Linköping Studies in Science and Technology
Dissertation No. 726

A USABILITY PERSPECTIVE ON REQUIREMENTS ENGINEERING

From Methodology to Product Development

Pär Carlshamre

INSTITUTE OF TECHNOLOGY
LINKÖPINGS UNIVERSITET

Department of Computer and Information Science
Linköpings universitet
SE-581 83 Linköping, Sweden

Linköping 2001

ISBN 91-7373-212-5
ISSN 0345-7524

Printed by UniTryck, Linköping 2001

Abstract

Usability is one of the most important aspects of software. A multitude of methods and techniques intended to support the development of usable systems has been provided, but the impact on industrial software development has been limited. One of the reasons for this limited success is the gap between traditional academic theory generation and commercial practice. Another reason is the gap between usability engineering and established requirements engineering practice. This thesis is based on empirical research and puts a usability focus on three important aspects of requirements engineering: elicitation, specification and release planning.

There are two main themes of investigation. The first is concerned with the development and introduction of a usability-oriented method for elicitation and specification of requirements, with an explicit focus on utilizing the skills of technical communicators. This longitudinal, qualitative study, performed in an industrial setting in the first half of the nineties, provides ample evidence in favor of a closer collaboration between technical communicators and system developers. It also provides support for the benefits of a task-oriented approach to requirements elicitation. The results are also reflected upon in a retrospective paper, and the experiences point in the direction of an increased focus on the specification part, in order to bridge the gap between usability engineering and established requirements management practice.

The second represents a usability-oriented approach to understanding and supporting release planning in software product development. Release planning is an increasingly important part of requirements engineering, and it is complicated by intricate dependencies between requirements. A survey performed at five different companies gave an understanding of the nature and frequency of these interdependencies. This knowledge was then turned into the design and implementation of a support tool, with the purpose of provoking a deeper understanding of the release planning task. This was done through a series of cooperative evaluation sessions with release planning experts. The results indicate that, although the tool was considered useful by the experts, the initial understanding of the task was overly simplistic. As a result, a number of design implications are proposed.

Acknowledgements

The research and experiences reported herein spans almost a decade. During such a long time you meet many people, some of which you come to continuously depend on for your own achievements, and some of which may only drop a word or thought that acts like a spark to you. Both are in a way equally important.

Most of all I would like to thank my supervisors: Kristian Sandahl, for his encouragement, guidance, constructive criticism, and his astounding memory cherishing innumerable instructive anectdotes; Jonas Löwgren, who has been my friend and mentor for many years, for his invaluable guidance and ability to put the finger on the right spot, and at the right time; and Sture Hägglund, whose wisdom, encouragement and sometimes mysterious maneuvres have meant so much, not only to myself but to many students.

I am also indebted to Martin Rantzer who provided the possibility for me to do this under the best conceivable circumstances.

Several people have also generously contributed with their time and expertise, often under circumstances where time is by far the most limited resource (and consequently, someone asking for a half-day interview or so, could easily be seen as a nuisance, to put it mildly). I would like to express my deepest gratitude towards Thomas Hellström, Stefan Johansson and Patrik Wiman on the NAM team at Ericsson Radio Systems AB for their courtesy and generosity during almost three years.

Joachim Karlsson, Magnus Höglund, Anders Brodin, Mikael Johansson, Sture Ågren and Mikael Lindvall have all offered several hours of their time and expertise in requirements management.

I would also like to thank Ulrik Pettersson who is the true initiator of the RDEM model, for all inspiring discussions about market-driven requirements management; Björn Regnell, for fruitful collaboration and also for putting me on the right track; and Lars Hult and Stefan Holmlid for their help, their critical views and insatiable hunger for debates.

Finally, thank you Hannah for your endless love and support, and for giving me time to do this. And thank you Ylva and Björn for stealing some of it.

Pelle
Lugnet, November 2001

Table of Contents

Introduction ... 1

Research questions and methodological issues ... 3

Overview of contributions ... 12

Overview of papers ... 13

Related publications not included in this thesis ... 16

Issues for further research ... 17

Frame of reference ... 21

Usability Engineering ... 22

Requirements Engineering ... 40

UE and RE - A Comparison ... 65

References ... 79

Paper I: *Technical Communicators and System Developers Collaborating in Usability-Oriented Software Development: A Case Study* **... 95**

Paper II: *A Usability-Oriented Approach to Requirements Engineering* **... 115**

Paper III: *Dissemination of Usability: Failure of a Success Story* **... 135**

Paper IV: *Requirements Lifecycle Management and Release Planning in Market-Driven Requirements Engineering Processes* **... 153**

Paper V: *An Industrial Survey of Requirements Interdependencies in Software Release Planning* **... 167**

Paper VI: *Release Planning in Market-Driven Software development: Provoking an Understanding* **... 187**

Chapter 1 Introduction

This thesis is concerned with a range of issues within requirements engineering (RE). It spans from method development and deployment to a detailed description of a specific RE task. The driving force behind the research reported here is based in a strong ambition to bring usability to the software engineering practice. As a consequence, the research is also of a highly empirical and applied nature. The main contributions could be summarized as an increased understanding of both industrial uptake of usability-oriented requirements engineering, and release planning in market-driven software product development.

Thus, there are two main bodies of research reported, separated *but also* tied together by an interlude of reflection. The first is concerned with the development and introduction of a usability-oriented requirements engineering process. The second deals with issues specific to release planning in software product development.

The first study, in the following referred to as the Delta study, was initiated in the spring of 1992. There were two major driving forces behind the project: First of all usability had been identified as important but lacking by the Ericsson-owned software consultancy involved in the study. Second, technical communicators' well known and documented abilities to contribute to the understanding of end-users was to be put to better use. The objective was to develop a usability-oriented requirements elicitation and specification approach adapted to the needs and the culture of the consultancy, and to study relevant issues related to this project. Of specific interest was the increased collaboration between software developers and technical communicators. The study itself spanned more than two years full-time and was reported in a Licentiate thesis [21]. Some salient themes are summarized in Paper I & II.

Upon completion of the Delta study, I worked in the software industry for five years as a usability specialist, with a continuous mission to introduce usability-oriented methodology in a wide variety of organizations. An emerging wisdom from this period was that the missionary approaches employed were often inappropriate for the cultures in which this mission was carried out. In general, little consideration was shown for the fundamental components of industrial software development and its practitioners. The confrontation between usability engineering and software engineering was always problematic, and the point of collision was usually represented by the requirements. This is described in Paper III.

These lessons were taken into consideration in the development of another process at Ericsson; this time a requirements engineering process aimed at evolutionary development with frequent releases. The personal motive was thus to integrate usability considerations with requirements engineering practice in a better way. The resulting process, outlined in Paper IV, was then employed by a product team at Ericsson. While studying its practical use, a specific problem with this evolutionary approach was the task of selecting the best set of requirements for each release. Specifically, the interdependencies between the requirements were difficult to manage, and caused much frustration to the requirements managers. Thus, a survey was carried out to gain an initial understanding of the nature and frequency of such interdependencies, as reported in Paper V. Then, again in the strong ambition to bring usability to software engineering practice, a tool was designed and put in the hands of the release planning experts, in order to gain a deeper understanding of the task of the release planning. The tool and the knowledge provoked by its practical use is covered in Paper VI.

Thus, on the whole, the research was carried out in the industry, in a strong ambition to allow real problems to govern the course of study, as is often requested within software engineering (e.g., [132][46]). As a consequence, the researcher can be confident of the practical relevance of the issues studied. On the other hand, reality may not be as timely and coherent as one could wish for a thesis.

In the following sections, the explicit research questions are stated together with a discussion of the research methodology employed. This is followed by a declaration of the contributions claimed. Chapter 2 aims to explain and summarize important concepts within both Usability Engineering and Requirements Engineering, to provide a theoretical context for the papers, which then follow.

1.1 Research questions and methodological issues

Since the research questions, and consequently the methods employed to address them, differ significantly between the two main areas of study, they are given separate treatments in the next two sections. This is followed by a critical discussion on relevance and validity.

1.1.1 THE DELTA STUDY

In the early 1990s, the need for an increased focus on usability issues in industrial software development was widely acknowledged, and a multitude of methods and techniques had already been provided by the human-computer interaction (HCI) community. Even so, several authors reported a meagre up-take by the software industry (e.g., [58][2][56]). Although such reports provided valuable information as to the needs and beliefs of practitioners, the knowledge gained by such surveys was inevitably superficial.

Furthermore, technical communicators (TCs), which are specialists responsible for the development of documentation such as users' manuals to computer applications, have played a subordinate role [68] in software design, even in these usability-oriented methods. Considering that the TCs often have extensive knowledge and experience of the needs of the users and are frequently reported to contribute to usability [28][160][1], it appeared reasonable that a usability-oriented process would benefit from their institutionalized participation. But little was known as to what effects a closer collaboration between TCs and system developers (SDs) would have on the development process and the product.

The general research goal of the Delta project was to gain first-hand knowledge and experience as to how a usability-oriented method, including the competence of technical communicators, could be adapted and applied in a commercial environment, where the concept of usability had previously not been of prime importance. More specifically, the primary research questions were:

1. How is the usability concept perceived, understood and developed over time, by the different stakeholders, such as the developers, the TCs, the users, the management, etc.?
2. How are usability-oriented techniques received and appreciated by these different stakeholders?
3. How does a closer collaboration between TCs and SDs affect the environment, the process and the product? Is such a collaboration beneficial?
4. Are there any obstacles to this type of collaboration, and in that case, what are they?

The research questions and the aims of our work indicate a search for a global and wide-ranging portrayal of the actors and their environment, and several factors guided our choice of method. First of all, the intrinsic vagueness in the research questions clearly indicate a lack of in-depth knowledge, which in turn called for an *exploratory* study that could lead to more specific questions. Secondly, the questions are concerned with human behavior, which is significantly influenced by the setting in which it occurs; thus one must study that behavior in its own contexts to gain a thorough understanding. Delicate matters of this sort could only be captured by analysis of rich *qualitative* data. Thirdly, we saw a need for *longitudinal* study, in order to allow for the collection of alternative interpretations of events, and permitting the actors' own interpretations of these events to be collected as they occur [42]. The fact that Ericsson Infocom, the specific Ericsson company involved in the study, had previously not employed usability-oriented techniques implied that the new way of working would have an educational effect that would have been difficult to capture otherwise. Finally, previous research within HCI had only had a limited impact on the software industry, and for that reason we saw a need to take an *active* part in the introduction of the Delta extension. This made us concentrate on the *action research* paradigm (discussed below). For a more comprehensive treatment of the methodological issues of the Delta study, the reader is referred to [21].

Case research

Benbasat *et al.* define the term "case study" in the following way [3].

> A case study examines a phenomenon in its natural setting, employing multiple methods of data collection to gather information from one or a few entities (people, groups, or organizations). The boundaries of the phenomenon are not clearly evident at the outset of the research and no experimental control or manipulation is used.

Case research is useful in the study of "why" and "how" questions (rather than "how much" or "how often") because these deal with operational links to be traced over time. By means of qualitative data collection, a case study is capable of capturing complex relations between entities studied and between the entities and the context in which they reside.

The main strength of the case research approach is also its weak point. Since phenomena are studied within a context which influences, or interferes, with the entities under study, the observer must interpret and convey this context to the reader. If not, the report and the reported results will only have a curiosity value. Another source of distortion is the interference of the researchers, their mere presence, and the activities that they may engage in for the data collection. Obviously, the more they interfere with the ongoing activities, the less could be said about what the activities would have been like without their participation.

There is a spectrum of approaches to case research, along the dimension of how active, or intrusive, the researcher is. In one end of this spectrum we find the *participant observation* approach (see, e.g., [17]), which in turn could range from "the complete observer" to "the complete participant". The complete observer role is identified with eavesdropping and reconnaissance in which the researcher is removed from sustained interaction with the informant (ibid, p. 82). On the other hand, the complete participant conceals the observer role, and participates in the ongoing activities as a native member of the local society.

However, at the extreme end of the spectrum, we find *action research* ([69], see [24] for a critical examination). Here, the researcher is not an independent observer, but instead takes an active part in the ongoing processes, and contributes intentionally to the positive outcome of the activities. Thus, he participates in, and contributes to, a change in the community under investigation. As a consequence, he obtains an in-depth and first hand understanding of the process [3].

Data collection and analysis

With this background, the Delta project was carried out as an action research project at Ericsson Infocom. I participated in the development of the Delta method as one of three researchers. During the pilot project, where the Delta method was evaluated, I took part in the work as a member of the development team, under the same conditions as the rest of the team members. This gave very good access to the field that was studied. I worked just as if I had been a consultant to Infocom, with the same hours and the same locations. I kept a personal diary from all the meetings, formal as well as informal. Internal protocols, progress reports, and other project documents were also collected.

After the pilot project, I withdrew to analyze the data. The analysis took a great deal of time and the process was very similar to the "hermeneutic circle" that Ödman [127, p. 78] describes as an alternation between interpretation and understanding. Salient themes emerged slowly as data points were grouped and re-grouped. At some points, where further information was needed for the interpretation, additional data was collected by means of semi-structured interviews.

1.1.2 THE RELEASE PLANNING STUDY

A consequence of the lessons learned from working with the Delta method over several years was an increased focus on integrating, rather than adding, usability issues to the requirements engineering process. The study of an RE process designed to accommodate this shift of perspective (RDEM, covered in Paper IV) gave rise to the issue of release planning.

Release planning gains importance as modularization techniques and increasing standardization drives the software industry from custom software towards software products, so called Commercial Off-The-

Shelf software. From a requirements engineering standpoint this poses new questions. From the purchaser's perspective one of the major questions is: How do we elicit and verify requirements on the third-party software? From the vendor's perspective the crucial problem is how to make sure that the product meets the demands of the market at the right time, i.e., how do we plan our product releases? Our focus is on the latter of these two.

The research issues appeared in the context of a product development effort within Ericsson Radio Systems in Linköping in 1999. The product, called NAM as in Number Analysis Manager, is developed by means of an adaptation of the RDEM process. The product could be described as a technical support tool for tracking down inconsistencies and other problems in the very large tables of numbers and routing information in a telephone exchange. The tool supports a variety of mobile telephony standards, each having a number of unique features. NAM is also provided for a variety of PC platforms, as well as for the Unix environment. Furthermore, it is sold both as a separate product, and provided as a module in a larger operation and maintenance system for mobile telephony. As a result, the product needs to be kept aligned with the continuous changes in telephony standards *and* operating systems *and* the surrounding O&M system *in addition to* the requests for new features by the separate customers. Thus release planning is a continuous issue.

In its simplest guise, the task of release planning could be described as one of prioritizing and resource-estimating the requirements–two tasks that are by themselves far from trivial–and then selecting requirements from the priority list until the estimates equal the available resources. However, in reality requirements are interwoven by interdependencies into a complex web, which further complicates the task greatly. In the case of NAM, although a state-of-the-art requirements management system was employed, this did not offer any support for such interdependencies. Thus, the initial question at hand was:

- How can the task of release planning, with explicit consideration of requirements interdependencies, be supported?

Designing support for complex human activity demands in-depth knowledge of the tasks at hand, not least from a usability engineering standpoint.

We also needed to understand the nature of the dependencies and how various types of dependencies affect release planning. The general problem has been identified previously (e.g., [92]), but former studies of requirements interdependencies have focused on issues that are not directly relevant for release planning purposes (see *Requirements interdependencies* on page 62). Thus we needed to answer the following questions first:

1. What kind of interdependencies exist between requirements, that are related to the task of release planning?
2. How frequent are such interdependencies?
3. Is it feasible to identify and manage such interdependencies in a commercial setting?

Questions like these could be addressed in a number of ways. A questionnaire-based approach would potentially result in a material allowing statistical analyses adequate for question number 2. On the other hand, gaining insight in the very nature of interdependencies and the effect that they have on release planning requires a more qualitative approach. Furthermore, it would be impossible to plan the data collection for question 2, unless question 1 was first answered.

By contrast, a thorough case study, as in the Delta situation, would have provided a deep insight in the qualitative issues for a single case such as NAM. The general question of how release planning can be supported, as stated above, begs for a qualitative study where several different approaches to the problem are evaluated. Although this would have been valuable indeed, commercial pressure did not permit that kind of investment on the product team's part. Thus, a survey approach was more feasible. Our study involved five different companies with products in different domains, and answered the first three questions. The reader is referred to Paper V for a detailed description of its design and accomplishment.

To understand the context of release planning, I followed the NAM product team for almost a year. This included attending meetings, conducting semi-structured interviews with the product team members, and as the focus narrowed, discussing specific issues concerning interdependencies and tool support. As I learned more about release planning, it became clear that traditional task analysis techniques and notations would not be able to capture the subtle judgements, the *fingerspitzengefühl* of the product managers. Thus, partly as a proof of concept, that a release planning support tool can handle interdependencies, but primarily as a way of provoking a dialog between the product managers and myself, I designed and implemented a computer based tool for supporting release planning. Hence using the provotype, as it was called, as an instrument in the search for a richer understanding I aimed at answering the last research question:

4. What is the nature of the release planning task, and how can it be supported?

In this case the data was collected by means of video-recording a series of formative evaluations of the provotype, together with the domain experts. This is described in the last paper, Paper VI.

1.1.3 RELEVANCE, VALIDITY AND SCOPE

Since the mid-1990s, there is an ongoing self-criticism within software engineering research. There are two main critiques: much research don't focus on real problems in real contexts, and many of the claims made about methods and tools are not supported by the studies conducted to validate them. As an example of the first, Potts [132] state that

> The problem stems from treating research and the involvement of industry in applying the research as separate, sequential activities. This phased approach to software engineering, which I call "research-then-transfer" leads to laboratory research that often fails to address significant problems.

He then goes on to point out that software engineering far too often is done at a superficial level, and that findings from surveys are rarely followed up. He continues:

> Instead of being inspired to conduct more detailed studies, most researchers are content to continue their laboratory research and cite general conclusions from the study that bolsters their intuitions.

This is further emphasized by Glass, who in an ironic paper [46] hopes that software engineering researchers will eventually give up their "arrogant and narrow" approach.

The second type of criticism is based on the fact that far too many studies of new tools or methods could be summarized as "I tried it, and I like it". Tichy et al [158] found that 50% of articles about new designs and models within Software Engineering completely lack thorough evaluation, and state that this situation is alarming. Zelkowitz and Wallace [169] showed that the by far most common validation methods used by researchers, as reported in prominent software engineering literature, could be characterized as "assertion". The same kind of argument is put forward by Fenton *et al.* [40], in their critical article about the status quo of Software Engineering research.

As a contributor to this field, it is important to consider this criticism, and examine any claims accordingly. There should be little doubt that the research reported herein is *relevant*. The very nature of applied and empirical research escapes much of the criticism by Potts and other critics, and indeed the present studies focus on real problems, identified and studied in real contexts.

Validity

The second issue is whether the claims are valid. In general, all studies reported here are based on empirical data, and performed in real situations. As an industrial Ph.D student I have been privileged with good access to people and data. My formal training in computer science and software engineering has helped me to understand the context in which I have performed the studies, but it has also influenced my view on usability issues, which could be characterized as an engineering perspective. This is fairly evident from the papers but needs to be pointed out (cf. [116]).

A related validity issue is the fact that I have been employed in part by one of the companies involved in the studies. Although it should be evident that the results herein are not of a kind that would be controversial from a business strategy point of view, it is important to point out that apart from the conventional anonymization policy, I have not been asked to conceal any information or data whatsoever. On the contrary, I am grateful for the great courtesy and openness I have met.

The first theme of investigation, the Delta study, was performed a few years ago, and consequently the contributions need to be seen in this light, but the question of current validity is nevertheless relevant. In terms of scientific validity there is, to the best of my knowledge, no more recent study reported with the focus and depth of the Delta study. The question of *practical* validity depends on the context in which it is seen. There are organizations which have had technical communicators on the development team as part of a usability strategy for a long time now. But there are certainly organizations that still find themselves on the same level as described in the study. For these, the results are still valid.

As for the release planning theme, which ends with a study of a new tool, we took notice of the criticism that Tichy et al., Zelkowitz and Wallace and others put forward. We chose to take a smaller, but steadier, step forward in the belief that small but steady steps will benefit the software engineering community more than large and staggering. Thus, again, the focus is on understanding and describing the problem, rather than assessing the utility of the tool.[1]

Scope

Although the study of real data in real situations is put forward as imperative within applied software engineering research by e.g. Fenton *et al.* [40], this also implies a strong coupling between the results and the context of the study. In general, the positivistic sciences' common notion of generalizability of results does not apply to this kind of research. Instead, one speaks of “transferability” or “exportability”, referring to the possibility for the readers to understand the results reported and adapt them to their own contexts.

As the predominant context of study could be characterized as an engineering culture, some of the conclusions drawn may not apply to other contexts. This is worth noting especially for the issues regarding systems development methodology development and introduction. For example, in Paper III it is argued that an increased focus on requirements would act as a catalyst for the dissemination of usability awareness. This is probably not true for more dynamic and creative environments, such as a small computer game vendor.

1. We found much inspiration for this in the thought-provoking book *Software Requirements and Specifications*, by Jackson [82].

1.2 Overview of contributions

In the research reported herein, we have contributed to the understanding of:

- Usability-oriented RE methods development and deployment and
- its (limited) uptake in industrial practice.
- The role of technical communicators in systems development.
- Connections between Usability Engineering and Requirements Engineering, and how they can benefit from each other.
- Requirements interdependencies.
- Release planning in market-driven software product development.

We have also proposed an approach to integrating usability-oriented techniques with systems development in a way that may avoid many of the problems usability experts face in industrial systems development. This approach also has some novel characteristics in that it is attribute driven (explained in Paper IV), and thus acknowledges the perspective that requirements engineering is a learning process more than anything else.

We have suggested a set of techniques for identifying and managing requirements interdependencies in a cost-effective manner.

Finally, as a by-product, we have designed, implemented and evaluated a support tool which in itself serves as a proof of concept, in that it encompasses not only some of the lessons learned about release planning, but also a selection algorithm which can handle interdependencies.

In terms of *practical* contributions, we have developed and deployed two methods, one of which has been widely used and has in itself contributed to an increased focus on usability issues, and on the benefits of a close collaboration between technical communicators and system developers.

The specifics of these contributions are distributed over a number of papers, which are outlined next.

1.3 Overview of Papers

Paper I: Technical Communicators and System Developers Collaborating in Usability-Oriented Systems Development: A Case Study

Pär Carlshamre

In *Proc. 12th ACM Annual International Conference on Systems Documentation (SIGDOC'94), pp. 200-207, 1994.*

The first paper presents the Delta study from the perspective of collaboration between technical communicators and system developers. It describes the study itself and some of the main themes that emerged. It provides evidence of the benefits of a close collaboration between the two competencies, and also describes some of the demands that such a collaboration puts on the environment. Specifically we argue that the usability of the developed product gained from the mixed perspective; the effort needed to produce user documentation decreased; but that a successful collaboration demands special care from an organizational point of view. We also identify a need for tools that support, rather than hinder, a close collaboration.

Paper II: A Usability-Oriented Approach to Requirements Engineering

Pär Carlshamre and Joachim Karlsson

In *Proc. 2nd IEEE International Conference on Requirements Engineering (ICRE´96), pp. 145-152, 1996.*

The second paper outlines the Delta method, and represents a more marked step towards requirements engineering. It also shows that this step was a very natural one from the Delta experience. The requirements specification provided initially in the pilot project (where the Delta method was evaluated) only had a few statements on a very high level, and the usability-oriented and task-based approach drove the elicitation of both functional and non-functional requirements. In this paper we claim that this is an effective approach to the early stages of requirements engineering.

This paper is based solely on the Delta study, but aligned to an requirements engineering audience, to which Dr. Karlsson can be said to belong. This reflects the division of responsibility and labor between the two authors.

Paper III: Dissemination of Usability: Failure of a success story

Pär Carlshamre and Martin Rantzer

ACM interactions, 8(1):31–41, 2000.

This is an experience paper, collecting salient themes and conclusions from many years of working with usability-oriented methodology deployment in various organizations. It reflects upon the common, and sometimes misleading way of describing usability dissemination and maturity on a depth scale, and argues that breadth is at least as important. We suggest that usability engineers and the mission to disseminate usability could gain from an increased understanding of and humility towards software engineering practice and its underpinnings. We also provide a number of normative conclusions which have formed some of the basis for the RDEM process described in the next paper.

The thoughts presented in this paper are mostly my own, although shared and inspired by Mr. Rantzer, who has worked with the Delta method for as long as myself, and has also been responsible manager of Delta within Ericsson. I also wrote the paper.

Paper IV: Requirements Lifecycle Management and Release Planning in Market-driven Requirements Engineering Processes

Pär Carlshamre and Björn Regnell

Proc. IEEE Int. Workshop on the Requirements Engineering Process. In Proc. 11th Int. Conf. on Data-base and Expert Systems Application (DEXA2000), 2000.

Paper IV describes the Requirements-Driven Evolutionary Model of Ericsson Radio Systems AB as contrasted by the REPEAT process of Telelogic AB. These are two approaches to managing requirements in a market-driven software development situation. The two processes are compared and similarities and discrepancies analyzed. The most important common quality is also the main contribution of the paper, namely the attribute-driven state-based approach to requirements management, which is well aligned with the view of systems development as a learning process. It is argued that this approach provides a natural environment for including usability issues *en par* with other quality attributes. A number of issues for further research are identified, including supporting release planning in general, and managing interdependencies between requirements in particular.

This paper grew out of several discussions between Dr. Regnell and myself. Dr. Regnell was involved with the development of REPEAT in the same way as I was involved in RDEM. There is an equal focus on both methods, and both authors also contributed equally in terms of writing.

Paper V: An Industrial Survey of Requirements Interdependencies in Software Release Planning

Pär Carlshamre, Kristian Sandahl, Mikael Lindvall, Björn Regnell, Johan Natt och Dag

In Proc. Fifth IEEE Int. Symposium on Requirements Engineering (RE'01), pp. 84–91, 2001.

This paper describes a study of interdependencies among requirements in five independent requirements repositories at five different companies. The results provide evidence for the importance of identifying and managing interdependencies. Among the results are that about 80% of the requirements in the material had some relationship relevant for release planning with another requirement; that interdependencies can be related to either value or function, and that the type of interdependencies varies between development situations. The use of graphs to show interdependencies is also described and reported to be beneficial for supporting reasoning about interdependencies.

In addition to important comments on early versions of the paper, Dr. Sandahl contributed with the idea of release coupling, and also helped me with some of the formalisms in the paper. Dr. Lindvall provided data from one of his projects as well as critical comments. Dr. Regnell provided substantial suggestions for improvements underway, and Mr. Natt och Dag performed the lexical analysis included in the paper. Beyond that, I designed the study, performed it, did most of the analysis and wrote the paper.

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

Pär Carlshamre

Submitted to *Requirements Engineering Journal, Springer.*

The final paper is based on the evaluation of a tool for supporting release planning, which takes not only priorities and estimates into consideration, but also interdependencies. The main focus is on understanding release planning, and the tool was designed and implemented as part of the instrumentation for the study. By means of a cooperative evaluation of the so called provotype, with real data and in three different industrial settings, we were able to collect rich data about the nature of the task. Release planning is found to be a wicked problem, which has several important implications for the design of supportive tools. The evaluation shows that our initial understanding of the task was overly simplistic, and thus the provotype 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 is proposed.

1.4 Related publications not included in this thesis

CARLSHAMRE, P., LÖWGREN, J., AND RANTZER, M. Usability Meets the Real World: A Case Study of Usability-Oriented Method Development in Industrial Software Production. In *Human Factors in Organizational Design and Management - IV, Proc. 4th Int. Symp. Human Factors in Organization Design and Management* (1994), G. E. Bradley and H. W. Hendrick, Eds., NORTH-HOLLAND, pp. 427–432

This was the first paper on the Delta study, and focuses more on the research process than the actual results. Since action research was not considered as an established research approach within computer science at the time, the process itself was important to discuss.

CARLSHAMRE, P. *A Collaborative Approach to Usability Engineering: Technical Communicators and System Developers in Usability-Oriented Development*. Licentiate thesis no. 455, Linköping University, S-581 83 Sweden, 1994.

This is the full Licentiate thesis on the Delta study.

CARLSHAMRE, P., AND TUMMINELLO, J. Technical communicators' current views on usability and collaboration. In *ACM SIGDOC'95 Conference Proceedings* (New York, 1995), ACM Press, pp. 11–19.

Since the Delta study was a single case, it was interesting to see how well the description of the technical communicators' situation applied to a wider context. This is a small cross-cultural interview study performed in collaboration with a technical communicator in the US to follow up the results.

MÅRDSJÖ, K., AND CARLSHAMRE, P. *Retoriken kring tekniken. [Rhetorics and technology]*. Studentlitteratur, Lund, 2000.

This book is primarily aimed at courses in technical writing and information design. Dr. Mårdsjö was involved in the original Delta study, and much of the content builds upon the experiences from that study and aims to relate them in (and to) a wider social and rhetorical context.

NATT OCH DAG, J., REGNELL, B., P., C., ANDERSSON, M., AND KARLSSON, J. Evaluating automated support for requirements similarity analysis in market-driven development. In *Proc. Seventh Int. Workshop on Requirements Engineering (REFSQ2001)* (Interlaken, Switzerland, 2001).

This paper is a full description of the study performed to investigate whether lexical similarity analysis of requirement slogans represents a feasible way of identifying possible interdependencies between requirements. My own contribution to this paper amounts to a minor section on interdependencies.

1.5 Issues for further research

An important product of research is also to point out the questions that remain unanswered, which provides opportunity for relevant further studies. There are apparent follow-ups and extensions of the studies reported herein, such as relating some of the experiences reported in Paper III to established organizational theories; evaluating the Delta method with respect to a formal framework (rather than its effects); or performing a larger study of interdependencies to gain more statistical significance. But there are also a few areas which stand out as more interesting for further studies. These are outlined in the following.

The Extended Interface Management System. One of the outcomes of the Delta study was the redefinition of the *extended interface*, as an acknowledgement of the fact that there is no definite borderline between "GUI" and "documentation". This division is traditional although it lacks rational grounds, but more importantly it serves as a severe barrier between technical communicators and GUI developers. Instead, we propose a division of interface objects into the two layers of "design" and "realization". This, in turn, provides a common arena for technical communicators and system developers to collaborate at the design level, where they can contribute equally. At the realization level, however, they can utilize their *different* skills.

A so called Extended Interface Management System (EIMS) was outlined in [21], where the fundamental data object is a task which, together with design rationale, embrace the design of one or several interface objects. These designs may then be realized either in terms of user documentation or graphical components, based on the informed decision by technical communicators and GUI developers in collaboration. The EIMS also supports several aspects of the usability engineering life cycle, such as fast iteration (version handling), management of usability test results (tied to the tasks), and management of several representations of the same task/design. Although the basic technologies behind the EIMS have been available for some time (see [21]), we have yet to see the synthesis of these ideas materialize. On the tool side, many current requirements management systems are capable of storing various representations of particular requirements. On the method side a few related approaches, combining various representations, such as scenarios, design rationale and concept demonstrators have been reported (e.g., [153][154]). However, in general these two sides have not yet met, and specifically, neither of the sides tends to have have an extended interface perspective. The development and study of EIMS would be based on the hypothesis that it would radically change important aspects of work practice and division of responsibilities within industrial software development.

Integrating Usability. The RDEM process provides an opportunity to integrate usability issues by incorporating usability-related attributes in the requirements attribute structure. Paper III reveals some bold hypotheses behind this approach:

- It will support a wide (rather than deep) dissemination of usability awareness.
- It will emphasize the importance of usability, but not at the expense of other quality attributes, as is sometimes the case in current practice.
- It will acknowledge the expertise of the system developers, and provide a problem-based approach to learning about usability issues, rather than the traditional way of “taking a course”. This allows the experts to use their familiar techniques (their expertise) to a greater extent.
- It will provide a way of associating non-functional qualities to functional requirements by means of a common derivation from a task description.
- It will avoid the problem of methodology exhaustion, in that it does not represent yet another monolithic method.
- The attribute structure may be used as a closed information space, providing means for measuring progress during development.

Although RDEM, to various extents, has been in use for some time now, none of these hypotheses have been tested. The opportunities for further studies are obvious.

Release Planning Tool. This thesis ends with a number of conclusions regarding a prospective tool for supporting release planning. It is argued that such a tool would need to:

- be highly interactive, direct manipulative and in this way support exploration.
- support several different models of requirements and releases, such as time lines, arbitrary sets, “nodes and edges” and hiearchies.
- support the planning of several consecutive releases.
- support comparison of release suggestions to a great extent.
- manage interdependencies by strength rather than by type.
- keep a low profile and act “humbly” towards the planning expert.

These conclusions need also be seen as hypotheses to be tested by future research.

Chapter 2
Frame of reference

This thesis is concerned with two areas of interest within computer science. The boundary line between them is indeed indistinct, and for those working in both areas probably even more so than for people professing themselves to either side. This chapter covers some of the theoretical background to both these areas. The aim has been to treat them first as separate areas of concern, avoiding mixed perspectives and terminology, and then to investigate what they have in common and what they might contribute to each other.

Since this chapter is also aimed at describing and explaining salient issues within usability engineering to requirements engineers and vice versa, it contains some basal information. Thus, the usability engineer may want to skip sections 2.1.1 and 2.1.2, and the requirements engineer may find sections 2.2.1 and 2.2.2 rather tedious.

We first consider Usability Engineering (UE), starting out with a discussion of basic definitions and then moving on to methodology issues. The aim is to give a context for the early papers on the Delta study, and to describe the state of knowledge and practice as it were. This section ends with a few remarks on the methodology developments since that time. Then, a similar treatment of Requirements Engineering is offered, beginning with an overview of foundational concepts, but then with a narrowed focus on specific issues relevant for the later papers.

2.1 Usability Engineering

The study of human-computer interaction (HCI) is a multi-faceted activity, bringing together experts and researchers from such seemingly disparate areas as linguistics, psychology, ergonomy and engineering. What brings them together is a common interest to contribute to the development of more usable computer systems. The term "multidisciplinary research" indeed suits as an epithet of the HCI field, which is one of the characteristics that make the subject so interesting. As the name itself implies, it includes the studies of humans, computers, and how these two parties interact, all of which are areas rich enough to carry their own large research communities. Although there is still no universally established definition of HCI, the ACM SIGCHI provides a good working definition [73]:

> Human-computer interaction is a discipline concerned with the design, evaluation and implementation of interactive computing systems for human use and with the study of major phenomena surrounding them.

The term itself was adopted in the mid-1980s as a means of describing a new and wider field of study than the psychological experiments in human information processing aspects that were typical for the 1970s and the early 1980s. At the same time, there was a veritable explosion in the methodology area of HCI, and methods based on various values, beliefs and assumptions saw the light in the increasing number of HCI publications and conferences. This is also where Usability Engineering was conceived, which we shall return to later.

By then, the prevailing view of the interaction between humans and computers was that of a single user using a single computer to solve a certain task. This is also evident from most of the definitions cited in the next few sections. However, with increased computing and communication power, allowing multitasking, multimedia and heavy graphics, it was soon recognized that HCI would have to embrace a vastly larger area of study. This included group working, integration of media, the social impact of the emerging technology etc. In recent years, with the web explosion, the field has expanded even more, with an increased focus on e.g., web design, information retrieval and animation.

The Delta project adopted the foundational values of the Usability Engineering paradigm, which is consequentially given a fairly thorough treatment below. The first couple of sections, however, discuss some of the basic questions involved, namely what a user interface is, what user documentation is, and, perhaps most importantly, what usability is.

2.1.1 USER INTERFACE AND USER DOCUMENTATION

The interpretation and use of the terms "user interface" and "user documentation", which naturally are two central concepts in HCI, varies considerably in the literature, and the definitions are indeed vague. According to the traditional view, hard-copy and online help facilities are grouped under the name "user documentation", whereas windows, buttons, menus and I/O devices constitute the "user interface".

This traditional separation may at a first glance seem conceptually quite appealing but a closer investigation reveals the opposite. As Grice [64] predicted, "the lines between hardware, software and information are getting blurred with the advent of interactive programming, firmware, new input devices, and displayable manuals." Our own view within the Delta project group, was that "user documentation" in some way should be considered as a part of the "user interface".

Most authors within the HCI discipline apparently share this view, and many of the major HCI textbooks cover user documentation as a part of the user interface (see, e.g., [145][134][36]. Allwood [1] devotes a substantial part of his book to user documentation and help facilities). This view is also reflected in their and many others' recommendations as to how the user interface should be developed. The prevailing opinion seems to be that user documentation is important and should be developed by specialists. In addition, cooperating with the technical communicators facilitates *their* job, and ensures a higher quality of the documentation[2].

This view of the user interface implies that the author of user documentation should be a specialist, and *for the benefit of the user documentation,* system developers should communicate with these authors. This did not correspond to our intentions within the Delta project, since one of the main ideas was to use the special skills of the technical communicator *for the benefit of the complete user interface*.

2. One of the major problems that technical communicators experience is to get information from the system developers [25].

A slightly different term, which is not marred by the ambiguity and confusion of the traditional "user interface", is introduced by Lewis and Rieman in their excellent freeware book "Task-Centered User Interface Design" [102]:

> The "extended interface" goes beyond the system's basic controls and feedback to include manuals, on-line help, training packages and customer support.

One emerging theme from the Delta study was the need for a redefinition of the extended interface to provide a common arena for technical communicators and system developers. As a result, it was stated that

> the extended interface comprises the system services and the system aids

where both *system services* and *system aids* were divided into their design and their realization. This provided a continuum between external (user interface) design and internal design (rather than between user documentation and user interface) in which the technical communicators could contribute on equal terms with the system developers depending on their interests and skills. The full discussion of the background and implications of this redefinition is given elsewhere [21, chapter 6].

2.1.2 USABILITY

Another very central term in this thesis is "usability". As most authors point out before they give their own definition of usability, there are probably as many definitions as there are authors on the subject. Spencer [150], who provides a thorough collection of definitions related to HCI, quotes Webster's Ninth New Collegiate Dictionary[3]: "[Usability means]...convenient and practical for use...", which in some sense expresses the intuitive meaning that most of us would agree upon. There are several other definitions, though.

The International Standards Organization states that [80, definition 3.1]

3. Webster's Ninth New Collegiate Dictionary, © 1983 by Merriam-Webster Inc., publishers of the Merriam-Webster® Dictionaries.

> [Usability is the] extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use.

Given a closer look, this definition could in fact be viewed as normative: "Specified users" implies that we need to learn who the potential users are, so that we can specify their characteristics. "Specified goals [...] in a specified context of use" implies that not only the specific tasks that are to be supported by the system, but also the environment in which these tasks are to be performed, must be studied and described. The rest of the definition focuses on the computer system itself, and gives an idea of how usability could be operationalized and measured. These activities will be elaborated on in *Setting goals and specifying usability* on page 33. Note also that usability is not a property of the system itself, but a quality of the *interaction* between users and products. This is important to note, since most other quality attributes specified in a requirements specification tend to describe some desired property of the product.

It may be relevant to address some related terms as well. The exact meaning of "usefulness", for example, has been a source for some debate (cf. [66][53]). Nielsen is very clear on this point, however, stating that "*utility* is the question of whether the *functionality* of the system in principle can do what is needed, *usability* is the question of how well users *can use* that functionality." [123, p. 25 my italics] Nielsen makes a clear distinction between usability and utility, which together constitute what he calls "usefulness". Thus, the definition used throughout the Delta project would in Nielsen's terminology be a definition of usefulness. Nevertheless, it translates more easily to Swedish than does the ISO definition[4], and reflects our view of these matters [104]:

> Usability is a result of *relevance*, *efficiency*, *learnability* and *attitude*.

4. Swedes have problems to discriminate between "efficiency" and "effectiveness" used in the ISO definition of usability. Furthermore, the ISO had at the time of the Delta project not agreed on a definition, even if there are proposed definitions cited in the literature as early as 1989 [14].

Where

- a system is *relevant* to the extent that it serves the users' needs, that is, it should support the "right" tasks.
- It is *efficient* to the extent that the users can carry out those "right" tasks efficiently, that is, at some required level of performance.
- It is *learnable* to the extent that 1) it is easy to learn for initial use and 2) the users remember their skills over time. Hence both short term and long term learnability is considered.
- The usability is also dependent on the users' *attitudes* towards the system.

In this definition most of the attributes described previously are accounted for. The relevance criterion says something about the system's *utility*, or *effectiveness*, as opposed to the efficiency criterion. The *attitude* attribute, which is one of the most important factors of usability (see, e.g., [75]) is included.

2.1.3 USABILITY ENGINEERING METHODOLOGY

Even though the HCI community has been able to agree to some extent on the definition of what constitutes usability, there is no generally accepted recipe as to how a usable system should be developed. One of the issues that different communities tend to agree on, though, is that the user should be involved in some way. To which extent the users should be involved is, however, largely a matter for debate. User involvement is nevertheless a suitable dimension to describe different approaches to usability-oriented development, in order to provide some context for the Usability Engineering school. As simplistically illustrated in Figure 1, the amount of user involvement spans the whole range from none to plenty.

Theory-based approaches, in which users do not participate directly, focus on the general (average) characteristics of individual users interacting with computers. Experiments were carried out in the early days of HCI, to investigate for instance how many items a menu should contain (cf. [147]), how menus should be designed in terms of breadth versus length, what the optimal speed is for a cursor to move on a VDU [57] etc. Although such experiments have been argued to render valuable results [67], some of which have found their way in to general user interface design guidelines (e.g., [148]) and tools for user interface evaluation [108], in retrospect their contributions seem to have been limited. Guidelines, constituting a way to practice HCI without direct involvement of users, have been criticized for their generality, making them difficult to apply in real-world projects (see [39] for an excellent overview). .

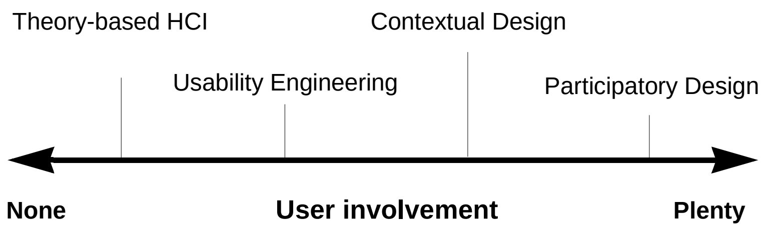

Figure 1: Different development approaches to enhance usability, ordered by the degree of user involvement

To the right of UE on our scale we find approaches involving users to a greater extent. Among these are Contextual Design, which relies heavily on *studies* of the users and their workplaces, and which is addressed to some extent in *Towards a wider context* on page 74. At the extreme right in the figure, we find Participatory Design (PD), which relies on *cooperation* with the users. Having its roots in the Scandinavian countries, PD is often called the “Scandinavian approach”. PD occurred first in Norway, in the beginning of the seventies, and has attracted much attention not only within the Scandinavian countries, but also at the American continent. The PD philosophy is based on political and ideological grounds, which can be illustrated by a few quotes from two of the most prominent proponents of PD [62, p. 1]:

- “Computer systems that are created for the workplace need to be designed with *full participation* from the users. Full participation, of course, requires training and active cooperation, not just token representation in meetings or committees.”
- “When computer systems are brought into a workplace, they should *enhance* workplace skills rather than degrade or rationalize

them. Enhancing skills means paying attention to things that are often left out of the formal specifications, like respect for tacit knowledge, building on shared knowledge and, most importantly, communication."

- "Computer systems are tools, and need to be designed to be under the control of the people using them. They should support work activities, not make them more rigid or operationalized."
- "The design process is a political one and includes *conflicts* at almost every step of the way. Managers who order the system may be at odds with workers who are going to use it. Different groups of users will need different things from the system and system designers will often represent their own interests. Conflicts are inherent in the process. If they are pushed to the side or ignored in the rush to come up with an immediately workable solution, that system may be dramatically less useful and continue to create problems."

Evidently, the PD philosophy shares some of the objectives with the previously described methods, but the impetus comes from somewhat different views. Whereas the usability engineering approach seeks to maximize return on investment for both the users, the developers and their respective organizations (see below), participatory design emphasizes the empowerment of the workers as one of the foremost goals [26][5]. Since the techniques used are powerful in eliciting and understanding needs of the users, and due to the problems of tacit knowledge, PD has gained some popularity also within Requirements Engineering.

Since the Delta method was conceived in an environment of industrial software development, with strong focus on commercial issues, it came to adopt the usability engineering approach. Hence, for the purposes of this thesis it suffices to describe usability engineering in some detail.

5. For this reason, advocates of the PD *techniques* sometimes feel that they need to make a political statement (e.g., [94]).

Commercial ground

For the advocates of usability engineering, user involvement *per se* has no intrinsic value, as opposed to the PD philosophy for example. The focus lies not on user participation, but on a *specified* level of usability, i. e., the product should not necessarily be as usable as possible, but rather usable *enough* to fulfill its usability specification. Of course, user involvement is crucial for the successful outcome of a project, and few usability engineers would even consider developing a computer system without some form of user participation. Nevertheless, this is an important distinction between the different approaches.

UE seems to hold a unique position, due to its explicit focus on cost-benefit issues. In industrial systems development not much work is done without a specification that acts as a contract between the customer and the developing organization. The development process is organized around the refinement and realization of specifications. Unless usability is specified chances are small that usability issues will be taken into consideration. Accounting for the usability of a product is proven and widely accepted as being profitable (e.g., [88][6]), but the message, to the extent that it has reached the industry, has not always convinced the intended recipients [58][55][56]. Despite this, usability engineering is clearly designed to meet the demands of the specification-based development dominating in industry [23]. The method includes several characteristics that are attractive for the commercial environment: the usability specification, the controlled way of incorporating users in the development life-cycle and the prevailing cost-benefit way of thinking.

The term “usability engineering” appeared in the mid-eighties as a collective name of a set of method steps. As the term implies, it is seen as an engineering activity. Good *et al.* [52] describe the characteristics of this approach very sententiously:

> Usability engineering is a process, grounded in classical engineering, which amounts to specifying, quantitatively and in advance, what characteristics and in what amounts the final product to be engineered is to have. This process is followed by actually building the product, and demonstrating that it does indeed have the planned-for characteristics. Engineering is not the process of building a perfect system with infinite resources. Rather, engineering is the process of economically building a

> working system that fulfills a need. Without measurable usability specifications, there is no way to determine the usability needs of a product, or to measure whether or not the finished product fulfills those needs. If we cannot measure usability, we cannot have a usability engineering.

Thus, usability, as any other aspect of product development, is seen as something which should be financially justifiable. Good *et al.* continue to describe usability engineering as consisting of the following steps:

- defining usability goals through metrics
- setting planned levels of usability that need to be achieved
- analyzing the impact of possible design solutions
- incorporating user-derived feedback in product design
- iterating the development until the planned usability levels are achieved

The first three steps as presented here are what characterizes this approach as an engineering discipline. However, to be able to specify proper usability goals, the characteristics of the user and the tasks must be understood. This is accounted for by means of some sort of user and task analysis. Furthermore, to be able to assess whether the planned levels of usability are achieved or not, some sort of usability testing must take place. These activities, along with the rest of the steps in usability engineering, will be elaborated on in the next few sections. Figure 2 gives an overview of the steps.

User profiling

One of the fundamental rules concerning the development of usable systems is “Know the user”[6]. As Thomas and Kellogg put it [157]:

A consideration of how “people really are” is just as necessary to the development of excellent systems as is consideration of how “the disk drive really works” or how “memory is really addressed” or how “machine logic really is”. It is clearly silly, for example, to develop an operating sys-

6. This expression has been attributed to Wilfred J. Hansen, who in 1971 presented four “user engineering principles”, of which “Know the user” is the first [71].

tem while ignoring how memory addressing occurs and building it instead on the basis of how you would have preferred the hardware to work."

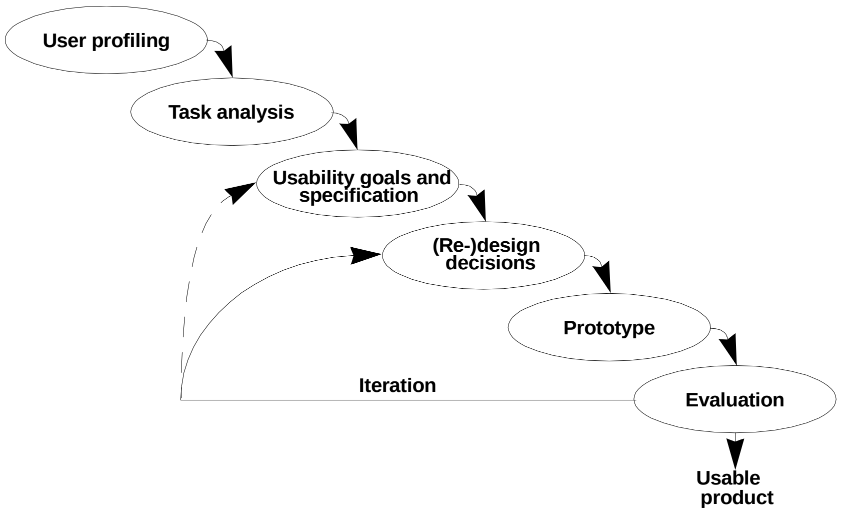

Figure 2: Overview of the usability engineering activities.

Unfortunately, there is little advice given as to how this "getting-to-knowing" should be done effectively. Not even some of the most comprehensive articles and textbooks in the HCI area (e.g., [55][134][145]) offer normative descriptions of how to get to "know the user".[7] As Nielsen says, "No universal solutions are available, except to recommend an explicit effort to get direct access to representative users and not be satisfied with indirect access and hearsay" [123, p. 74].

However, the type of information gathered from the user profiling should include aspects such as age, computer experience, work experience, educational level and particular problems or disabilities affecting the use of computers. Although Nielsen (ibid.) offers a couple of examples of how this information may guide the design, most of the interpretation is left to the experience and skill of the designer. So for the

7. The participatory approaches described earlier focus on these issues explicitly, and their advocates suggest that there in fact *is* no way to get to know the user without actually working together.

inexperienced designer (and perhaps even for the experienced one), questions like “What implications does it have that the user population has an average age of 62?” or “What if 89% are women?” remain unanswered.

Task analysis

To be able to effectively support the “right” tasks with the prospective system, the developers must naturally have a thorough understanding of those tasks. The activities carried out to capture this knowledge are collectively termed “task analysis”.

There is a bewildering number of approaches (see, e.g., [134] for an overview) to understanding and describing users’ tasks on various levels. The techniques could be said to fall in three categories: cognitive, procedural or organizational task analysis. All three levels need to be considered in order to accomplish usable systems. Cognitively oriented task analysis techniques, such as GOMS [20], are usually based on theoretical models of problem solving and human cognitive (and motor) abilities. They seek to model mental representations and processing for the purpose of designing tasks that can be undertaken more effectively by humans. Procedural task analysis aims more at an accurate description of the steps required to complete a task, often by decomposing tasks into sub-tasks and thus creating hierachical diagrams of tasks. Even if there are tasks that can be formalized and operationalized, such as printing a document or opening a file, many tasks do not lend themselves to this kind of analysis. Consider the example of preparing a diagram or an illustration of some topic[8]; it is likely that no two people would come up with identical solutions. Furthermore, environmental or organizational problems leading to users operating under time pressure, feeling watched over, experiencing hindrances to communicate or to retrieve information, will hardly fit the formal models and notations, has led to a shift in focus to less detailed and formalized techniques [134, p. 426]. Many task analysis techniques are also considered difficult to use and to integrate with software engineering practice, which is also acknowledged by their advocates (e.g., [35]).

8. Another example is provided in Paper VI, namely release planning.

Issues on an organizational level have been considered by the information systems community for a long time [66][4]. Here, the aim is to provide a organizational (and procedural) context for user tasks. An example of task analysis on a higher level is offered by Goldkuhl [51], from which the techniques and notations used in the Delta method were adapted. This sort of notation will inevitably leave out important information for the usability engineer, but will outline the procedures and the procedural context of a task and relate the task to a more general organizational goal. The information which is not captured in the diagram must be recorded by means of, for instance, notes or tape recordings.

A typical result of this type of task analysis is a document describing what tasks the users should be able to accomplish with the system; What information will they need to be able to do this? What actions or steps are needed and what are the relationships between these steps? What constraints does the work environment impose? What problems do the users perceive today etc? The typical way to capture this knowledge about the users' tasks is by means of interviews and surveys, and by observing the users' habits and behaviors as they are carrying out their work, by analyzing local work descriptions and other documents etc. The task descriptions are usually in the form of some graphical notation with textual annotations.

Setting goals and specifying usability

Generally speaking, the first two phases focus on the current situation, aiming at an understanding of the users and the tasks. For the next step we turn to the future and start modelling the properties of the prospective system.

Usability is often considered difficult to specify and verify (see *UE and RE–a comparison* on page 65). But if we again look at the ISO definition of usability (repeated below), it is almost self-evident how this should be done.

> [Usability is the] extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use.

Specifying usability is thus a matter of describing who should be able to accomplish what, to which degree of efficiency, effectiveness and satisfaction. In the same way, to verify usability we have to study the specified users attempting to solve the specified tasks in the specified context, and see if they can do this according to our own specified standards[9]. Since users were specified during the user profiling, and the goals/tasks and the context in which they occur was specified during task analysis, the remaining task is to define what is meant by efficient, effective and satisfactory. Or, according to our own definition, relevance, efficiency, learnability and attitude.

In the discussion of usability definitions we also noted that usability is a quality of the interaction between user(s) and system–not a property of the system itself. Consequently, usability requirements should focus on the interaction and not on the system (as in e.g., "Provide undo."). For a discussion, see the section on "Objective proximity" on page 71.

There exists a large number of possible measurement criteria. Whiteside *et al.* [162], provide a list of 22 such criteria, and Nielsen [123] and Löwgren [104] provide additional attributes, which have been grouped according to the four usability attributes in Table 1

Note that some measurement criteria could be relevant to several usability attributes, which indicates that the usability attributes cannot easily be seen as orthogonal dimensions[10]. This is also quite intuitive; if, for instance, it takes unnecessarily long time for a user to complete a certain task, he or she will probably not have a positive attitude towards the system either. Furthermore, one measure may explain different things. For example, if the number of available commands invoked by the user is low relative to some value, it may be due to the fact that those commands used are very powerful (high efficiency), or that there are many commands that are not necessary (low relevance) or maybe the user could not remember the right commands (low learnability) and could not get help to find them in the manual (low efficiency of

9. In practice there may be a number of reasons to deviate from this, but the general concept is very clear and each deviation should be justified. Provided, of course, that the ISO definition is accepted.
10. A recent study indicates that the correlation between various usability attributes depend on the domain, the users' experience and the context of use [43].

documentation). An example of how usability can be specified is given in Paper II.

Table 1: Different usability measurement criteria grouped according to the four usability attributes: relevance, efficiency, learnability and attitude. (Compiled from [162][123][104])

Relevance	**Efficiency**	**Learnability**	**Attitude**
• number of good and bad features recalled by users	• time to complete a task	• ratio of successes to failures (over time)	• percent of favorable/unfavorable user comments
• number of available commands not invoked by users	• percent of task completed	• time spent in errors	• number of good and bad features recalled by users
• number of available commands invoked by users	• percent of task completed per unit time (speed metric)	• percent or number of errors	• number of users preferring the system
• number of times user needs to work around a problem	• time spent in errors	• number of commands used	• number of times user loses control of the system
• percent of task completed	• number of commands used	• frequency of help and documentation use	• number of times the user is disrupted from a work task
	• frequency of help and documentation use	• time spent using help or documentation	• number of times user expresses frustration or satisfaction
	• time spent using help or documentation	• number of repetitions of failed commands	
	• number of repetitions of failed commands	• number of runs of successes and of failures	
	• number of runs of successes and of failures	• number of available commands not invoked by users	
	• number of times interface misleads user	• number of features or commands that can be remembered after a test	
	• number of times user needs to work around a problem	• number of logical errors made	
	• number of times the help facilities solve the user's problem		

Testing for usability

As indicated previously, testing for usability is a matter of studying users using the system and verifying that the interaction works as smoothly as defined in the specification. This is, however, only half the truth, since there are two fundamentally different approaches to evaluating usability: summative evaluations and formative evaluations [104]. In the case of summative evaluation, the basic underlying question is: "Is this system usable?", and with strict adherence to the definition there is no other way to find out than by studying users using the system. In formative evaluation, however, the basic underlying question is: "How can we improve the current version of the system (so that we will be able to meet the specification)" Thus, formative evaluations seek design information, which we will get back to shortly.

Usability testing is typically carried out in specially designed "usability labs" (see, e.g., [5] for a good description). These are rooms with several video cameras, a one-way mirror through which the user can be observed while attending to the test tasks, and a control room behind that mirror from which the video cameras are operated. The rooms may have specially equipped computers that could log the sessions, and the rooms are typically well isolated from the environment, to avoid distraction of the users. Large software companies may have their own usability labs, and there are consultancy firms that offer the same services on a rental basis. Rowley [141] describes how usability testing could be accomplished in the field, which may be advantageous for, e.g., a distributed customer base or when there are logistic problems in bringing the users to the stationary usability laboratory. Field testing is of course preferable from a validity perspective, since one is able to perform the test in the *specified context*.

Usability testing can also be used for formative evaluations, but for such purposes there are other techniques too. Maguire and Sweeney [113] identify three categories of usability evaluation of which the above discussion only has bearing on the first:

- User-based approaches, where the system is evaluated during usage.
- Theoretical approaches, which are based on the application of formal models of task performance
- Expert-based approaches, based on HCI experts' opinions.

The latter two categories involve no user participation, but are based on findings from several years of research in cognitive psychology and human-computer interaction. *Computer-based tools*, for example, helping the designer to design and/or evaluate a user interface, belong in this category. Byrne *et al.* [19] present a system for automated evaluation of a user interface. The method used is based on GOMS analysis [20], and the system is integrated with a user interface design tool, providing a comprehensive design environment for the user interface designer. One of the problems with these tools is that by enforcing guidelines in the design process they necessarily tend to constrain the creativity of the designers. In response, Löwgren and Nordqvist [107] propose a different type of evaluation tool, which based on a knowledge-base containing general design guidelines and toolkit specific style guides, gives *critique* to the designer. The system is able to give messages [108] in the form of :

> The items in the **File** menu are not standard. The following items should be in the menu: **New**, **Open**..., **Save**, **Save As**..., **Print** or **Print**..., **Close** and **Exit**. (Motif Style Guide 3.3.2, 4.2.3, pp. 7-3, 7-4)

Evaluation tools, especially when integrated with user interface development tools, do indeed have a potential place in the development work [106][39]. This holds especially when real users are hard to find or get responses from. In most cases, some form of user participation is preferred, though.

Even so, *heuristic evaluation* (attributed to, and intensely advocated by Nielsen), which is carried out by HCI experts has been suggested as a very powerful method (see, e.g., [121][122][125]). The procedure could simply be described as "looking at an interface and trying to come up with an opinion about what is good and bad about the interface." [123, p. 155]. The evaluation is carried out according to a set of heuristic rules, such as "*Speak the users' language*" and "*Minimize the users' memory load*".[11] It has been suggested that by such heuristic evaluation 75% of all usability problems could be discovered, if carried out by 5 different evaluators [124]. However, the method requires that the evaluators have extensive experience to be effective [122]. Such evaluators are usually a scarce resource. I suspect that few companies, especially in Sweden, even have access to five HCI experts. However, users are sometimes also a scarce and sometimes expensive resource, in which case this technique may be useful. Nielsen offers what he calls the "discount usability engineering" approach, as the name implies intended to be less costly than full usability engineering [120].[12] This discount approach includes such heuristic evaluation instead of usability testing.

Other evaluation methods include the use of general guidelines for evaluation, in much the same manner as in heuristic evaluation, and the so called "cognitive walkthrough" techniques. These techniques are based on the user's cognitive activities, and specifically the goals and knowledge of a user while performing a specific task. During walkthroughs, the steps required to accomplish the task are evaluated by examining the behavior of the interface and its effect on the prototypical user [161]. The evaluation is done with respect to how well the system leads the user to perform actions and how well the results of those actions match the expectations of the user. Several variations on this theme exist (see, e.g., [142][61][89]).

11. It may seem as these principles are trivial, and that most designers are probably aware of them, but Molich and Nielsen has shown the opposite [119].
12. The "discount" approach was presented in response to an article by Mantei and Teorey [114], in which it was claimed that the "cost required to add human factors elements to the development of software in a certain case was $128,330". The calculation was based on Boehm's COCOMO model [12]. Nielsen [120] showed that by investing half of that sum ($65,330), significant improvements could be made.

Table 2: Findings from a comparative study of four evaluation techniques. Source: [83]

	Advantages	**Disadvantages**
Heuristic evaluation	Identifies many more problems Identifies more serious problems Low cost	Requires UI expertise Requires several evaluators
Usability testing	Identifies serious and recurring problems Avoids low-priority problems	Requires UI expertise High cost Misses consistency problems
Guidelines	Identifies recurring and general problems Can be used by software developers[a]	Misses some severe problems
Cognitive walkthrough	Helps define users' goals and assumptions Can be used by software developers[a]	Needs task definition methodology Tedious Misses general and recurring problems

a. As opposed to HCI experts. (My comment)

Comparisons made between different evaluation techniques indicate that there are pros and cons with them all. Table 2 summarizes findings from a study by Jeffries *et al.*, who conclude that the usability testing and heuristic evaluation techniques have some advantages over the use of general guidelines or cognitive walkthrough techniques, in that they tend to discover more severe usability problems.

Usability evaluation is the final step of one design cycle. In the case of usability engineering, the general idea is to iterate design (as shown in Figure 2 on page 31) until the usability requirements have been met.

Current status of Usability Engineering

The late 1980's and the early 1990's were the years when the debate on usability-oriented methodology issues saw its peak in the academic literature. In a foreword to the proceedings of HCI'96 [144, p. ix] the editors state: "Methods and tools, which were the focus of the early years, have moved out into the hands of the practitioners who are providing feedback on their applicability to a range of design problems." This summarizes to a great extent what we have seen since the time of the Delta project. UE has moved out in the industry, where most large software companies today have usability engineers employed[13]. There are several consultancies offering usability services, either as part of a larger business, or as the sole business concept.

As for recent contributions in academic literature, methodology papers have been few, and usually concerned with applicability of UE techniques to new domains (e.g., web usability) or new situations (e.g., remote usability testing). To a great extent usability engineering has made way for more contextual approaches (cf. *Towards a wider context* on page 74). Even in more practice-oriented conferences (e.g., the Usability Professionals Association, UPA) methodology issues seem to have received less attention as usability-oriented techniques have become more commonplace.

2.2 Requirements Engineering

Usability is but one of a great number of qualities that software may have or lack. It competes for development resources with properties like how easy it should be to migrate the software to different platforms, how easy it should be to re-use the code for other purposes, and how secure it should be, just to mention a few. The process of deciding on what combination of properties a software system should have is called Requirements Engineering. More formally [97 p.9],

13. However, this does not necessarily mean that their prevailing development processes are usability-oriented.

> a requirements engineering process is a structured set of activities which are followed to derive, validate and maintain a systems requirements document.

As is evident from this and several other definitions (e.g., [11][140][16]), the *requirements document* plays a central role to RE, and represents the main tangible product of the activity[14], whereas perhaps the most important product is intangible; *understanding*. Thus, the primary issues within RE could be described in terms of arriving at a high-quality requirements document. Davis *et al.* [32] probably provide the most comprehensive list of 24 such quality attributes which may serve as a map for outlining some of the main research issues within RE.

The requirements specification (RS) should be *unambiguous*, i.e., each requirement should have one and only one possible interpretation. This has led to a large body of research into formal representation (e.g., [151]) of requirements. Formally representing the requirements also allows for machine translation or interpretation, so that the RS may be directly *executable*, i.e., transformed into a behavioral model [168]. However, strict formalism threatens the *understandability* of the RS, and here matters are arranged for an incessant debate (e.g., [99][81]). On the one side of the RS there are the humans, e.g., customers, users, designers, project managers, who must all be able to gain a common understanding of the prospective system, and who all speak natural language. On the other hand there are machines (that don't understand natural language and have little tolerance for the inherent ambiguity of non-formal languages) which are supposed to carry out the actions–should they do exactly what the stakeholders *say*, or exactly what they *mean*?

The RS should also be *complete*, i.e., every property and capability that the software is supposed to have should be accounted for in the specification, and each property should be *traceable* from stakeholder to implementation [54][128]. This raises the question of how we identify all these properties, and a plethora of techniques within this major area

14. The terms "document" and "tangible" are still valid with the proliferation of requirements databases; most specifications are still materialized as documents.

called requirements elicitation have been proposed over the years (e.g., [49] for a classification). Note that identifying these properties is also one of the main goals of the usability engineering process, and several techniques, e.g., prototyping, are common to the two communities.

On the other hand, completeness must not entail *redundancy* or lack of *conciseness*, and above all, there should be no requirements that do not contribute to the satisfaction of some real need. This *correctness* criterion is related to the relevance attribute of usability (see *Setting goals and specifying usability* on page 33).

Different stakeholders may have different needs, and quite often even conflicting needs. Such conflicts have to be sorted out [38][9][8] as early as possible, so that the RS is both internally and externally consistent. Moreover, the stakeholders may have different views of the relative importance of the requirements, and what should come first in a world of limited resources. Unifying these viewpoints is not trivial (e.g., [41][47][118]), but nevertheless the requirements should be annotated by a consentient priority, stability and by which version of the future system they are scheduled for.

The operative goal for the RS is to serve as input for design and testing. To this end it is of course vital that the requirements are achievable at all (and prototyping plays an important role even in this case), but on the other hand it is equally important that the requirements do not put unnecessary constraints on the design. One of the cardinal rules within requirements engineering is that the requirements should describe “what” rather than “how”[15], or in other words, the problem rather than the solution, and thus allow for as many different solutions as possible–the RS should be *design independent*. For testing purposes it should be easily verifiable whether a requirement is met or not, and to this end it has to be assigned a precise target value on a defined scale [96][45].

Finally, Davis *et al.* propose a few criteria concerning the structure and style of the document itself (e.g., it should be well organized, contain cross-references, and be electronically stored) which has more to do

15. Quite often, one person’s “what” is another person’s “how”. See [31 p. 17] for an entertaining treatment of this dilemma. (See also [82 p. 1][146])

with the documentation process, and which also comes as a side-effect of employing current requirements management tools.

Some of these issues need further elaboration to provide a rich context for the understanding of the more RE-oriented papers in this thesis, and section *2.2.4 Focussing in on release planning* will address these in some depth. But before that, a more general discussion of the RE fundamentals is provided, seeking answers to what requirements really are, and what activities these requirements give rise to. As noted in the introduction to this chapter, the RE-savvy reader is advised to skip sections 2.2.1 and 2.2.2 to avoid boredom.

2.2.1 REQUIREMENTS

"*The purpose of a requirement is to reproduce in the mind of the reader the intellectual content which was in the mind of the writer*", Harwell *et al.* [72] sententiously states. Although this would be true for all writing, indeed most communication, and although the perspective of a writer and a reader is way too narrow, the statement still points at an important duty of requirements; to reproduce intellectual content.

Most RE textbooks have a chapter or section dedicated to the question of what a requirement is (see, e.g., [31][82][156][103]), and many of them also cite the IEEE definition [79]:

1. A condition or capability needed by a user to solve a problem or achieve an objective.
2. A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification or other formally imposed documents.
3. A documented representation of a condition or capability as in 1 or 2.

Thus, the term "requirement" is used to denote both the needed condition/capability, and its documented representation. More notably, a requirement is a requirement even before it is documented in any form (this is called a "raw" requirement in the IEEE terminology [78]), and even regardless of whether the user is aware of his or her need. Thus, requirements are seen as existing in some requirements space, independent of our awareness, and ready to be captured by requirements engineers (see [84] for a critical point of view). Just as design solutions are sometimes seen as residing in design space for the designer to explore (see, e.g., [109, p.69 ff]).

Several classification schemes for requirements have been proposed. Kano *et al.* (cited in [90, p. 18 ff.]) suggest that there are *normal*, *expected* and *exciting* requirements, of which only normal requirements are usually stated explicitly. Expected requirements are so obvious that they may not even be mentioned (for example, we expect a car to have a steering-wheel), but may of course be disastrous if neglected. Exciting requirements, on the other hand, represent features or properties that come as positive surprises to the customer, and may provide a competitive advantage for the supplier. Extensive reliance on such requirements could be traced in today's IT industry, and unfortunately one can suspect that many of them are more exciting to the developers than to the users.

Functional requirements

Another way of characterizing requirements is the ubiquitous division of functional (or behavioral) and non-functional (or non-behavioral) requirements[16]. The general idea is that functional requirements specify what output the system should provide given some (also specified) input (e.g., [31 p. 212 ff.]). An example of a functional requirement would be:

> Pressing the "x^2" button shall result in the square of the current entry being calculated and displayed.

Basically this means that when the functional requirements are satisfied, the system "works" in a mechanistic sense. However important this is, there is no guarantee that the system is *useful* in any way to any human being. For example, even if only one person alive could ever find and remember where the "x^2" button is located in the interface, and even if it took several weeks for the system to compute the square, the requirement would still be fulfilled. Such considerations are specified by non-functional requirements.

16. Functional requirements are *capabilities*, and non-functional requirements are *conditions* in the IEEE terminology [78, sect 6.1], but these terms do not seem to have been adopted to a great extent by the RE community.

Non-functional requirements

As the term implies, non-functional requirements describe every aspect of a system except its functionality. It is by means of non-functional requirements that we can make sure that most prospective users of the square function from the previous example can actually find the "x^2" button (usability) and get a result within reasonable time limits (performance). Thus, they come in many forms, and many authors have suggested different classifications. Boehm, in his classical paper [10], defines a hiearchy of "software quality characteristics", including 15 attributes on the lowest level (e.g., "accuracy", "accountability", "completeness") and shows also how different attributes relate to each other (e.g., "human engineering" is made up by "robustness", "accessibility" and "communicativeness"). Kotonya and Sommerville [97] define three classes; process, product[17] and external requirements, in which they place 12 different types of non-functional requirements, whereas Karlsson [90, p. 21] suggests a division of "quality requirements", "constraints" and "other requirements". If we appeal to authority to ease our confusion, we see that the standard [79] changed from 13 different classes in the 1993 revision to 5 in the 1998 revision, and the Encyclopedia of Software Engineering [115, p. 963] instead states 11 different types of "quality" requirements. Obviously, there is no general agreement on how to classify non-functional requirements.

The lack of a generally agreed classification scheme reflects the fact that it is exceedingly difficult to draw straight lines between different types of requirements. The division of functional and non-functional requirements is problematic, as noted by several authors, because many non-functional requirement on one level may result in functional requirements on a lower level. Security issues could be solved by authentication functionality, and some usability issues could be fulfilled by providing undo functionality, to give just a couple of examples. Moreover, non-functional requirements are rarely independent of each other, or orthogonal. Security facilities always stand the risk of becoming usability issues, as do performance requirements, for example.

17. Usability is categorized as a product requirement in Kotonya and Sommerville's taxonomy. By contrast, see the definition of usability on page 24.

Good requirements

A legitimate question at this point is what constitutes a “good” requirement. According to the quote by Harwell *et al.* above, a requirement statement would be good if it is able to reproduce in the mind of the reader the intellectual content in the mind of the writer. Many of the quality criteria for the general requirements document have bearing on just this, and IEEE provides some further guidance as to the *form* of specific requirements. They define what they call a *well-formed requirement* as [78]

> ...a statement of system functionality (a capability) that can be validated, that must be met or possessed by a system to solve a customer problem or to achieve a customer objective, and that is qualified by measurable conditions and bounded by constraints.

Thus, well-formed basically means that the requirement is stated explicitly (as opposed to “raw” requirements, cf. page 43) and that it is verifiable, since the rest of the definition is merely a paraphrasing of the IEEE definition of requirements. According to the same standard, each requirement should possess the following properties:

> a) *Abstract*. Each requirement should be implementation independent.
>
> b) *Unambiguous*. Each requirement should be stated in such a way so that it can be interpreted in only one way.
>
> c) *Traceable*. For each requirement it should be feasible to determine a relationship between specific documented customer statement(s) of need and the specific statements in the definition of the system given in the [system requirements specification] as evidence of the source of a requirement.
>
> d) *Validatable*. Each requirement should have the means to prove that the system satisfies the requirements.

Apparently, the quality of a requirement statement is judged by its form. But if we take one step back it may be reasonable to paraphrase Harwell *et al.*: *“The purpose of a requirement is to reproduce in the mind of the* designer *the intellectual content which was in the mind of the* issuer*”*. The obvious consequence of this would be that a good requirement is one that conveys to the designer an understanding of the actual need of, for example, the user. And thus, the validatability property above would be described more along the line with *Each requirement should* *have the means to prove that the system satisfies the actual needs of the customer.* Admittedly, the traceability property, and its "abstract" and perfectly "unambiguous" explanation, reveals an intention to keep track of the issuer's *documented statements* in case the real need perchance should attract further attention after the requirements have been specified. It is clear, however, that this measure will only propagate the problem of specification to the issuer.

In the section labeled "Objective Proximity" on page 71, another quality aspect of requirements will be discussed, which has to do with how well a requirement statement maps to the real objective of the stakeholder.

2.2.2 REQUIREMENTS ENGINEERING PROCESSES

The requirements document is the product of RE, so let us now briefly turn to the process of producing such a document, to set the stage for a more focused treatment of the issues relevant for this thesis. For more comprehensive descriptions, the reader is referred to Loucopoulos & Karakostas, [103] or Davis [31] for a general treatment, and Sommerville & Sawyer [149], or Graham [60] for more practical and normative accounts.

Most software development processes include at least one separate phase for "elaborating user needs" or "requirements analysis". In the simplest, two-step, description of software development; 1) "Deciding on what to build" and 2) "Building it", RE represents the first step. RE starts when a problem has been perceived, which seems possible to solve with software[18]. The process then aims at taking one step on the path from the problem to a solution. Pohl provides a useful model for thinking about this path [131]. It differs from most other descriptions in that it has no stages, i.e., no implicit or explicit chronology. Instead, Pohl suggests that there are three continuous and orthogonal dimensions of RE:

18. In many cases, the solutions actually come before the problem has been perceived. This is especially common in market-driven development situations (see *Market-driven systems development* on page 57.)

- The *specification* dimension, which deals with the degree of requirements understanding at any given time. The objective for the RE process is to move from a vague imagination of the prospective system (*opaque*) to a thorough understanding of the same (*complete*).
- The *representation* dimension, which has to do with the degree of formality of the specification. Here, the objective is to move from an *informal* representation, such as sketches and natural language descriptions, to a *formal* specification, using some sort of formal notation. .

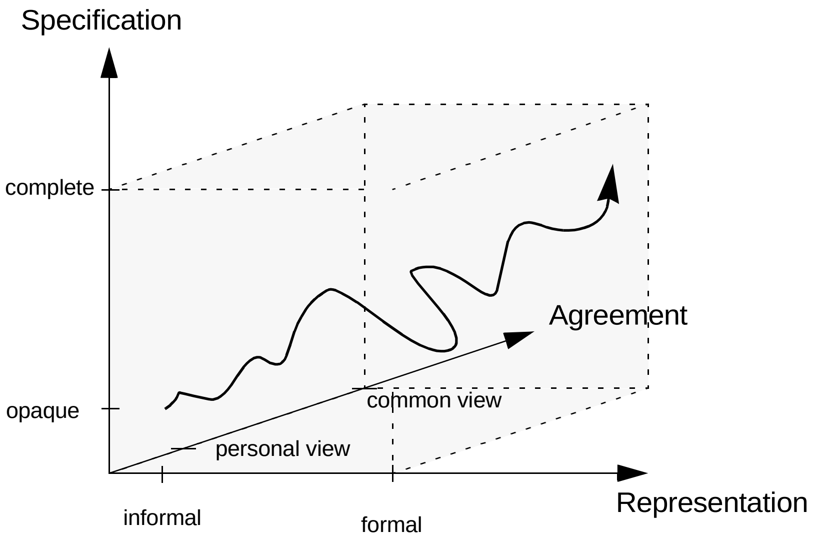

Figure 3: The three dimensions of requirements engineering [131]

- The *agreement* dimension represents the degree of consensus among the people involved in the process. The RE process should seek to merge all different individual ideas of the prospective system, that will inevitably exist in the beginning, to one common and consentient view.

Even if the ideal end-point of the model, i.e., a complete, formal and agreed upon specification, could be questioned from a practicality point of view, it provides an appealing conceptual model for the endeavours of any RE process.

The most common way of describing the RE process, however, is to divide it in four or more groups associated with 1) gathering requirements, 2) understanding and analyzing them, 3) documenting or representing them, and 4) validating these requirements against e.g., the customer's needs. These four major tasks are outlined next.

Elicitation. This stage, sometimes called requirements acquisition or discovery, includes consulting stakeholders, system documents, domain and organizational knowledge, standards and regulations, and market information. The techniques used include questionnaire surveys, workshops, scenario-based techniques, interviews etc. See page 69 for further comments, and [49] for a general discussion on elicitation techniques.

Analysis & Negotiation. Once elicited, the requirements need to be analyzed thoroughly, and during this process there will inevitably arise conflicts and inconsistencies which have to be resolved. The analysis also involves:

- checking the requirements for necessity and/or prioritizing them (see *Planning for incremental development* on page 60).
- checking them for feasibility, which may involve prototyping
- estimating the resources needed to realize them.

For a classification of analysis techniques, see [77].

Documentation. This process is commonly referred to as requirements modelling, and serves to produce a requirements document with the qualities described earlier. A multitude of modelling techniques and notations has been proposed, including data-flow models, entity-relationship diagrams, object-oriented models, state-transition diagrams, and process models showing principal activities and deliverables involved in carrying out some process.

Validation. The final stage of requirements engineering aims at certifying that the requirements document represents an acceptable description of the system which is to be implemented. This primarily involves stakeholders, who should verify that the specification describes the system that they need, and system designers, who should verify that the specification is sufficient for the design of the proposed system.

It should be noted that even in the most rigid system development methods, these four activities do not take place in a strict sequence, and in most practical cases there are no distinct boundaries between them. Instead the activities are interleaved throughout the RE process.

2.2.3 THE REQUIREMENTS-DRIVEN EVOLUTIONARY MODEL

The Requirements-Driven Evolutionary Model (RDEM), described in Paper III and Paper IV, has some characteristics that may need a more elaborate discussion than permitted by the common paper format. Several authors have proposed taxonomies or frameworks for comparison of RE approaches (e.g., [77][74][166][18]), but since RDEM is more of a meta-model for requirements management, these frameworks do not apply very well. Hughes *et al.* [77], for example, use *concerns* (i.e., functionality and quality) and *frames* (i.e., various modelling techniques) as their point of departure, whereas Hofmann [74] suggests the two dimensions of data- vs. process oriented and viewpoints- vs. specification-oriented as base for categorization. Yadav *et al.* [166] propose three questions as a framework for classification: *What should requirements be?*, *How should requirements be stated?* and *How should requirements be derived?* RDEM does not provide detailed answers to any of these questions. Instead it suggests an infrastructure for requirements management that each organization can instantiate to suit their specific needs, traditions and preferences. In fact, they can continue to use much of their current tools and techniques.

As different users need different software systems, different development organizations need different processes. This becomes very clear in the context of a large development organization as Ericsson, which is spread over hundreds of countries and cultures. Still, these organizations need to be able to at least synchronize their development efforts. Thus, it is important to find a way of doing this without necessarily forcing all organizations to operate in the same way. RDEM is an attempt to overcome this, by simply stating what should be known about the requirements before next step of the process is allowed to be taken–not how this knowledge should be acquired. The prize for this flexibility is paid in terms of operational strength, in that it does not stipulate any procedures.

This focus on knowledge and understanding is another important foundation for RDEM. (RE) processes are sometimes seen as an "organized set of activities which transforms input to output" [97, p.25]. Or it can be described as a way of achieving a correct and complete specification of software requirements [74]. At a higher level of abstraction, it is easy to see how it is in fact a learning process. Lawrence is clear on this point [100]:

> Many authors have claimed that the primary output of a requirements process is a requirements specification. Not so. The primary output is our collective understanding [...] The specification is only a representation, a *model* of that understanding.

When a project starts out very little is known by any of the stakeholders, but as people meet and communicate, contemplate, make statements and review these statements, the stakeholders learn more about the prospective software. After a while, many of the "what" and some "why" questions have been answered but there is still a predominant vagueness surrounding the "how" questions. Later, the focus is on *how* various features and qualities should be realized. As knowledge increases, the details fall into place, and the chaos is curbed. Borrowing terminology from the field of physics, we could say that the entropy decreases as the development progress[19]. This also maps well to Pohl's specification dimension described above (see Figure 3 on page 48). RDEM reflects this view of software development as a learning process, in that it defines progress in terms of levels of understanding of the requirements. Go/no-go decisions are based on how much is known, i.e., whether an adequate subset of the requirements attributes are set or not. As a consequence, what drives the development forward is a matter of "setting attributes".

19. From a physicist perspective this is of course a horrendous misuse of the term. According to the second law of thermodynamics, entropy is continuously increasing.

Measuring progress

One may speculate about the possibilities of measuring progress of a development effort employing an attribute-driven approach. Measuring progress between phases or states in software development has always been a problem, and the "real soon now" syndrome is widely acknowledged. If we could measure the entropy at any given point in time, we could also define decision-points in a quite simple way. For instance, we could say that the implementation phase may not begin (in part or whole) until we have reduced the entropy to 20%.

The attribute-driven approach may provide such a measure, if we define the entropy of a project loosely as the proportion of attributes not yet set. The problem is that some attributes may be more important than others, so simply counting the number of set attributes will not suffice. Instead, attributes will have to be weighted in some way. For instance, the attribute describing the typical user associated with a specific requirement will most probably be more important to have as a base for decisions, than the time stamp of the requirement.

One could think of an even more refined way of weighting the attributes. Take the attribute "cost" as an example. The requirement "superclass" may contain the attribute cost, which is an estimate of the resources needed to implement the requirement. This attribute will be very important when it comes to prioritizing the requirements. At first the cost could be a rough conjecture, but as the requirement is classified and refined, the cost could be (in part) derived from cost attributes of the requirements subclasses. For example, once we know that there are three different user categories, and we will have to make 15 interviews, we will know the cost of the user profile attribute. Thus, we may be able to revise the cost value for the superclass. The in part calculated value of the cost attribute should then have more weight than the "wild guess" value of the same attribute.

Again, these are speculations that need to be tested, and the issues of progress measurement is also mentioned among the suggestions for further research (on page 17).

2.2.4 FOCUSSING IN ON RELEASE PLANNING

We have outlined some issues regarding the product and the process of requirements engineering in general, but the task of release planning demands a more narrow description of its context. In this section a few associated concepts are explained and contrasted with related issues.

Incremental systems development

Software process models determine the order of the various stages or activities involved in software development. The Waterfall process model presented in 1970 [143] is by far the most influential such model. It is as well known as the arguments against it, despite of which it has survived in many incarnations into this century. The 1980s brought several alternatives to the monolithic approach of the waterfall model, including several types of incremental approaches (see, [13] for a brief overview, and [112, p.35ff] for a more elaborate treatment). Since this is yet another area within software engineering where terminology is bewildering and concepts like "incremental" and "iterative" are easily confused, this section is an attempt to explain my own view. I make no claims as to presenting stringent definitions.

Iterative development means that one or more stages of the development process are passed more than once. Iteration occurs in all systems development at various points, but the *process model* is not iterative unless the iteration occurs on the process level. For example, the usual turn-around of coding/compiling/debugging does not make the development *process* iterative, whereas the explicit loop of Usability Requirements/Conceptual Design/Prototyping/Usability Testing of the Delta method (described in Paper I) represents iteration on a process level.

Incremental development is the development of a system in a series of partial products, increments, throughout the development timeframe. In its ultimate form, an increment is a self-contained functional unit of software, together with all supporting material, e.g., specifications, test plans, user manuals, estimates, resourcing, quality assurance and configuration management information.

The fact that the development of a system is done incrementally does not automatically imply that each increment is delivered to a customer–neither internal nor external. On the contrary, when a system is developed from scratch, it is common to use several increments just for learning purposes, i.e., for requirements elicitation and analysis. The first release of the system is made when the system is mature enough for practical use. Therefore, it is important to make the distinction of development and delivery, as done by e.g., Graham [60, p.31].

There are several forms of incremental development/delivery, which are explained synoptically in the following, with some terminology borrowed from [59].

Evolutionary delivery.[20] This is the purest form of incremental development, where useful increments are delivered quite early in the life-cycle, and knowledge from previous increments is fed back continuously to the process. Each delivery contains some functionality that is relatively easy to produce, but still is useful to the customer. Deliveries may be as frequent as once a week, but once per month to once per six months is a more common pace.

The change requests that are elicited from each delivery should be planned for in advance, and drives the development to a large extent. Changes are inexpensive, compared to the monolithic approach, and the customer confidence typically increases as concrete evidence of development is shown frequently. There is usually additional processes surrounding the evolutionary model, with responsibility to set system and business objectives, architecture design, planning and quality assurance, for instance.

The advantages of evolutionary delivery (Figure 4) are more salient once the first release of a product has been made. The approach is less suited for development from scratch, in particular for very large systems. The evolutionary model represents one of the underpinnings of the RDEM model described in Paper IV, and is especially suited for market-driven product development, described later.

20. Some authors prefer to make a class distinction between incremental and evolutionary development (e.g., [37]). My own view is that evolutionary development belongs to the incremental process models, since the difference lies merely in the degree to which requirements generated from the use of one release are allowed to govern the next.

.

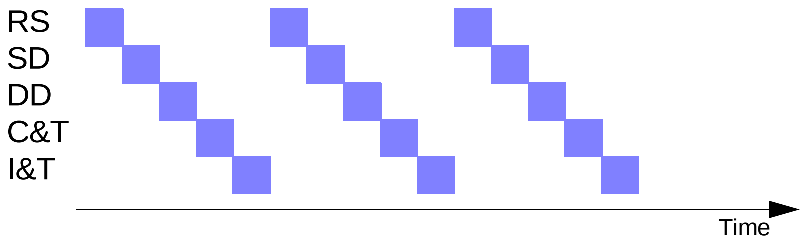

Figure 4: Evolutionary delivery approach. (RS= Requirements specification, SD= System design, DD=Detailed design, C&T=Coding and testing, I&T=Integration and testing. Note also that iteration may exist anywhere in the model, but have been omitted in the figure.)

Phased development. This model (Figure 5) is often used in the development of large systems, and the main difference to the evolutionary model lies in the size of the increments (phases). There is no clear definition of what is a "large" increment, in the evolutionary sense, nor a "small" phase. However, phases tend to be large and growing, and there is usually a temptation to put as much as possible into the current phase, instead of focusing on a minimal but functional and useful set of functions. Deliveries are usually made upon completion of each phase.

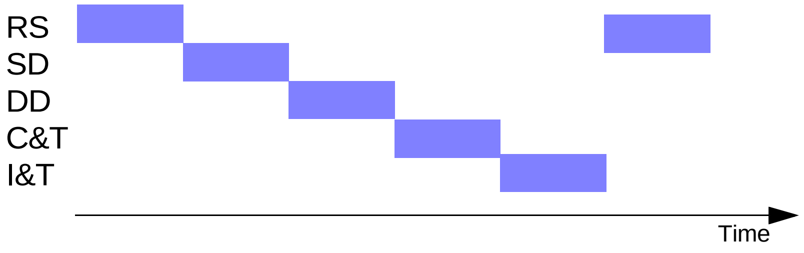

Figure 5: Phased development approach

Framework incremental development. This model (Figure 6) could be seen as a compromise between the waterfall model and evolutionary delivery. Here, the initial requirements specification and the architectural design is sufficient to guide the development process over several increments. It is important, however, that the specifications are considered to be half-completed and subject to change to a large degree.

The advantage with this model is that the development process moves in a direction that is apparent to everyone, while at the same time allowing quite early deliveries.

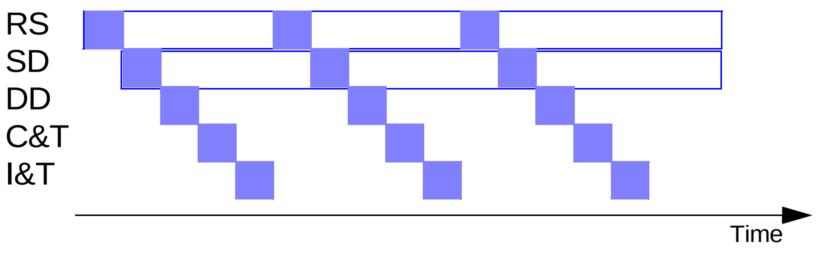

Figure 6: Framework incremental approach

Incremental build and test. According to this model (Figure 7), the incremental development begins in the coding phase, until which a monolithic approach is used. This is a common approach used in small projects and by individual developers constrained by process requirements. The incrementality in the later stages increases the internal quality to some extent, but it is often the case that the increments are not complete releases, i.e., with full documentation for example.

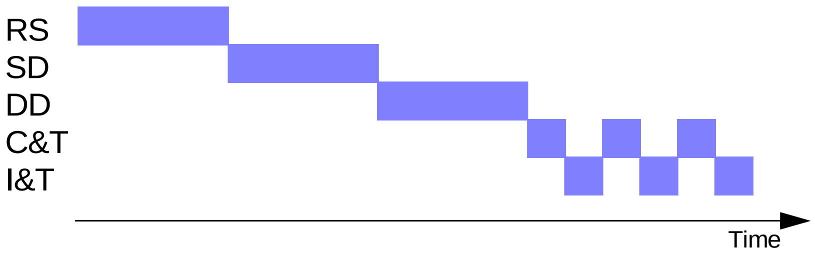

Figure 7: Incremental build and test approach.

It is evident from the illustrations that the various models have different implications for the RE process, and more specifically the degree of continuity of the RE activities. In the evolutionary model, requirements engineering is a continuous activity, whereas in Incremental build and test this is not the case.

In market-driven systems development (described next) it is common to employ the evolutionary delivery model once the first release has been made. RDEM, described in Paper IV, represents a separate class of process models, since it does not describe a sequence of events, but instead a sequence of states that the requirements should propagate through.

Market-driven systems development

The vast majority of educational and research literature within RE is based on the assumption that a large company or institution orders a large software system from either an external vendor or internal IS department, using a requirements specification as a contract[21]. During the 1990s it became apparent that this is not always the case (e.g. [22]). Instead, many software vendors develop one or several software *products for an anticipated market*, which puts numerous and fundamentally different demands on the RE activities. Table 3 provides a compilation of the differences between these two development situations, as found in the literature. A few of the entries in the list are worth some extra attention:

- *Requirements specifications are rarely written*. As we recall from the definition of RE, the requirements specification is most central to the whole process. In market-driven development, however, there is no contractual situation, and requirements are communicated

21. My conjecture is that the reason for this is that most of the early research within RE was funded by military or governmental agencies. Of the very few widely cited field studies in recent years, the first one by Curtis *et al.* [29] was explicitly aimed at large systems. In 1993 Lubars *et al.* [110] had a wider focus, and it was discovered that 14 out of 23 “projects” were market-driven.

verbally within the development organization to a greater extent [110].

- *The user is not known*[22]. There may not even be a user until the first release of the product. Thus, requirements cannot be elicited. Instead:
- *Requirements are invented.* Since the primary stakeholder is the development organization, this is also where the product is conceived. First after release, real users may supply new requirements [110][133].
- *The primary goal is time-to-market.* This is a survival attribute [126] to all market-driven organizations. Unless a product reaches the market in proper time, the market (shares) may be lost to a competitor. The release dates are thus kept fixed, at the possible expense of lower-priority requirements. In contrast, in bespoke development it is more common to keep a fixed requirements specification and find some flexibility in the time schedule.Another common characteristic is that the release dates are set to the same dates each year, so that the customers can be sure that there is a new release each June 15 and each December 15, for example. Thereby it is possible for the customers to plan their own organizational or infrastructural changes.
- *Specific issues include prioritization and release planning.* As a consequence of the strict time-to-market policy, requirements need to be prioritized continuously, so that the most vital requirements are guaranteed to be implemented in some explicit release. This is also what release planning is about.

.

22. For a perspective on user involvement in the two types of development (and a third: “in-house development”) see [65].

Table 3: Differences between bespoke and market-driven RE. (Compiled from [110][133][167][126][86][95])

Aspect	**Bespoke development**	**Market-driven development**
Primary goal	Compliance to the specification	Time-to-market. Requirements are jettisoned rather than allowing delay of release.
Measure of success	Satisfaction, acceptance	Sales, market share, product reviews
Life cycle	One release, then maintenance.	Several releases, as long as there is a market for the product.
Requirements conception	Elicitated, analyzed, validated	Invented. Either the market (marketing department) permits a feature, or technology does.
Requirements specification	Used as a contract between customer and supplier.	Rarely exist in orthodox RE terms, if so, they are much less formal. Requirements are communicated verbally.
Users	Known or identifiable. Termed *user, end user.*	Unknown, may not exist until first product is on the market. Termed *customer.*
Physical distance to users	Usually small	Usually large
Main stakeholder	Customer organization	Developing organization
Specific RE issues	Elicitation, modelling, validation, conflict resolution.	Managing a steady stream of new requirements. Prioritizing, cost-estimating, release planning.
Developer's association with the software	Short-term (until end of project)	Long-term, promoting e.g. investment in maintainability.
Validation	Ongoing process	Very late, e.g., at trade fairs.
Use of RE standards and explicit methods	More common	Rare
Use of iterative techniques	Less common	More common
Domain expertise available on the development team	More common	Less common product development often break new ground).

As pointed out, these fundamental differences give rise to new RE processes. For example, the REPEAT process developed by Regnell *et al.* in collaboration with a CASE tool vendor [136] focuses explicitly on prioritization and cost-estimation. Here, the requirements planned for a specific release are divided into a "must-have" set, and a "good to have" set, making up for 70% and 30%, respectively, of the resources and calendar time available until the fixed release date. Furthermore, the requirements can be issued by anyone, internal or external to the development organization, by means of a web application. Consequently, requirements are "dispatched" by requirements engineers as they arrive in a steady stream (on average three requirements per day), and propagated through the requirements management system as priorities, estimates and shallow analyses are added.

Planning for incremental development

In incremental systems development, one of the most crucial tasks on a process level is to decide what requirements should be realized in what increment. Yet it is difficult to find any guidelines in credible literature as to what should govern these decisions.

There are two main categories of planning, depending on the development situation. First, in the case of incremental development *without* incremental deliveries to a customer or market, the basis for selection of requirements can be more system-oriented. In this case considerations may focus on what is best from a system architectural standpoint, or what is best from a reliability perspective (crucial modules are developed first and thus tested at several points during development). This case is similar to traditional systems design and project planning. Incremental development of large systems often employs the Framework Incremental Approach (see Figure on page 56), in which plans are made on two levels: on the product level and on the increment level. In both cases the term increment planning may be used but, in simple terms, on the product level it denotes the task of deciding "what when", whereas on the increment level it refers to the activity of assigning resources within the scope of an increment, i.e., "who does what".

By contrast, in the case of market-driven development, the planning problem is more associated with what changes the market requires at various points in time. Here, the general goal is to select a set of requirements from a large set of candidates, that maximizes the value added for the customer within the constraints of the fixed release date and the resources available. In this case, more often referred to as *release planning*, or even *product planning*, the two most crucial parameters are: requirements priority and requirements estimated cost of resources.

Prioritizing a set of requirements is by no means a simple task. One only needs to consider such a relatively simple task as purchasing a package of milk: Is it more important that it has added vitamin-D than that the best-before date is two days later? Or is it worth the extra 15% on the price that it is "ecological"? The strawberry-flavored milk is tastier as it is, but it's not very useful for cooking, etc. Karlsson [91][92] (see also [63]) did some pioneering work on requirements prioritization, and the approach used was to let the stakeholder(s) decide on relative priorities based on pairwise comparisons of requirements. Thus, instead of ranking all requirements in absolute terms, the stakeholders are asked whether requirement R_1 is more important than requirement R_2 and if so, how much on a relative scale. Then, the same technique is applied to the estimation task, so that the relative cost of R_1 is compared to that of R_2. The partial relationships are then combined into a total ranking order for the set of requirements. The result of the comparisons can be displayed in a two-dimensional diagram, where it becomes clear which requirements provide most value for the cost.

However, in a market-driven situation there may be stakeholders all over the world, and these stakeholders may not all be equally important from a strategic point of view. Consider, for example, a software product that has a large market in the United States, a little smaller in England, and a few but enthusiastic users in India. It may not be reasonable to attach equal importance to these stakeholders, and thus some way of ranking the stakeholders and distributing the prioritization process is needed, such as the one presented by Regnell *et al.* [137].

Hence, if requirements were singular, i.e., not dependent on each other, the release planning problem would be one of prioritizing and assigning estimates to the requirements, and then selecting, in priority order, as many requirements as would fit the available resources. In operations research this is called a binary knapsack problem, and although the general knapsack problem is NP-hard[23], Jung has shown that there is a simple solution to this optimization problem in this particular case [85].

However, in Paper V we showed that only about a fifth of the requirements were truly singular. As an effect, selecting a requirement from the priority list may necessitate considering several others, further down the list. In the next section, we briefly present some theoretical background to requirements interdependencies.

Requirements interdependencies

Paper V and VI in this thesis deals with the notion of requirements interdependencies. The specific type of interdependencies studied relate to the problem of selecting a number of requirements for a certain release. These interdependencies (referred to as RP-dependencies in the following) may be either functional, e.g.,

A **requires** B to function (REQUIRES)

or value-related, e.g.,

A **increases the customer value of** B (CVALUE)

A **decreases the implementation cost of** B (ICOST)

Specifically, RP-dependencies increase the complexity of release planning significantly, since they render impossible the straightforward (although not simple in any way) approach of prioritizing the requirements and selecting the ones with the highest priority. This was identified by Karlsson and Ryan [93] and some steps were taken to support the consideration of interdependencies during prioritization, but the results are unclear and the authors request further research in the area.

However, there are many other types of relationships between requirements in a typical software development situation, which are related to other activities of RE.

One of the quality attributes of a requirement specification mentioned previously was *consistency*. In this context, an inconsistency can be defined broadly as “any situation in which two parts of a [require-

23. A problem is NP-hard if solving it in polynomial time would make it possible to solve all problems in class NP in polynomial time [117]. In essence, it is not known how to solve these problems within reasonable time limits.

ments] specification do not obey some relationship that should hold between them" [38]. Inconsistencies may be obvious once found, as in

> ReqX: *The contacts database shall provide fields for name, title and telephone number.*
>
> ReqY: *The email application shall retrieve email addresses from the contacts database.*

or they may be a little more difficult to identify, as in

> ReqX: *The network connection shall be based on the GSM standard for mobile telephony.*
>
> ReqY: *The network connection shall allow proper transfer of real-time video.*

Inconsistencies may have technical or practical causes. In a requirements document or database there may be several hundreds or thousands of requirements, which makes it extremely difficult to detect them–both among the existing requirements and when new requirements are added. The requirements themselves may also be complex, so that inconsistencies are hidden until further studies reveal them.

Inconsistencies also often arise as consequences of conflicts, which in turn exist in all organizations. Different stakeholders have different values, beliefs, interests and desires [111], for example, and all these cannot be satisfied by the development of a single system within reasonable cost.

Within RE, conflicts have also gained increasing interest during the 1990s, and the notion of ViewPoints [41] has emerged as a concept for managing different views, terminology and desire among stakeholders. A viewpoint is described as a "collection of information about a system or a related problem, environment or domain which is collected from a particular perspective" [97], p. 172. Since viewpoints result in differing requirements, viewpoint management normally involves conflict resolution, both on a social level and on a technical level. One advantage of the viewpoint model is that it allows individuals to have more than one point of view by abstracting away from the actual people involved in the process.

Robinson *et al.* [139], in a comprehensive report, introduce a new term and define *requirements interaction management* as "*the set of activities directed towards the discovery, management, and disposition of critical rela-*

tionships among sets of requirements." At face, this definition would include even the RP-problems, but a closer look at the underlying definitions reveals that it is mainly conflicts and inconsistencies that are targeted. For example, Robinson *et al.* state that:

> Two requirements, labeled R_1 and R_2, are said to *interact* if (and only if) the satisfaction of one requirement affects the satisfaction of the other.

This would still include RP-dependencies, but by satisfaction they mean:

> A requirement is satisfied by a component if the (implemented) component exhibits all the properties specified in the requirement–components operationalize requirements.

Satisfaction is then explained to be a scalar property, such that a component can satisfy a requirement to a certain degree. From the perspective of RP-dependencies this would cover the functional type. If R_1 requires R_2 to function, R_2 consequently affects the satisfaction of R_1, and thus there is a clear interaction between them according to Robinson *et al.* But in the case of a value-related RP-dependency, this does not hold. If R_1 increases the customer value of R_2, for example, it has nothing to do with the satisfaction of R_2, as defined above. Here, satisfaction has a pure functional meaning and is defined as a relationship between the *realization* (component) of the requirement and the requirement itself.

Another related research area within RE is traceability, which refers to the ability to describe and follow the life of a requirement in both a forwards and backwards direction [54]. Davis [31, p.190] instead use the terms *traced* and *traceable*, as seen from the point of view of the requirements specification, where traced means that the origin of each requirement is clear (backward), whereas traceable means that each requirement points to its realization (forward). Traceability information could certainly be viewed as dependencies[24], but differ significantly from the RP type of interdependencies. Furthermore, Kotonya

24. Several requirements management tools provide facilities for keeping traceability information, and we have investigated the feasibility of using these mechanisms for RP-dependencies. In short, it was concluded that such mechanisms were insufficient due to the conceptual difference between our purposes and those designed for.

and Sommerville [97, p.129] explicitly notes that "...Davis does not mention what we consider to be the most important traceability information namely information which records the dependencies between the requirements themselves."

The intention here is not to split semantic hairs, but to point to the fact that the vast majority of work done on dependencies between requirements has focused on either resolving functional inconsistencies in a requirements specification, or resolving social conflicts.

The RP-dependencies considered initially included temporal dependencies, e.g.,

A **has to be implemented before** B,

and also "OR" dependencies, e.g.,

Only **one of** A, B **needs to be implemented**

However, as is described in Paper V, these types were found to be redundant from a practical point of view: If A needs to be implemented before B, one might say that B **requires** A. Also, in the case of an OR dependency, this may either reflect the fact that one of A, B is redundant, or the more common fact that A and B may be very similar, but are aimed at e.g. different platforms or different markets or different use situations. In the latter case, it is probably wise to consider A and B together to save development efforts, which would consequently be an **ICOST** dependency.

2.3 UE and RE–a comparison

Considering the common concerns of HCI and Software Engineering, to strive for better software systems for the customers, it appears surprising how disparate these fields are, in practice as well as within research. But the disparity between Usability Engineering and Requirements Engineering, sharing not only basic concerns, but also a great deal of fundamental beliefs, values and preferences, is even more facinating.

In the following I present some of my personal views on the gap between RE and UE. It is mostly based on my own experience (and prejudices) from having worked in the intersection of the software industry and academia on the one hand, and in the intersection of requirements engineering and usability engineering on the other hand, for long period of time. In some cases I have found support for these views in the literature.

2.3.1 REQUIREMENTS FROM A UE PERSPECTIVE

It is my experience that usability engineers have a hard time relating their work to software engineering practice. One only needs to go to an HCI workshop or conference for practitioners to realize this, since sessions–whatever their main topic may be–frequently end up in therapeutical discussions about “the engineers”, or the “programmers”. They don’t speak the same language, and in general they come from a different culture. Goguen speaks of the “dry and the wet” in his interesting paper on computer cultures, and the meaning of the metaphorical terms are obvious to most of us [48].

These different perspectives collide at several points during software development. First, the “usability people” have to fight to get resources (project time) to do the user analyses. Once they do, however, the documented results often include very valuable information, but there is no receiver of this information. Not until it is transformed into requirements–and here comes the next collision; usability people have major difficulties specifying requirements. Either they are too vague to pass the first inspection meeting, or they are not usability requirements at all, but functional requirements labeled “usability” because they come from the usability people. But, since the team usually agrees[25] that the vague usability “ambitions”, or “goals” are quite important, they end up in a separate[26] “usability specification”, which may then be verified during separate usability tests in separate usability labs (this separation is further discussed in Paper III).

One of the most difficult problems is the gap between user requirements and functional requirements. Since, to many developers, the software engineering process starts with a functional specification and

25. If everyone doesn’t agree, the usability people will write such a document anyway. In their spare time, if needed.
26. And they will find support in the literature for this. Whiteside *et al.* [162] suggest that the usability specification should be seen as an *addendum* to the “real” specifications.

ends with a verification of the same, usability requirements (and other system requirements) are difficult to get into this "inner loop" of the development process[27]. This narrow perspective also reinforces the view that functional specifications are incontestable, which in turn is inconsistent with the view of software development as a learning process. (See [27] for an interesting case study related to the interplay between usability requirements and functional requirements.)

From my perspective, we, the usability people, have not always tried hard enough to align ourselves with software development practice. It seems so natural that if it is important that "the user should feel in control" of a certain application, we state that as a requirement. But when that requirement, or even more operational derivations of it, ends up in the hands of a function tester, he or she will not know what to do with it. As usability engineers we should know this. We should know how important it is to speak the language of the software developers.

Several attempts have been made recently to bridge the gap between RE and HCI by providing more formalism on a specification level (e.g., [101][165][138][34]), but we have yet to see this in widespread industrial use.

2.3.2 USABILITY FROM AN RE PERSPECTIVE

With a usability engineering background, it is tempting to approach comprehensive RE textbooks by first looking for the term "usability" in the index. This is usually a disappointing experience. For example, the most impressive and widely cited 500-page textbook *Software Requirements–Objects, Functions and States* by one of the most renowned authorities in the field [31], gives a pitiful treatment of what is called "human engineering" (which is what the term "usability" is referred to in the index). As the chapters concerned with specifying behavioral and non-behavioral requirements cover 140 pages, the potential reader may hope to learn something about specifying usability requirements. Instead, what he will find is a brief history of the devel-

27. The requirements attribute structure described in Paper IV represents an attempt to overcome this by distributing system qualities over functional requirements. How this mapping should be done is, however, a matter for further research.

opment from command language interfaces to WIMP, some comments about bad error messages, and a hint that user interface requirements should be included in terms of, e.g., example menus. Since this book probably was written in the late 1980s[28], before usability engineering had become widespread, this criticism may be unfair, but even in recent textbooks (e.g., [97][149][60]) usability and usability requirements are shown scant interest if any at all.

Where reference to usability requirements is made within RE literature, they are usually classified as a system-level requirements. Furthermore, they belong to the non-functional requirements, which are considered troublesome, in lack of a better word. Non-functional requirements are difficult to formalize, and usability seems to be more troublesome than most others. Even those who acknowledge that they are quantifiable and thus verifiable (as is actually stated in the IEEE Recommended Practice for Software Requirements Specifications, §5.2.1.2 [79]) seem to feel that they are inherently arbitrary (e.g., [97, p. 203]). They just don't seem to fit in. This is true even in cases where one could expect usability requirements to play an important role, which is illustrated by the following anectdote:

> At a recent conference on requirements engineering, I took a tutorial by the name "Bridging the Gap: From User Needs to Solution Definition". With my backround I was of course expecting to learn how requirements engineers (the teachers were experienced and renowned requirements engineers) suggest how one should consider usability issues. One of the presenters had continuous problems with the presentation software: There were two different ways of switching between editing mode and presentation (full-screen) mode, but they resulted in different slides being showed; either the first slide in the presentation *or* the one that was currently being edited. In one of the pauses that ensued from this I asked the presenters how their proposed method for bridging the gap between user needs and solution definitions would capture and eliminate this usability problem. The essence of the answer was, "I'm afraid it wouldn't".

Thus, from my point of view I can only agree with Parker *et al.* [129] who state that

28. It was first published in 1990.

> It is our contention that in requirements engineering, no established method for investigating and defining usability properties exist.

The lack of interest in usability issues in the RE literature does not only reflect the current state of affairs, but unfortunately it threatens to preserve the gap between the two fields [15]. RE is taught as a part of most software engineering programmes, and as long as the basic textbooks treat usability issues as insignificant, what can we expect from software engineers?

2.3.3 COMMON INTERESTS

To summarize the above discussion, the UE field would benefit from a better understanding of requirements, and the RE field would benefit from a better understanding of what usability is. Our users would benefit from both.

As an example of a specific area where RE could contribute to HCI is in requirements elicitation. Elicitation has always been at the heart of RE, and since elicitation activities involves interaction with stakeholders such as the real users, the methodology proposed overlaps with standard HCI procedures. Requirements elicitation is described by Davis[29] as [31, p. 42] :

> ...the activity that encompasses learning about the problem to be solved (often through brainstorming and/or questioning), understanding the needs of potential users, trying to find out who the user really is, and understanding all constraints on the solution.

which is a fairly good description of what usability engineers do most days of the week. Many RE textbooks devote a substantial treatment to elicitation techniques, and provide concrete guidance to various problem areas:

29. Davis uses the term “Problem Analysis” instead of requirements elicitation.

- Interview techniques
- Questionnaire design
- Scenario-based techniques
- Prototyping
- Interpersonal aspects (how to treat users)
- Background and organizational knowledge acquisition
- Knowledge structuring mechanisms (e.g., partitioning, abstraction, projection)
- Observation and social analysis
- Negotiation and conflict management

Sometimes these descriptions are more comprehensive and more concrete than what is usually found in the HCI literature, and could beneficially be used in many undergrad HCI courses. In return, something that is offered little space in the RE literature is task analysis. There are related techniques, such as scenario-based analysis (e.g., [154], see [159] for an overview), and use-case driven RE [135], which includes a user perspective to some extent, but social/organizational or cognitive perspectives are usually not considered. Instead, the structured software engineering approaches tend to have a system perspective, focussing on data and/or functions [4]. As Sutcliffe [152] points out:

> Social organisational [task analysis] methods are required to improve the early stages of analysis and to expand the scope beyond the software components of systems; while cognitive methods are necessary to ensure that interactive design takes people and their information processing limitations into account.

On a general note, user involvement beyond the elicitation phase is rarely seen within RE [129], perhaps due to the consanguinity between RE and traditional software engineering.

Another area where usability engineers and requirements engineers need to collaborate is in the transformation of design knowledge into good requirements. Most requirements engineers would agree that understanding user needs is fundamental to systems development. Usability engineers are experts in this area. Likewise, most usability engineers would agree that this understanding has to be mapped, in one way or another, to requirements, since they represent the language and culture of software engineering. Here, requirements engineers are the experts. Both seem to have some trouble with each other's area, and thus the need for a collaboration to work out feasible procedures is as evident as it is urgent.

As an example of a specific issue that usability engineers and requirements engineers need to deal with in the transformation of design knowledge into good requirements, we may consider an aspect of requirements quality which I have called objective proximity.

Objective proximity.

An aspect of requirement statements which should be of vital importance for the resulting system from a usability standpoint, is the question of how closely the requirement matches the *real* objective, rather than relying on *assumptions related* to the real objective. This is also noted by Jirotka and Goguen who state that the classical methods "*are largely concerned with notation for describing requirements, rather than with the nature of what is being described.*" [84, p. 4]. I have seen many examples of this in practice too. To illustrate this problem:

> Suppose that we are developing an administrative system for a telecom network. One of the main and most frequent tasks for the user is to set various parameters of type X. Thus, the "requirement" that the stakeholders have in mind at the outset is probably something along the line with *"It shall be easy to set parameters of type X"*. In the requirements document, this objective may be covered by a requirement at various conceptual distance from the real objective:
>
> **Type 1**: *It shall be easy for users of type U to set parameters of type X in context C.*
>
> This *is* the real objective, it focuses on the **interaction** between the user and the system, and it also represents a usability requirement as described in Paper II.
>
> **Type 2**: *Default values shall be provided for parameters of type X.*
> *Parameters of type X shall be clearly marked.*
> *The user interface shall comply with UI standard doc no 123*
>
> This is a little further from the real objective. The statement focuses on the **system** or **product**, and rests on the

assumption that "if we design the system according to these rules, it will be easy to set parameters of type X".

Type 3: *GUI-inspection shall be performed each week.*
This is yet a step away from the real objective, and focuses on the **process**. The statement rests on the assumption that "if we follow this system development method, it will be easy to set parameters of type X".

Type 4: *The development team shall represent the following skills: UI design, Human Factors, Java programming, Information design...*
This is very far away from the real objective. If focuses on the collective **competence** of the development team, and rests on the assumption that "if we have the right people on the team, it will be easy to set parameters of type X"

A requirement of type 1 is the closest representation of the real objective that could be stated in a requirements document. Given that further definitions of what is meant by "easy" (see *Setting goals and specifying usability* on page 33), "users of type U", and "context C" are given, it is also a *verifiable* requirement. Furthermore, it describes "what" should be achieved, rather than "how", because it is completely detached from design and implementation issues (in contrast with type 2). As an effect of this it is probably more *stable* than any other form. Thus, it has many of the important properties that a requirement statement should have (see *Good requirements* on page 46).

One may object that requirements like these are hard to verify until the system is finished, but 1) that is only partially true, since usability evaluations can be done on early prototypes, and 2) is that a good reason for neglecting the real objectives of the whole development effort?

In contrast, requirements of type 2 are product oriented. Such requirements are popular, because they comfortably describe a property of the object in the hands of the developers. They also seem to be easily verifiable –in the example, one only needs to put a mark in a checkbox if default values for parameters of type X are given. Quite often such design rules are collected in separate user interface design guidelines, and referred to from the requirements specification. However, several reports indicate that such design rules, especially in the form of guideline books, are hard to use for both formative and summative purposes (e.g., [70][33][155]). Furthermore, the product-oriented type of requirement does not necessarily lead to a better product from a usage perspective. In the above example, if the wrong default values are given, the product may be even harder to use. But the requirement statement would still be satisfied.

Even further away from the real objective are process requirements, type 3 in the above example. Although some literature advice against including such requirements in requirements specifications [78], such requirements are fairly common in other contractual documents. It is sometimes argued that process requirements are essentially all that is needed to make a successful product, since the process could define all necessary steps to ensure that the product is, in fact, successful (according to those terms defined during the process). While this may be true in a logical sense, other types of requirements are quite certainly elicited during the process. Requirements of this type are easy to state, but elusive, since they are difficult to verify; Very few real projects follow a given method to the point, and very few (practical) method descriptions or development situations are such that no improvisation is needed. Finally, given the best of all methods, it may still be problematic for user U to set parameter X.

As for type 4, competence requirements, these tend to be more on an organizational level, and are rarely seen in an actual requirements specification. As in the previous case, they are easily stated and their actual effect on the system rarely verified.

This is not meant to be an argument against requirements of type 2, 3 and 4. On the contrary, such requirements (or solutions, as they really are) are needed to actually accomplish a system with which a user of type U finds it easy to set parameters of type X. In fact, it is generally agreed that the skills of the individuals on a development project is the most decisive factor for the success of its end-result. But without the first type of requirement, representing the real objective, the development effort may still be a complete failure, even if the system complies with the specification to the letter.

On a concluding remark, note also that the form most likely to appear in a traditional requirements specification, namely "*It shall be possible to set parameters of type X*" in no way captures the real objective.

Towards a wider context

On a higher level, a common development within HCI and RE is also the movement towards a more contextual and holistic view. To illustrate this, we first take a look at the shift from usability engineering to contextual design within HCI, after which parallels are drawn from the RE field.

The article *Usability Engineering: Our Experience and Evolution,* by Whiteside, Bennett and Holzblatt [162], is one of the most frequently cited articles by proponents of the hard-core usability engineering approach. Whiteside and colleagues are recognized as some of the pioneers in that area. However, the latter part of the article (the *Evolution* part) reveals a shift of perspective, which is not as frequently quoted. To give a flavor of this shift of perspective, some quotes from this latter part are included.

> ... we sought to develop a set of measurable usability objectives that reflected [the developers'] understanding of usability attributes. We then helped develop products to meet those specifications. What typically resulted was a product that the developers saw as usable, since the product met their usability expectations. But how do we know that the developers' understanding of usability is the same as the users' understanding of usability? [...] If developers erroneously assume they know what constitutes a usable product for intended users, we may develop systems that "meet specifications" but that will not be usable to these people. [ibid., p. 805]

Evidently, the notion of usability specification is questioned. They then go on to question the usability tests, which are typically performed in laboratory settings:

> Consider the context of the laboratory experiment, as opposed to an actual work-place situation. The language used to describe experiments reveals the differing context. Experiments involve an experimenter and a subject. A subject is someone who is under the power, control, influence, observation, or direction of another person. Indeed, experimenters wield considerable power over subjects–they remove them from their usual social context, prescribe what work they are to do during the experiment, prescribe the time available for completing the work, minimize "external variables" such as interruptions (e.g. phone calls from home), and often attempt to conceal the

> details of why the experiment is being run (for fear of contaminating the data). This makes the experimental context radically different from the work-place context to which the experiment is nonetheless supposed to generalize. [ibid., p. 806]

The sheer underpinnings of usability engineering are thus questioned. It was stated earlier that once something is defined in operational terms, it is usually not difficult to measure (see *Setting goals and specifying usability* on page 33). The usability engineering approach rests heavily on such operational definitions, and admittedly, this is a weakness. Perhaps the concept of usability does not lend itself to such objectification and measurement? Is it even self-evident that an artifact could have such a property as usability, in the same way as it could be red, square, two metres high and have a heart on the door?[30] However it may be, Whiteside and Wixon [163], in contrast with the hard-core engineering approach as manifested in Good *et al.* [52], which Whiteside co-authored, now take a "softer" approach when defining usability:

> ... software usability refers to the extent to which software supports and enriches the ongoing experience of people who use that software. This direct emphasis on experience is at variance with "standard" definitions of usability that concentrate on, as primary, measurable manifestations of usability ...

The emerging approach was called "Contextual Design".

It is interesting to note that the two articles referred to above merely indicate a shift of perspective. No concrete advice for action is given. A couple of years after these papers, the procedures were still not concretized. Wixon *et al.* [164] present a summary of the procedures which could be characterized as "generally useful tips", such as "Be concrete, talk about what the user is doing, just did, or talk in the context of a work product." and "Summarize your understanding at the end of each session to determine who to talk to next and what to focus on next." Advice that just about any usability engineer would give to someone

30. If there are answers to these questions, I am not capable of providing them, but the reader is encouraged to read Dahlbom and Mathiassen's *Computers in context* [30] for a discussion relevant to computer science, and Pirsig's classical *Zen and the Art of Motorcycle Maintenance* [130] for a more general treatment of the topic.

who is about to make his or her first task analysis. It took another three years before more concrete advice was offered, this time by Holtzblatt and Beyer [76].

Contextual Design is a step closer to the user–"we want our feet to be sore where their shoes pinch", as Holtzblatt and Beyer put it, and a step towards a more holistic view of computer systems as part of a social organization. At the same time it represents a repudiation of the more structuralistic usability engineering methodology.

Turning now to the field of requirements engineering, it is interesting to see how a similar shift occurred at about the same time in this field. The absolute point in time, as well as the specific authors, are somewhat more difficult to establish than in the case of HCI[31], but an anthology by some of the most renowned authors within RE may be taken as a demarcating example: Requirements Engineering–Social and Technical issues [84], edited by Jirotka and Goguen. As in the case of the article by Whiteside et al quoted previously, this book contains two parts. The first part could be seen as state of the art from a traditional perspective, and the contributions are also original and by distinguished authors. The second part represents a shift of perspective, which can be illustrated by the following quotes from the first article in that part [50, p. 165-166]:

> Much of the information that requirements engineers need is embedded in the social worlds of users and managers [...] At its source, this information tends to be informal and highly dependent on its social context for interpretation. On the other hand, many representations that appear in constructing and using computer-based systems are formal [...] so that their interpretation is relatively independent of social context. [...]
>
> The need for progress in requirements engineering is acute [...]. The fact that the social issues at the root of many difficulties cannot be modelled by the usual technical methods suggest that novel approaches will be needed. Moreover the immaturity of the field suggests that a willingness to be eclectic rather than dogmatic could be valuable. Consequently, we are exploring

31. References are often made to, from an RE perspective, occasional studies, e.g., the early work by Mumford, Checkland, and the Scandinavian school of thought. This is actually also true in the case of contextual design.

> techniques from a variety of fields, especially ethnomethodology and discourse analysis [...]

Thus, acknowledging the problems related to the traditional narrow outlook, ethnomethodology, discourse analysis and even semiotics has then been proposed as alternative approaches to the RE community, in order to capture a wider context.

On a final note, it is a fascinating exercise to compare two separate but related fields, and there are many interesting reflections to make–both in the small and in the large. As an example of the former, both fields have their favorite standard justification examples. In the case of RE this is the report by the Government Accounting Office [44], published in 1979, and which showed that out of $7 million spent by the U.S. Federal Government on nine contracts, only $119K resulted in software that met its requirements[32]. This is used as an argument for the importance of RE as much and as frequently as the study by Karat [87], is used within UE. Karat showed that usability engineering could save $6,800,000 for a large telecommunications operator, and studies like these are of course eagerly adopted and widely spread by a community starving for quantitative justification.

As an example of the latter, on a paradigmatic level it appears as if requirements engineering as a separate field has gone through many of the phenomena one has seen within HCI, at about the same time. The first textbooks came in the early 1980s, and then the quest for common definitions resulted in the first standards in the early 90s, for example. As a consequence of the proliferation of methods, researchers found a

32. A matter of curiosity: It was shown already in 1991 (as cited in [46][170]) that the conclusions drawn from the report were quite questionable, since the GAO's numbers were based on contracts selected because *they were already in some form of trouble, such as litigation*. Despite this, the tale goes on and on, and has survived for two decades (as it appears in a reprint of [98], in [156]).
I have not been able to obtain the GAO report, but it is so ubiquitously cited, that I suspect that I will learn nothing more from it when I do. Except possibly for the exact number of projects investigated, which differs among the citations. Which in turn may indicate that I'm not the only one who has had trouble finding the original report.

need to create classification schemes (e.g., [74][77]), first based on products or processes (“which method is best?”, or perhaps “whose method is the best?”) and then on ideological or philosophical grounds (e.g., [7][105]). The “revolution” against the formal and narrow-minded view of users came somewhat later within RE, but could be traced to the early 1980s, with Checkland’s “soft system approach”. One can wonder about this synchronicity; is it because the two fields engage the same people to a great extent, or the same *kind* of people, or is it due to general movements in society?

References

[1] ALLWOOD, C. M. *Människa-datorinteraktion [Human-computer interaction]*. Studentlitteratur, Lund, Sweden, 1991. (In Swedish).

[2] BELLOTTI, V. Implications of current design practice for the use of HCI techniques. In *People and Computers IV, Proc. 4th Conf. British Computer Society, HCI Specialist Group* (1988), D. M. Jones and R. Winder, Eds., pp. 13–34.

[3] BENBASAT, I., GOLDSTEIN, D. K., AND MEAD, M. The case research strategy in studies of information systems. *MIS Quarterly* (Sept. 1987), 369–386.

[4] BENYON, D. The role of task analysis in systems design. *Interacting with computers 4*, 1 (1992), 102–123.

[5] BERTAGGIA, N., MONTAGNINI, G., NOVARA, F., AND PARLANGELI, O. Product usability. In *Methods and Tools in User-Centered Design for Information Technology*, M. Galer, S. Harker, and J. Ziegler, Eds., vol. 9 of *Human Factors in Information Technology*. North-Holland, Amsterdam, 1992, ch. 5.

[6] BIAS, R. G., AND MAYHEW, D. J., Eds. *Cost-Justifying Usability*. Academic Press, Boston, MA, 1994.

[7] BICKERTON, M., AND SIDDIQI, J. The classification of requirements engineering methods. In *Proc. Int. Symp. Req. Engineering* (Los Alamitos, California, 1993), IEEE CS Press, pp. 182–186.

[8] BOEHM, B., AND IN, H. Identifying quality requirement conflicts. *IEEE Software 13*, 2 (March 1996), 25–35.

[9] BOEHM, B., AND IN, H. Conflict analysis and negotiation aids for cost-quality requirements. *Software Quality Professional 1*, 1 (1998), 1–9.

[10] BOEHM, B. W. Software engineering. *IEEE Transactions on Computers* (December 1976), 1266–41.

[11] BOEHM, B. W. Guidelines for verifying and validating software requirements and design specifications. In *IFIP 1979*, P. A. Samet, Ed. North-Holland, 1979, pp. 711–719.

[12] BOEHM, B. W. *Software engineering economics*. Prentice-Hall Inc., Englewood Cliffs, N.J., 1981.

[13] BOEHM, B. W. A spiral model of software development and enhancement. *IEEE Computer 21*, 5 (1988), 61–72.

[14] BOOTH, P. A. *An introduction to human-computer interaction*. Lawrence Erlbaum Associates, Hove, UK, 1989.

[15] BROWN, J. Exploring human-computer interaction and software engineering methodologies for the creation of interactive software. *SIGCHI Bulletin 29*, 1 (January 1997), 32–35.

[16] BUBENKO, J. A. Challenges in requirements engineering. In *Proceedings: 2nd IEEE International Symposium on Requirements Engineering* (1995), IEEE Computer Society Press, pp. 160–162.

[17] BURGESS, R. G. *In the field. An introduction to field research*. Contemporary social research. Routledge, London, UK, 1991. Originally published 1984 by Unwin Hyman Ltd.

[18] BYRD, T. A., COSSICK, K. L., AND ZMUD, R. W. A synthesis of research on requirements analysis and knowledge acquisition techniques. *MIS Quarterly 16*, 1 (1992), 117–138.

[19] BYRNE, M. D., WOOD, S. D., SUKAVIRIYA, P., FOLEY, J. D., AND KIERAS, D. E. Automating interface evaluation. In *Proceedings of CHI'94 Conference on Human Factors in Computing Systems* (1994), pp. 232–237.

[20] CARD, S. K., MORAN, P., T., AND NEWELL, A. *The psychology of human computer interaction*. Lawrence Erlbaum, Hillsdale, NJ, 1983.

[21] CARLSHAMRE, P. *A Collaborative Approach to Usability Engineering: Technical Communicators and System Developers in Usability-Oriented Development*. Licentiate thesis no. 455, Linköping University, S-581 83 Sweden, 1994.

[22] CARMEL, E., AND BECKER, S. A process model for packaged software development. *IEEE Trans. on Eng. Manag. 42*, 1 (February 1995).

[23] CARROLL, J. M., AND ROSSON, M. B. Usability specifications as a tool in iterative development. In *Advances in Human-Computer Interaction*, H. R. Hartson, Ed. Ablex, Norwood, NJ, 1985, ch. 1, pp. 1–28.

[24] CHECKLAND, P. From framework through experience to learning: The essential nature of action research. In *Information systems research: Contemporary approaches and emergent traditions*, H.-E. Nissen, H. K. Klein, and R. Hirschheim, Eds. Elsevier Science Publishers B.V. (North-Holland), 1991, pp. 397–403.

[25] CHISHOLM, R. M. Improving the management of technical writers: Creating a context for usable documentation. In *Effective documentation: What we have learned from research*, S. Doheny-Farina, Ed. The MIT Press, Cambridge, MA, 1988, ch. 14.

[26] CLEMENT, A., AND BESSELAAR, P. V. A retrospective look at PD projects. *Communications of the ACM 36*, 6 (1993), 29–37.

[27] COBLE, J. M., KARAT, J., AND KAHN, M. G. Maintaining a focus on user requirements throughout the development of clinical workstation software. In *Proceedings of ACM CHI 97 Conference on Human Factors in Computing Systems* (1997), vol. 1 of *PAPERS: Bringing Users Into Design*, pp. 170–177.

[28] COX, K., AND WALKER, D. *User-interface design*, 2nd ed. Prentice Hall, 1993, ch. 7.

[29] CURTIS, B., KRASNER, H., AND ISCOE, N. A field study of the software design process for large systems. *Communications of the ACM 31*, 11 (Nov. 1988), 1268–1287.

[30] DAHLBOM, B., AND MATHIASSEN, L. *Computers in context*. NCC Blackwell, Oxford, UK, 1993.

[31] DAVIS, A. *Software Requirements: Objects Functions and States*. Prentice Hall, Englewood Cliffs, NJ, 1993.

[32] DAVIS, A., OVERMYER, S., JORDAN, K., CARUSO, J., DANDASHI, F., DINH, A., KINCAID, G., LEDEBOER, G., REYNOLDS, P., SITARAM, P., TA, A., AND THEOFANOS, M. Identifying and measuring quality in a software requirements specification. In *Software Requirements Engineering, 2nd Ed.*, R. Thayer and M. Dorfman, Eds. IEEE Computer Society, 2000, pp. 194–205.

[33] DE SOUZA, F., AND BEVAN, N. The use of guidelines in menu interface design: Evaluation of a draft standard. In *Proceedings of IFIP Interact'90* (1990), Elsevier Science, pp. 435–440.

[34] DIAPER, D. Integrating HCI and software engineering requirements analysis; a demonstration of task analysis supporting entity modeling. *SIGCHI Bulletin 29*, 1 (January 1997), 41–50.

[35] DIAPER, D., AND ADDISON, M. Task analysis and systems analysis for software development. *Interacting with computers 4*, 1 (1992), 124–139.

[36] DIX, A., FINLAY, J., ABOWD, G., AND BEALE, R. *Human-computer interaction*. Prentice Hall, 1993.

[37] DORFMAN, M. Requirements engineering. In *Software Requirements Engineering*, R. Thayer and M. Dorfman, Eds. IEEE Computer Society Press, Los Alamitos, CA, 2000, pp. 7–22.

[38] EASTERBROOK, S., AND NUSEIBEH, B. Using ViewPoints for inconsistency management. *Software Engineering Journal 11*, 1 (January 1996), 31–43.

[39] ERICSSON, M. *Supporting the Use of Design Knowledge: An Assessment of Commenting Agents*. PhD thesis, Dissertation no. 592, Linköping University, 1999.

[40] FENTON, N., PFLEEGER, S. L., AND GLASS, R. L. Science and substance: a challenge to software engineers. *IEEE Software 11*, 4 (July 1994), 86–95.

[41] FINKELSTEIN, A., KRAMER, J., NUSEIBEH, B., AND GOEDICKE, M. Using ViewPoints for inconsistency management. *Int. Journal of Software Engineering and Knowledge Engineering 2*, 1 (1992).

[42] FRANZ, C. R., AND ROBEY, D. An investigation of user-led system design: Rational and political perspectives. *Communications of the ACM 27*, 12 (December 1984), 1202–1209.

[43] FRØKJÆR, E., HERTZUM, M., AND HORNBÆK, K. Measuring usability: are effectiveness, efficiency, and satisfaction really correlated ? In *Proceedings of the 2000 Conference on Human Factors in Computing Systems (CHI-00)* (N. Y., April 1–6 2000), T. Turner, G. Szwillus, M. Czerwinski, and P. Fabio, Eds., ACM Press, pp. 345–352.

[44] GAO. *Contracting for Computer Software Development-Serious Problems Require Management Attention to Avoid Wasting Additional Millions. Report FGMSD-80-4.* U.S. Government Accounting Office, November 1979.

[45] GILB, T. Quantifying the qualitative. http://www.result-planning.com; Accessed in June 2001, Sept. 1997.

[46] GLASS, R. L. The software-research crisis. *IEEE Software 11*, 6 (Nov. 1994), 42–47.

[47] GOEDICKE, M., MEYER, T., AND TAENTZER, G. View point-oriented software development by distributed graph transformation: Towards a basis for living with inconsistencies. In *Proceedings: 4th IEEE International Symposium on Requirements Engineering* (1999), IEEE Computer Society Press, pp. 92–99.

[48] GOGUEN, J. The dry and the wet. In *Information Systems Concepts: Improving the Understanding. Proc. IFIP Working Group 8.1 Conference.* (Holland, 1992), Elsevier, pp. 1–17.

[49] GOGUEN, J., AND LINDE, C. Techniques for requirements elicitation. In *Proc. International Symposium of Requirements Engineering* (1993), IEEE, pp. 152–164.

[50] GOGUEN, J. A. Requirements engineering as the reconciliation of social and technical issues. In *Requirements Engineering; Social and Technical Issues*, M. Jirotka and J. A. Goguen, Eds. Academic Press, 1994, pp. 165–199.

[51] GOLDKUHL, G. Contextual activity modelling of information systems. Tech. Rep. LiTH-IDA-R-93-05, Dept. Computer and Information Science, Linköping University, Sweden, Mar. 1993.

[52] GOOD, M., SPINE, T. M., WHITESIDE, J., AND GEORGE, P. User-derived impact analysis as a tool for usability engineering. In *Proceedings of CHI'86 Conference on Human Factors in Computing Systems. New York: ACM Press* (April 1986).

[53] GOODWIN, N. C. Functionality and usability. *Communications of the ACM 30*, 3 (1987), 229–233.

[54] GOTEL, O. C. Z., AND FINKELSTEIN, A. C. W. An analysis of the requirements traceability problem. In *Proceedings: 1st International Conference on Requirements Engineering* (1994), IEEE Computer Society Press, pp. 94–101.

[55] GOULD, J. How to design usable systems. In *Handbook of Human-Computer Interaction*, M. Helander, Ed. Elsevier Science Publishers (North-Holland), 1988, pp. 757–789.

[56] GOULD, J., BOIES, S., AND LEWIS, C. Making usable, useful, productivity-enhancing computer applications. *Communications of the ACM 34*, 1 (1991), 74–85.

[57] GOULD, J., LEWIS, C., AND BARNES, V. Effects of cursor speed on text-editing. In *Proceedings of CHI'85 Conference on Human Factors in Computing Systems* (1985).

[58] GOULD, J. D., AND LEWIS, C. Designing for usability: Key principles and what designers think. *Communications of the ACM 28*, 3 (Mar. 1985), 300–311.

[59] GRAHAM, D. R. Incremental development and delivery for large software systems. In *Proc. Software Engineering for Large Software Systems* (London, 1989), pp. 156–195. Also available from http://home2.swipnet.se/~w-26311/evo-gilb/graham1.htm.

[60] GRAHAM, I. *Requirements Engineering and Rapid Development*. Addison-Wesley, Harlow, England, 1998.

[61] GRAY, W. D., JOHN, B. E., AND ATWOOD, M. E. The precis of project Ernestine, or an overview of a validation of GOMS. In *Proceedings of CHI'92 Conference on Human Factors in Computing Systems* (1992), pp. 307–312.

[62] GREENBAUM, J., AND KYNG, M. Introduction: Situated design. In *Design at work: Cooperative design of computer systems*, J. Greenbaum and M. Kyng, Eds. Lawrence Erlbaum Ass., Hillsdale, NJ, 1991, ch. 1.

[63] GREER, D., BUSTARD, D. W., AND SUNAZUKA, T. Prioritisation of system changes using cost-benefit and risk assessments. In *Proceedings: 4th IEEE International Symposium on Requirements Engineering* (1999), IEEE Computer Society Press, pp. 180–188.

[64] GRICE, R. A. Information development is part of product development–not an afterthought. In *Text, ConText, and HyperText*, E. Barrett, Ed. The MIT Press, Cambridge, MA, 1988, ch. 8.

[65] GRUDIN, J. Interactive systems: Bridging the gaps between developers and users. *IEEE Computer 24*, 4 (April 1991), 59–69.

[66] GRUDIN, J. Utility and usability: research issues and development contexts. *Interacting with Computers 4*, 2 (1992), 209–217.

[67] GRUDIN, J., AND MACLEAN, A. Adapting a psychophysical method to measure performance and preference tradeoffs in human-computer interaction. In *Human-Computer Interaction - INTERACT'84* (1985), B. Shackel, Ed., Elsevier Science Publishers (North-Holland), pp. 737–741.

[68] GRUDIN, J., AND POLTROCK, S. E. User interface design in large corporations: Coordination and communication across disciplines. In *Proceedings of CHI'89 Conference on Human Factors in Computing Systems. New York: ACM Press* (1989), pp. 197–203.

[69] GUMMESSON, E. *Qualitative methods in management research*. Chartwell-Bratt, Bromley, UK, 1988.

[70] HAMMOND, N., GARDINER, M., CHRISTIE, B., AND MARSHALL, C. The role of cognitive psychology in user-interface design. In *Applying cognitive psychology to user-interface design*, M. Gardiner and B. Christie, Eds. John Wiley & Sons, Chichester, 1987, ch. 2, pp. 13–53.

[71] HANSEN, W. J. User engineering principles for interactive systems. In *Proc. Fall Joint Computer Conference* (1971), AFIPS Press, Montvale, NJ, pp. 523–532.

[72] HARWELL, R., ASLAKSEN, E., HOOKS, I., MENGOT, R., AND PTACK, K. What is a requirement? In *Software Requirements Engineering*, R. Thayer and M. Dorfman, Eds. IEEE Computer Society, Los Alamitos, CA., 2000, pp. 23–29.

[73] HEWETT, BAECKER, CARD, CAREY, GASEN, MANTEI, PERLMAN, STRONG, AND VERPLANK. ACM SIGCHI Curricula for Human-Computer Interaction, 1992. Obtainable from http://www.acm.org/sigchi/.

[74] HOFMANN, H. F. Requirements engineering: A survey of methods and tools. Tech. Rep. 93.05, Institut für Informatik der Universität Zürich, 1993.

[75] HOLCOMB, R., AND THARP, A. What users say about software usability. *Int. Journal of Human-Computer Interaction 3*, 1 (1991), 49–78.

[76] HOLTZBLATT, K., AND BEYER, H. Making customer-centered design work for teams. *Communications of the ACM 36*, 10 (1993), 93–103.

[77] HUGHES, K. J., RANKIN, R. M., AND SENNETT, C. T. Taxonomy for requirements analysis. In *Proceedings: 1st International Conference on Requirements Engineering* (1994), IEEE Computer Society Press, pp. 176–179.

[78] IEEE STD 1233-1998. IEEE guide for devloping system requirements specifications. In *Software Requirements Engineering, Second Edition*, R. H. Thayer and M. Dorfman, Eds. IEEE Computer Society, Los Alamitos, CA., 2000, pp. 245–280.

[79] IEEE STD 830-1998. IEEE recommended practice for software requirements specifications. In *Software Requirements Engineering, Second Edition*, R. H. Thayer and M. Dorfman, Eds. IEEE Computer Society, Los Alamitos, CA., 2000, pp. 207–244.

[80] ISO 9241-11:1998. Ergonomic requirements for office work with visual display terminals (VDTs)– part 11: Guidance on usability, 1998.

[81] JACKSON, D. Counterpoint: Requirements need form, maybe formality. *IEEE Software 13*, 2 (Mar. 1996), 21–22.

[82] JACKSON, M. *Software Requirements & Specifications; a Lexicon of Practice, Principles and Prejudices.* Addison-Wesley, 1995.

[83] JEFFRIES, R., MILLER, J. R., WHARTON, C., AND UYEDA, K. M. User interface evaluation in the real world: A comparison of four techniques. In *Proceedings of CHI'91 Conference on Human Factors in Computing Systems* (1991), pp. 119–124.

[84] JIROTKA, M., AND GOGUEN, J. A., Eds. *Requirements Engineering; Social and Technical Issues.* Academic Press, London, 1994.

[85] JUNG, H.-W. Optimizing value and cost in requirements analysis. *IEEE Software 15*, 4 (July/August 1998), 74–78.

[86] KAMSTIES, E., HÖRMANN, K., AND SCHLICH, M. Requirements engineering in small and medium enterprises. *Requirements Engineering Journal 3* (1998), 84–90.

[87] KARAT, C.-M. Cost-benefit analysis of iterative usability testing. In *Human Computer Interaction – INTERACT '90* (1990), D. Diaper, Ed., Elsevier Science Publishers, B. V. (North-Holland), pp. 351–356.

[88] KARAT, C.-M. Cost-justifying human factors support on software development projects. *Human Factors Society Bulletin 35*, 11 (Nov. 1992), 1–4.

[89] KARAT, C.-M., CAMPBELL, R., AND FIEGEL, T. Comparison of empirical testing and walkthrough methods in user interface evaluation. In *Proceedings of CHI'92 Conference on Human Factors in Computing Systems* (1992), pp. 397–404.

[90] KARLSSON, J. *Towards a Strategy for Software Requirements Selection*. Licentiate thesis no. 513, Dept. Computer Science, Linköping University, 1995.

[91] KARLSSON, J. *A Systematic Approach to Prioritizing Software Requieremtns.* PhD thesis, Dissertation No. 526, Linköping University, 1998.

[92] KARLSSON, J., AND RYAN, K. A cost-value approach for prioritizing requirements. *IEEE Software 14*, 5 (1997), 67–74.

[93] KARLSSON, J., AND RYAN, K. Improved practical support for large-scale requirements prioritising. *Requirements Engineering 2*, 1 (1997), 51–60.

[94] KEEP, J., AND JOHNSON, H. HCI and requirements engineering: Generating requirements in a courier despatch management system. *SIGCHI Bulletin 29* (January 1997), 51–53.

[95] KEIL, M., AND CARMEL, E. Customer-developer links in software development. *Communications of the ACM 38*, 5 (May 1995), 33–44.

[96] KELLER, S. E., KAHN, L. G., AND PANARA, R. B. Specifying software quality requirements with metrics. In *System and Software Requirements Engineering*, R. H. Thayer and M. Dorfman, Eds. IEEE Computer Society Press, 1990, pp. 145–163.

[97] KOTONYA, G., AND SOMMERVILLE, I. *Requirements Engineering, Processes and Techniques*. John Wiley & Sons Ltd, West Sussex, England, 1988.

[98] KOTONYA, G., AND SOMMERVILLE, I. Requirements engineering with viewpoints. *Software Engineering Journal 11*, 1 (January 1996), 5–18.

[99] LAWRENCE, B. Point: Do you really need formal requirements? *IEEE Software 13*, 2 (Mar. 1996), 20, 22.

[100] LAWRENCE, B. Counterpoint: Designers must do the modeling. *IEEE Software 15*, 2 (Mar.slash April 1998), 31, 33.

[101] LEVESON, N. G. Intent specifications: An approach to building human-centered specifications. *IEEE Transactions on Software Engineering 26*, 1 (Jan. 2000), 15–35.

[102] LEWIS, C., AND RIEMAN, J. *Task-Centered User Interface Design: A practical introduction*. Shareware, obtainable via ftp from ftp.cs.colorado.edu, 1993.

[103] LOUCOPOULOS, P., AND KARAKOSTAS, V. *System Requirements Engineering*. McGraw-Hill, UK, 1995.

[104] LÖWGREN, J. *Human-computer interaction*. Studentlitteratur, Lund, Sweden, 1993.

[105] LÖWGREN, J. Perspectives on usability. Tech. Rep. LiTH-IDA-R-95-23, Dept. of Computer Science, Linköping University., 1995.

[106] LÖWGREN, J., AND LAURÉN, U. Supporting the use of guidelines and style guides in professional user interface design. *Interacting with Computers 5*, 4 (1993), 385–396.

[107] LÖWGREN, J., AND NORDQVIST, T. A knowledge-based tool for user interface evaluation and its integration in a UIMS. In *Human-Computer Interaction — INTERACT'90* (1990), Diaper et al., Ed., pp. 395–400.

[108] LÖWGREN, J., AND NORDQVIST, T. Knowledge-based evaluation as design support for graphical user interfaces. In *Proceedings of CHI'92 Conference on Human Factors in Computing Systems* (1992), pp. 181–188.

[109] LÖWGREN, J., AND STOLTERMAN, E. *Design Av Informationsteknik: Materialet Utan Gränser [Designing Information Technology: The Material Without Properties.]*. Studentlitteratur, Lund, 1998. In Swedish.

[110] LUBARS, M., POTTS, C., AND RICHTER, C. A review of the state of the practice in requirements modelling. In *Proceedings of IEEE International Symposium on Requirements Engineering, RE'93* (San Diego, California, Jan. 1993), IEEE Computer Society Press, pp. 2–14.

[111] MACAULAY, L. A. Seven-layer model of the role of the facilitator in requirements engineering. *Requirements Engineering 4*, 1 (1999), 38–59.

[112] MACRO, A. *Software Engineering; Concepts and Management*. Prentice Hall, UK, 1990.

[113] MAGUIRE, M., AND SWEENEY, M. System monitoring: garbage generator or basis for comprehensive evaluation system? In *People and Computers V. Proc. Fifth Conference of the British Computer Society Human-Computer Interaction Specialist Group* (1989), pp. 375–394.

[114] MANTEI, M. M., AND TEOREY, T. J. Cost/benefit analysis for incorporating human factors in the software lifecycle. *Communications of the ACM 31*, 4 (April 1988).

[115] MARCINIAK, J. J., Ed. *Encyclopedia of Software Engineering*, vol. 2. Wiley & Sons, 1994.

[116] MARSHALL, C., AND ROSSMAN, G. B. *Designing qualitative research*. Sage Publications, Newbury Park, CA, 1989.

[117] MARTELLO, S., AND TOTH, P. *Knapsack Problems: Algorithms and Computer Implementations*. Wiley, Chichester, 1990.

[118] MENZIES, T., EASTERBROOK, S., NUSEIBEH, B., AND WAUGH, S. An empirical investigation of multiple viewpoint reasoning in requirements engineering. In *Proceedings: 4th IEEE International Symposium on Requirements Engineering* (1999), IEEE Computer Society Press, pp. 100–110.

[119] MOLICH, R., AND NIELSEN, J. Improving a human-computer dialogue. *Communications of the ACM 33*, 3 (1990), 338–348.

[120] NIELSEN, J. Usability engineering at a discount. In *Proceedings of Third International Conference on Human-Computer Interaction* (Boston, MA, Sept. 1989), pp. 2–8.

[121] NIELSEN, J. Big paybacks from discount usability engineering. *IEEE Software 7*, 3 (1990), 107–108.

[122] NIELSEN, J. Finding usability problems through heuristic evaluation. In *Proceedings of CHI'92 Conference on Human Factors in Computing Systems* (1992), pp. 373–380.

[123] NIELSEN, J. *Usability Engineering*. AP Professional, Cambridge, MA, 1993.

[124] NIELSEN, J., AND LANDAUER, T. K. A mathematical model of the finding of usability problems. In *Proceedings of INTERCHI'93: Human Factors in Computing Systems* (1993), pp. 206–213.

[125] NIELSEN, J., AND PHILLIPS, V. L. Estimating the relative usability of two interfaces: Heuristic, formal, and empirical methods compared. In *Proceedings of INTERCHI'93: Human Factors in Computing Systems* (1993), pp. 214–221.

[126] NOVORITA, R. J., AND GRUBE, G. Benefits of structured requirements methods for market-based enterprises. In *Proceedings of International Council on Systems Engineering Sixth Annual International Symposium on Systems Engineering: Practice and Tools (INCOSE96)* (Boston USA, July 1996).

[127] ÖDMAN, P. *Tolkning, förståelse, vetande — Hermeneutik i teori och praktik. [Interpretation, understanding, knowledge — The theory and practice of hermeneutics]*. Almqvist & Wiksell Förlag AB, Stockholm, 1979. (In Swedish).

[128] PALMER, J. Traceability. In *Software Requirements Engineering*, R. Thayer and M. Dorfman, Eds. IEEE Computer Society Press, Los Alamitos, CA, 1997, pp. 364–374.

[129] PARKER, H., ROAST, C., AND SIDDIQI, J. Towards a framework for investigating temporal properties in interaction. *SIGCHI Bulletin 29*, 1 (January 1997), 56–60.

[130] PIRSIG, R. M. *Zen and the art of motorcycle maintenance*. The Bodley Head, Great Britain, 1974.

[131] POHL, K. The three dimensions of requirements engineering. In *5th International Conference on Advanced Information Systems Engineering* (1993), C. Rolland, F. Bodart, and C. Cauvet, Eds., Springer Verlag, pp. 275–292.

[132] POTTS, C. Software-engineering research revisited. *IEEE Software 10*, 5 (1993), 19–28.

[133] POTTS, C. Invented requirements and imagined customers: Requirements engineering for off-the-shelf software. In *Proceedings: 2nd IEEE International Symposium on Requirements Engineering* (1995), IEEE Computer Society Press, pp. 128–130.

[134] PREECE, J., ROGERS, Y., BENYON, D., HOLLAND, S., AND CAREY, T. *Human-Computer Interaction*. Addison-Wesley, 1994.

[135] REGNELL, B. *Requirements Engineering with Use Cases - a Basis for Software Development*. PhD thesis, Lund University, Sweden, 1999. ISRN LUTEDX/TETS–1040–SE+225P.

[136] REGNELL, B., BEREMARK, P., AND EKLUNDH, O. A market-driven requirements engineering process; results from an industrial process improvement programme. *Requirements Engineering Journal 3*, 2 (1998), 121–129.

[137] REGNELL, B., HÖST, M., NATT OCH DAG, J., BEREMARK, P., AND HJELM, T. An industrial case study on distributed prioritization in market-driven requirements engineering for packaged software. *Requirements Engineering Journal 6*, 1 (2000), 51–62.

[138] ROAST, C. Specifying cognitive interface requirements. *SIGCHI Bulletin 29*, 1 (January 1997), 66–57.

[139] ROBINSON, W. N., PAWLOWSKI, S. D., AND VOLKOV, V. Requirements interaction management. Tech. Rep. GSU CIS Working Paper 99-7, Dept. of Computer Information Systems, Georgia State University, Atlanta, GA 30302, 1999. Accessed May 2001: http://cis.gsu.edu/~wrobinso.

[140] ROMAN, G. C. A taxonomy of current issues in requirements engineering. *IEEE Computer* (April 1985), 14–22.

[141] ROWLEY, D. E. Usability testing in the field: Bringing the laboratory to the user. In *Proceedings of CHI'94 Conference on Human Factors in Computing Systems* (1994), pp. 252–257.

[142] ROWLEY, D. E., AND RHOADES, D. G. The cognitive jogthrough: A fast-paced user interface evaluation technique. In *Proceedings of CHI'92 Conference on Human Factors in Computing Systems* (1992), pp. 389–395.

[143] ROYCE, W. W. Managing the development of large software systems: Concepts and techniques. In *Proc. Wescon* (August 1970). Also available in Proc. ICSE9, Computer Society Press, 1987.

[144] SASSE, M. A., CUNNINGHAM, J., AND WINDER, R. Preface: Maturing nicely. In *People and Computers XI, Proceedings of HCI 96* (London, 1996), M. A. Sasse, J. Cunningham, and R. Winder, Eds., Springer, pp. ix–xi.

[145] SHNEIDERMAN, B. *Designing the user interface*, 2nd ed. Addison Wesley, 1992.

[146] SIDDIQI, J. Challenging universal truths of requirements engineering. *IEEE Software 11*, 2 (Mar. 1994), 18–19.

[147] SIMES, D. K., AND SIRSKY, P. A. Human factors: An exploration of the psychology of human-computer dialogues. In *Advances in Human-Computer Interaction*, H. R. Hartson, Ed. Ablex, Norwood, NJ, Norwood, NJ, 1985, ch. 1, pp. 1–28.

[148] SMITH, S., AND MOSIER, J. Guidelines for designing user interface software. Tech. Rep. ESD-TR-86-278, Mitre Corp., Bedford, MA, 1986.

[149] SOMMERVILLE, I., AND SAWYER, P. *Requirements Engineering: A Good Practice Guide*. Wiley & Sons, West Sussex, England, 1997.

[150] SPENCER, R. H. *Computer usability testing and evaluation*. Prentice-Hall, Englewood Cliffs, New Jersey 07632, 1985.

[151] SPIVEY, J. M. An introduction to Z and formal specifications. *Software Engineering Journal 4*, 1 (1990), 40–50.

[152] SUTCLIFFE, A. Task analysis, systems analysis and design: Symbiosis or synthesis? *Interacting with Computers 1*, 1 (1989), 6–12.

[153] SUTCLIFFE, A. A technique combination approach to requirements engineering. In *Proceedings: 3rd IEEE International Symposium on Requirements Engineering* (1997), IEEE Computer Society Press, pp. 65–77.

[154] SUTCLIFFE, A. G., AND RYAN, M. Experience with SCRAM, a SCenario Requirements Analysis Method. In *Proceedings: 3rd International Conference on Requirements Engineering* (1998), IEEE Computer Society Press, pp. 164–173.

[155] TETZLAFF, L., AND SCHWARTZ, D. R. The use of guidelines in interface design. In *Human Factors in Computing Systems (CHI'91 Proceedings)* (1991), pp. 329–333.

[156] THAYER, R. H., AND DORFMAN, M., Eds. *Software Requirements Engineering, Second Edition*. IEEE Computer Society, Los Alamitos, CA., 2000.

[157] THOMAS, J. C., AND KELLOGG, W. A. Minimizing ecological gaps in interface design. *IEEE Software 6*, 1 (1989), 78–86.

[158] TICHY, W. F., LUKOWICZ, P., PRECHELT, L., AND HEINZ, E. A. Experimental evaluation in computer science: A quantitative study. *Journal of Systems Software 28* (1995), 9–18.

[159] WEIDENHAUPT, K., POHL, K., JARKE, M., AND HAUMER, P. Scenarios in system development: Current practice. *IEEE Software 15*, 2 (Mar.slash April 1998), 34–45.

[160] WEISS, E. H. Breaking the grip of user manuals. *ACM SIGDOC Asterisk 14*, 2 (1988), 4–10.

[161] WHARTON, C., BRADFORD, J., JEFFRIES, R., AND FRANZKE, M. Applying cognitive walkthroughs to more complex user interfaces: Experiences, issues, and recommendations. In *Proceedings of CHI'92 Conference on Human Factors in Computing Systems* (1992), pp. 381–388.

[162] WHITESIDE, J., BENNETT, J., AND HOLTZBLATT, K. Usability engineering: Our experience and evolution. In *Handbook of Human-Computer Interaction*, M. Helander, Ed. Elsevier Science Publishers (North-Holland), 1988, pp. 791–817.

[163] WHITESIDE, J., AND WIXON, D. The dialectic of usability engineering. In *Human Computer Interaction – INTERACT '87* (1987), H.-J. Bullinger and B. Shackel, Eds., Elsevier Science Publishers, B. V. (North-Holland), pp. 17–20.

[164] WIXON, D., HOLTZBLATT, K., AND KNOX, S. Contextual design: An emergent view of system design. In *Proceedings of CHI'90 Conference on Human Factors in Computing Systems* (April 1990), pp. 329–336.

[165] WRIGHT, P., FIELDS, B., AND HARRISON, M. Deriving human-error tolerance requirements from tasks. In *Proceedings of the IEEE International Conference on Requirements Engineering* (1994), IEEE Computer Society Press.

[166] YADAV, S. B., BRAVOCCO, R. R., CHATFIELD, A. T., AND RAJKUMAR, T. M. Comparison of analysis techniques for information requirement determination. *Communications of the ACM, CACM 31*, 9 (Sept. 1988), 1090–1097.

[167] YEH, A. Requirements engineering support technique (REQUEST); a market driven requirements management process. In *Proceedings of Second Symposium of Quality Software Development Tools* (May 1992), IEEE Computer Society Press, pp. 211–223.

[168] ZAVE, P. An insider's evaluation of PAISLey. *IEEE Transactions on Software Engineering 17*, 3 (March 1991), 212–225.

[169] ZELKOWITZ, M. V., AND WALLACE, D. Computing practices: Experimental models for validating technology. *Computer 31*, 5 (May 1998), 23–31.

[170] ZVEGINTZOV, N. Frequently begged questions and how to answer them. *IEEE Software 15*, 2 (March/April 1998).