell B, Natt och Dag J. An industrial survey of requirements interdependencies in software release planning. In: Proceedings of the 5th IEEE international symposium on requirements engineering, 2001, pp 84–91
4. Boehm B, In H. Identifying quality requirement conflicts. IEEE Software 1996;13(March):25–35
5. Mogensen P. Towards a provotyping approach in systems development. Scand J Inform Syst 1992;4:31–55
6. Monk A, Wright P, Haber J, Davenport L. Improving your human–computer interface: a practical technique. Prentice-Hall, London, 1993
7. Benyon D. The role of task analysis in systems design. Interact Comput 1992; 4(1):102–123
8. Martello S, Toth P. Knapsack problems: algorithms and computer implementations. Wiley, Chichester, 1990
9. Jung H-W. Optimizing value and cost in requirements analysis. IEEE Software 1998;15(July/August):74–78
10. Richman BM. A rating scale for product innovation. In: Spitz E (ed). Product planning. Auerbach, Princeton, NJ, 1972
11. Regnell B, Höst M, Natt och Dag J, Beremark P, Hjelm T. An industrial case study on distributed prioritization in market-driven requirements engineering for packaged software. Requirements Eng 2000; 6(1):51–62
12. Rittel H, Webber M. Planning problems are wicked problems. In: Cross N (ed). Developments in design methodology. Wiley, Chichester, 1984, pp 135–144
13. Shum S. Negotiating the construction of organisational memory using hypermedia argument spaces. In: Proceedings of the workshop on knowledge media for improving organisational expertise, First international conference on practical aspects of knowledge management, Basel, Switzerland, 1996
14. Bauhs JA, Cooke NJ. Is knowing more really better? effects of system development information in human-expert system interactions. In: Proceedings of ACM CHI'94 conference on human factors in computing systems, vol. 2 of Interactive posters, 1994, pp 99–100
15. Muir BM. Trust between humans and machines, and the design of decision aids. Int J Man–Machine Stud 1987; 27(5/6):527–539