SUPPORTING THE COOPERATIVE DESIGN PROCESS OF END-USER TAILORING

Jeanette Eriksson

Blekinge Institute of Technology
Doctoral Dissertation Series No. 2008:03
School of Engineering

Supporting the cooperative design process of end-user tailoring

Jeanette Eriksson

Blekinge Institute of Technology Doctoral Dissertation Series
No 2008:03
ISSN 1653-2090
ISBN 978-91-7295-130-3

Supporting the cooperative design process of end-user tailoring

Jeanette Eriksson

Department of Interaction and System Design
School of Engineering
Blekinge Institute of Technology
SWEDEN

Department of Interaction and System Design
School of Engineering
Publisher: Blekinge Institute of Technology
Printed by Printfabriken, Karlskrona, Sweden 2008
ISBN 978-91-7295-130-3

To my family…

Abstract

In most business areas today, competition is hard and it is a matter of company survival to interpret and follow up changes within the business market. The margin between success and failure is small. Possessing suitable, sustainable information systems is an advantage when attempting to stay in the front line of the business area. In order to be and remain competitive, these information systems must be up-to-date, and adapt to changes in the business environment. Keeping business systems up-to-date in a business environment such as this one, the telecom business, that changes rapidly and continuously, is a huge challenge. One way to approach this challenge is through flexibility in systems. The power of flexibility is that it keeps the system usable and relevant and allows it to evolve.

This thesis is concerned with end-user tailorable software. Tailorable software makes it possible for end users to evolve an application better to fit altered business requirements and tasks. In the view of tailorable software taken in this thesis, the users should be seen as co-designers, as they take over the design of the software when it is in use. In this work, it is important that the users are aware of the possibilities and limitations of the software.

However, tailoring is not enough, because the tailoring capabilities are always limited, meaning that tailoring cannot support completely unanticipated changes. The tailoring capabilities must therefore be extended, and tailoring activities must be coordinated with software evolution activities performed by professional developers. This allows the system to adapt continuously to a rapidly changing business environment and thereby live up to the intention of the system. Studies so far have tended to look at evolution from either a user perspective or a system perspective, resulting in a gap between development and use. This thesis takes an overall stand and states that it is possible to benefit from both the user and system perspectives, through collaboration between users, tailors and developers. It is necessary for users and developers to collaborate closely in order to make tailorable information systems both durable and adaptable to rapid changes in the business environment. In this way, the development of useful, sustainable software, which adapts easily to changes in an evolving environment, can be achieved.

This thesis also presents a set of tools to support collaboration on equal terms between users and developers, in the technical design process of evolving the tailorable software and extending the tailoring capabilities. The toolkit aims at building a common understanding of tailoring, supporting democratic agreements and a common understanding of what kind of tailoring to implement. It makes it possible for the users to take part in technical design decisions and have a better understanding of trade-offs and system boundaries. These are key factors for the successful future evolution of a tailorable system, as it is the users who are the designers of the software during its future use.

All of the research is based on field studies including participatory observations, interviews and workshops with users and developers. These studies led to the creation

of prototypes and tools that act as mediating artefacts when exploring the research questions.

The contribution of the thesis is twofold. Firstly, the thesis elucidates the *need* for a cooperative design process to ensure that end-user tailorable software remains useful and sustainable. Secondly, the thesis suggests a toolkit with four different tools to *support* such a cooperative design process.

Acknowledgements

First of all I want to express my gratitude to my research group. Without the good fellowship in U-ODD this thesis would not have been possible. Thanks to *Kari, Olle, Christina and Jeff* for illuminating discussions, feedback and support.

I want to thank the people at *Telenor Sverige AB in Karlskrona*, my industrial research partner, who have participated in my studies, or otherwise supported my research in various ways.

I also want to take the opportunity to thank…

… *Professor Lars Lundberg* and *Professor Claes Wohlin* for valuable feedback on my thesis.

… *Jeff Winter* for helping me improve my written English.

… *Associate Professor Yvonne Dittrich* for her everlasting, 'borderless' support. I also want to thank Yvonne for giving me the opportunity to do research in the first place.

… *Dr. Annelie Ekelin* for participation and valuable feedback in evaluation sessions.

… *Peter Warren* for fruitful cooperation during my time at the Space and Virtuality Studio (Interactive Institute AB) in Malmö.

I am also tremendously grateful to *my family* for putting up with me during the most intensive periods of work and for supporting me whenever I needed it. Last but not least, I also want to thank *my parents* who always support me with practical things, making everyday life easier.

This work was partly funded by The Knowledge Foundation in Sweden under a research grant for the project "Blekinge - Engineering Software Qualities (BESQ)" (http://www.bth.se/besq).

Papers Included in the Thesis

Paper I (Chapter 2): Olle Lindeberg, Jeanette Eriksson & Yvonne Dittrich, *Using Metaobject Protocol to Implement Tailoring; Possibilities and Problems,* the 6th World Conference on Integrated Design & Process Technology (IDPT 2002), June 2002.

Paper II (Chapter 3): Jeanette Eriksson, Olle Lindeberg & Peter Warren, *An Adaptable Architecture for Continuous Development; User Perspectives Reflected in the Architecture,* the 26th Information Systems Research Seminar (IRIS'26), Finland, August 2003.

Paper III (Chapter 4): Jeanette Eriksson, *Can End Users Manage System Infrastructure? - User-Adaptable Inter-Application Communication in a Changing Business Environment*, the 4th WSEAS International Conference on Applied Informatics and Communications, Spain and WSEAS Transactions on Computers, 6(3), December 2004.

Paper IV (Chapter 5): Jeanette Eriksson & Yvonne Dittrich, *Combining Tailoring and Evolutionary Software Development for Rapidly Changing Business Systems - What is required to make it work?* Journal of Organizational and End-User Computing, 19(2), 2007.

Paper V (Chapter 6): Jeanette Eriksson, Olle Lindeberg, Yvonne Dittrich, *Four Categories of Tailoring as a Means of Communication*, submitted to the Journal of Information Technology, February 2008.

Paper VI (Chapter 7): Jeanette Eriksson**,** *Support of Cooperative Design of End-user Tailorable Software*, the 2nd IFIP Central and East European Conference on Software Engineering Techniques CEE-SET 2007, October 2007. The title of the chapter is *Characteristics of End-User Tailorable Software.*

Paper VII (Chapter 8): Jeanette Eriksson, *Usability Patterns in Design of End-user Tailorable Software* , the 7th Conference on Software Engineering Research and Practice in Sweden, SERPS'07, October 2007.

Paper VIII (Chapter 8): Jeanette Eriksson, *Design Patterns in Design of End-User Tailorable Software*, submitted to the 10th biennial Participatory Design Conference (PDC 08).

Related Papers

Paper IX: Jeanette Eriksson, Olle Lindeberg & Yvonne Dittrich, *Leaving Variability Management to the End User; a Comparison between Different Tailoring Approaches*, Extended abstract in the Proceedings of the Software Variability Management Workshop, Gronningen, February 2003.

Paper X: Jeanette Eriksson, Olle Lindeberg & Yvonne Dittrich, *Leaving Variability Management to the End User; a Comparison between Different Tailoring Approaches*, Technical Report 2003:10, Blekinge Institute of Technology.

Paper XI: Yvonne Dittrich, Kari Rönkkö, Olle Lindeberg, Jeanette Eriksson, Christina Hansson, *Co-operative Method Development Revisited*, Workshop on Human and Social Factors of Software Engineering (HSSE) at the 2005 International Conference on Software Engineering (ICSE 2005) May 2005.

Paper XII: Yvonne Dittrich, Kari Rönkkö, Jeanette Eriksson, Olle Lindeberg, Christina Hansson, *Co-operative Method Development – Combining Qualitative Empirical Research with Process Improvement*, accepted for the Empirical Software Engineering Journal, published online December 2007, <http://www.springerlink.com/content/712m872162v41l86/?p=5f045b7d307f4f379423ac3e07a4af64&pi=0>

Paper XIII: Jeanette Eriksson, *Prioritizing Requirements – Planning Game vs. Kano Model*, not published.

Contents

Chapter One

Part I - Cooperation

Chapter Two

Chapter Three

Chapter Four

Chapter Five

Part II - Support

Chapter Six

Chapter Seven

Chapter Eight

Chapter Nine

Chapter Ten

Chapter One

Introduction

Chapter One

Introduction

Everything that can be counted does not necessarily count;
everything that counts cannot necessarily be counted.

Albert Einstein

In most business areas today, competition is hard. It is a matter of company survival to interpret and follow up changes within the business market. The margin between success and failure is small. Possessing suitable, sustainable information systems is an advantage when attempting to stay in the front line of the business area. In order to be and remain competitive, these information systems must adapt to changes in the business environment. This thesis is concerned with just such information systems, e.g. adaptable special purpose software used in a continuously and rapidly changing environment.

Keeping business systems up to date in a rapidly and continuously changing business environment such as, in this case, the telecom business, takes a lot of effort. Owing to the fast pace of change, flexibility in software is necessary to prevent software obsolescence and to keep the software useful. This inevitably means that the system has to evolve (Lehman, 1980). One way to provide the necessary kind of flexibility is end-user tailoring. End-user tailoring enables the end user to modify the software while it is being used, as opposed to modifying it during the initial development process (Henderson and Kyng, 1991). Software development, which is mostly done by professional software developers, involves transferring some domain knowledge from users to developers (Bennett and Rajlich, 2000) which may take some time and effort. End users, however, already possess the domain knowledge, so by providing support for end-user tailoring, enabling end users to make task related changes, alterations can be made immediately, as needed. Since time is money, a company can gain advantageous competitiveness if the business software can be at the forefront of the market changes. Thus, there is a strong motive to ensure that tailorable software is sustainable and lives up to the intention of the system.

So, the intention of tailorable systems is to make it possible for end users to evolve an application better to fit altered requirements and tasks, and to make the system more sustainable. The focus of this thesis is to explore how tailorable systems can continue to live up to the initial intention of the system in a rapidly changing environment and how this process can be supported.

If a software system is expected to adapt to changes in the environment, as tailorable systems are, the question is how to adapt to changes in a way that ensures that business and software systems can keep up with expanded requirements in a rapidly changing environment to ensure competitiveness and

for users to experience quality in use. The software system will be expected to deal with a range of changes that can be either anticipated or unanticipated.

There are two ways of adapting software to changing requirements:

- letting the end user adapt the system through tailoring or
- letting professional developers make the changes.

Changes made by users take place quickly and thus quickly satisfy the users' extended requirements, whereas software evolution performed by professional developers has the advantage of providing more far-reaching solutions.

The contribution of the thesis is to combine and coordinate tailoring with software evolution activities, to support the evolution of tailorability. A set of tools to support collaboration on equal terms between users and developers in the technical design process of evolving the tailorable software and extending the tailoring capabilities is also suggested.

1.1 Research Questions and Projects

The objective for the thesis is to explore how to support tailorability in a rapidly changing environment. By implementing tailorability, a tailorable system can continuously adapt to expanding requirements and thereby remain the competitive tool it was designed to be.

The main research questions for the thesis are:

RQ1: How can tailorability be supported to ensure that end-user tailorable software systems remain useful and sustainable and work as intended in a rapidly changing environment where requirements continuously expand?

The first question led to the main conclusion that allowing the tailorable software to evolve continuously requires a cooperative design process. Consequently, the second research question arose.

RQ2: How can the cooperative design process of end-user tailoring be supported?

The two research questions correspond to Part I and Part II of the thesis. The result of investigating the initial research question, how to ensure that end-user tailorable systems remain useful, sustainable and work as intended in a rapidly changing environment where requirements continuously expand, was the knowledge that a cooperative design process is needed, where both end-users and developers are together regarded as designers.

Exploring the second research question, dealing with how to support such a cooperative design process, resulted in a toolkit that can be used to make it possible for end-users to engage in the technical design process.

The thesis is based on four projects. Projects 1, 2 and 3 elaborated the first research question (RQ1) and the second research question (RQ2) drove Project 4.

The next section presents central themes in the thesis, which are relevant when exploring the research questions.

1.2 Focus

Cook et al. state "Programs that depend on or interact with the real world…must, in practice, be continually adapted to remain faithful to its application domain" (Cook et al., 2006, p. 9). Sommerville (2001) states that instead of developing systems and then maintaining them until the system has to be replaced, we should instead create evolutionary systems. Evolutionary systems are designed to change in reaction to changed requirements (Sommerville, 2001). Tailorable software is certainly in line with this and without doubt can be regarded as evolutionary systems. However, even though tailorable software is prepared for change, there will unavoidably come a time when unanticipated changes are needed that cannot be handled by the tailors. End-user tailoring differs from other types of interactive software in the fact that the software is under-designed (Fischer et al., 2004) and that the tailors continue to design the software during use. The tailors are co-designers. These issues form the foundation of the reasoning in this thesis and are shown in Figure 1 : 1

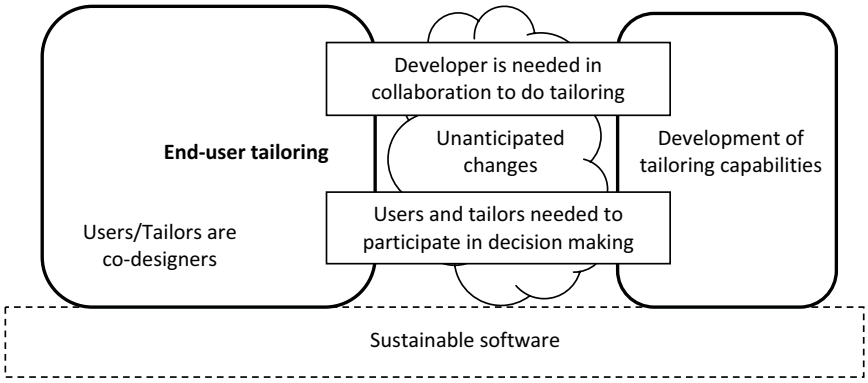


Figure 1 : 1 Central themes in the thesis

To keep the software sustainable and due to the fact that unanticipated change will occur there is a need for *development of tailoring capabilities*. This has been observed in Projects 1, 3 and 4. In Projects 2, 3 and 4 it was observed that *tailoring requires collaboration with the developer*. Furthermore, since the tailors are co-designers, there is a need for the *users and tailors to participate in decision making* to understand the possibilities and limitations of the software. This was also observed in Projects 2, 3 and 4. This thesis is about how to

involve the developer in the tailoring process, how to develop new tailoring capabilities and how to involve users and tailors in the design process.

The central themes from Figure 1 : 1 can be related to three areas: end-user tailoring, software evolution and Participatory Design. The collaboration between different roles involved when tailoring occurs is related to the area of tailoring itself, and how tailoring can take place. The development of tailoring capabilities is software evolution, and the need for user participation in design decisions is related to democratic decision making, which is central to Participatory Design.

The next section outlines the rest of the Chapter One.

1.3 Outline of Chapter

The rest of this chapter is organized as follows in Figure 1 : 2.

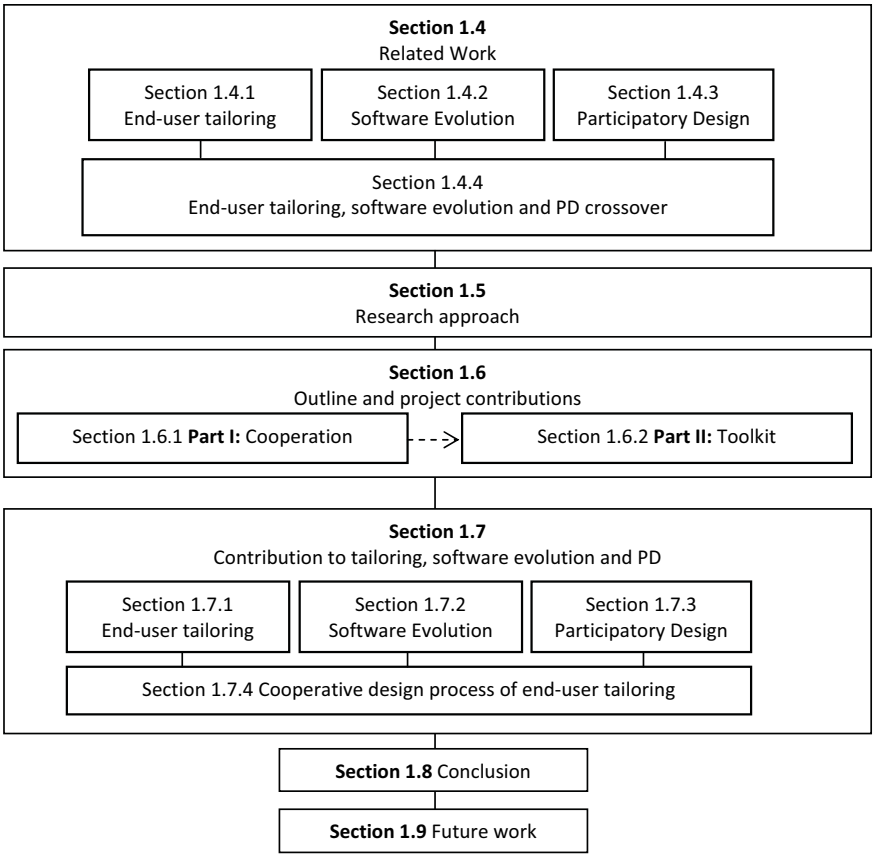


Figure 1 : 2 Overview of Chapter One

First, related work is presented. In Section 1.5 the Research Approach is described. The thesis is based on four projects which are presented in Section 1.6, which also describes the contribution of each of the following chapters (Chapters Two to Ten). Thereafter the contribution is related to the areas of end-user tailoring, software evolution and Participatory Design (Section 1.7). The section ends with a description of what in this thesis is called the cooperative design process of end-user tailoring. The contribution section is followed by the conclusions in Section 1.8. Finally, future research is presented in Section 1.9.

1.4 Related Work

Tailoring can be said to be "further development of an application during use to adapt it to complex work situations" (Kahler et al., 2000, p. 1) or "the activity of modifying a computer application within the context of its use" (Kahler et al., 2000, p. 1), hence tailoring is situated somewhere between development and use. End-user tailoring means that the tailor takes decisions about the design when he or she tailors the software.

1.4.1 End-User Tailoring

The research approaches can be divided into two principal areas:

- How tailorable systems and interfaces should be designed.
- How the end users work with tailoring.

The two categories do not have a clear boundary; most researchers discuss both categories simultaneously.

How tailorable systems and tailoring should be designed

When it comes to the design of tailorable systems, the prevalence of component-based solutions is noticeable. In (Mørch et al., 2004) the authors suggest new metaphors and techniques for choosing and bringing together components to facilitate end-user development. Stiemerling (2000) and Hummes and Merialdo (2000) also propose a component based architecture. Hummes and Merialdo also advocate dividing tailoring activities, as well as the application itself, into two parts: customization of new components and insertion of components into the application. The customization tool does not have to be a part of the application at all. This approach corresponds to Stiemerling's (2000) discussion of 'the gentle slope' where users can either just put together a few predefined components or, if more skilled, customize the components for more complex tasks.

In his doctoral dissertation, Paul Dourish (1996) proposes another approach, to make use of open implementation techniques to open up CSCW[1] toolkits,

[1] Computer Supported Cooperative Work

making it possible to manipulate the application to match the actual need. Dourish also states that there are basic connections between usage and system issues.

Fischer and Girgensohn (1990) take up another side of tailorable systems. They state that even if the goal of tailorable systems is to make it possible for users to modify systems, it does not automatically mean that the users are responsible for the evolved design of the system. There will be a need for modifications of the users' design environment and Fisher and Girgensohn provide a rationale and techniques for handling this type of change.

An area that is also interesting is the mapping between the adaptable system and the users; which interfaces to provide. Mørch (1995) introduces three levels of tailoring, customization, integration and extension, which provide the users with increasing possibilities to tailor the system. Customization provides only opportunities to make small changes, whereas extension is when code is added, which means that more comprehensive changes can be made. Together with Mehandjiev, Mørch (2000) also presents how to support the three different types of tailoring by providing different graphical interfaces for each of the tailoring types.

Costabile et al. (2006) works with a methodology they call the software workshop approach. The software shaping workshop (SSW) makes it possible for users to develop software artefacts without using traditional programming languages. SSW means that the software is organized to fit various environments. The software is specific for different sub-communities. When a user (called domain-expert) wants to develop an artefact only the required tools are available. The users experience that they just manipulate objects as they do in the real world (Costabile et al., 2006).

Letondal (2006) is exploring how to "provide access to programming for non-professional programmers" (Letondal, 2006, p. 207). She makes it possible for users to do general programming at use time. Her approach also involves the possibility to modify the tool used.

How the end users work with tailoring.

In (Mørch et al., 2004, p. 62) the authors state that an area for future research is "How to support cooperation among different users who have different qualifications, skills, interests, and resources to carry out tailoring activities." The area addressed is how the users work with tailoring. This area is well represented in the CSCW community. In the following, some research in the category is presented.

MacLean et al. stated in 1990 (MacLean et al., 1990) that it is impossible to design systems that suit all users in all situations and they continue by expressing the need for tailorable systems. However, it is not enough to provide the users with a tailorable system. To be able to achieve flexibility there is a

need for a tailoring culture, where it is possible for the users to have power and control over the changes. It also requires an environment where tailoring is the norm.

Wendy Mackay (1991) describes how she finds that although the users have tailorable software they do not customize the software, because it takes time from the ordinary work. There is a trade-off between how much time the tailoring takes to learn and how beneficial the change may be. To encourage users to customize the software, the customization has to allow users to work as before, and the customization must also increase productivity by just one single click of a button.

In another paper Mackay (1990) observes that customization of software is not mainly individuals changing the software for personal needs, but is a collaborative activity where users with similar or different skills share their files with each other. One group that has received attention is a group called 'translators'. Translators are users who are not as technically skilled as members of the highly technical group, but are people who are much more interested in making work easier for their colleagues. Mackay says that the translator role should be supported in organizations with tailorable systems. She also claims that not all sharing is good and that opportunities for sharing files have to be provided in the organization.

Gantt and Nardi (1992) find a role similar to the translator in a CAD (Computer Aided Design) environment. They identify gardeners and gurus. Gardeners and gurus are domain experts, not professional developers, who have the role of local developers providing support for other users. Gardeners and gurus differ from other local developers in that they receive recognition for their task of helping fellow employees.

As exemplified above, tailoring activities are often carried out in cooperation. This is also pointed out by Kahler (2001) who states that there is a lack of support for collaborative tailoring activities. Kahler therefore makes eight suggestions for how collaboration can be supported. The suggestions range from software issues to social-technical concerns. For example, Kahler suggests that a tailoring culture should be supported and that an awareness of the tailoring activities should be provided.

Susanne Bødker (1999) discusses computers as mediators between design and use. She provides an understanding of computer artefacts and how they transform in design, but also in use. Bødker states that designing software is a design embracing all environments of use.

Costabile et al. classifies different user (domain-expert) activities. They group the activities into two classes. Class 1 means that the user chooses from predefined options. Class 1 contains the activities of parameterization and annotation. Parameterization means hat the user specifies some constraints in the data. Annotation is when users write comments next to the data to clarify

what they mean. Class 2 contains several types of activities, All activities in Class 2 involves altering the artefact in some way (Costabile et al., 2006).

Changes in the environment or the organization influence software systems, and this phenomenon is recognized in tailoring literature. To manage organizational and technical changes Wulf and Rohdein (1995) provide a framework where both issues can be dealt with in an evolutionary and participatory fashion.

Pipek and Kahler (2006) describe four different scenarios for collaborative activities: Shared Use, Shared Context, Shared Tool and Shared Infrastructure. Shared Use means the users share knowledge of how to individually tailor the software. Shared Context occurs when the users collaborate to perform, for example, a shared task. When a groupware tool is used for collaboration (Shared Tool) the users get more dependent of each other, but they still have some possibilities to have individual configurations of their software instance. The fourth scenario, Shared Infrastructure, brings about severe dependencies which can cause problems (Pipek and Kahler, 2006).

There is a growing need for tailorable systems (Stiemerling et al., 1997) because of the variety of requirements on groupware. Stiemerling et al. (1997) suggest using participatory and evolutionary design approaches such as interviews, workshops, user advocacy, thinking aloud, mock-ups and prototyping when designing tailorable systems. This is in line with what is presented in this thesis, even though the application type is not groupware.

Summing Up

To sum up, it can be said that most researchers in the tailoring community approach tailoring and tailorable systems from a user perspective, irrespective of whether the main focus is on the design of tailorable systems or on how users use tailorable systems. To facilitate the developers' work is not considered, except as a side effect of trying to improve the interface (Stiemerling, 2000). The developer is not considered a member of the team in terms of changing the system (Figure 1 : 3), only as an assistance resource when the users' skills are not sufficient to allow them to tailor the system on their own (Henderson and Kyng, 1991). Mørch and Mehandjiev (2000) address the collaboration between users and professional developers by introducing 'multiple representations' of software entities. However, their approach means an indirect collaboration between users and developers. As shown in the figure the tailor only deals with the visible part of the software and the developer and the tailor only meet in the work of the developer when he or she has modified the software so that the tailor can do tailoring again. Consequently there is a gap between users and developers.

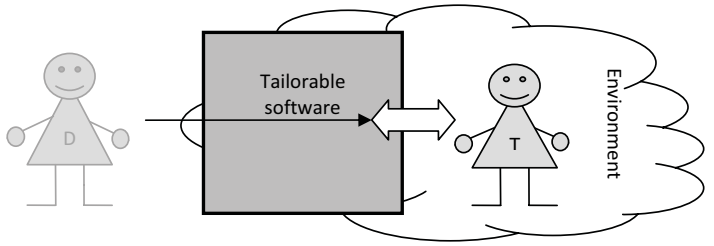


Figure 1 : 3 Tailoring

1.4.2 Software Evolution

Since there is no standard definition of software evolution; many researchers use it as a substitute for maintenance (Bennett and Rajlich, 2000). However, evolution expresses something more positive than maintenance, it means a lifelong positive change (Lehman, 1980).

A paper cited whenever software evolution is discussed is Lehman's paper 'Programs, Life Cycles, and Laws of Software Evolution' (Lehman, 1980). In the paper Lehman divides programs into three groups: S-programs, which can be derived directly from a ***s***pecification, P-programs that are programs that model and solve ***p***roblems, such as for example chess, and a third group, E-programs (E for ***e***volving), which are embedded in the world they model. In practice P-programs complied with the definition of either S-programs or E-programs according to Lehman's taxonomy (1980). Cook et al. (2006) have revised the definition of P-programs. P- and S-type programs are both programs where the stakeholders have made explicit policy decisions of what kind of evolution can happen in the system. A P-program is consistent with this strategy or ***p***aradigm throughout its lifetime. The kind of tailorable systems discussed in this thesis might be considered E-programs.

Lehman also states that questions about correctness, suitability and satisfaction will arise as soon as the application is used, and this leads to a need for changing the application (Lehman, 1994). In other words, Lehman states that the environment pressures the application to change and software evolution is inevitable.

Software evolution, in the same manner as software engineering, can be divided into efforts of software process and software product. The software *process* consists of four activities (Sommerville, 2001):

1. Software specification
2. Software development
3. Software validation
4. Software evolution

The initial development process involves specification, design and implementation activities, and testing, e.g. specification, implementation, validation. Then the software is finished, delivered and taken into operation. After a while the software is no longer satisfactory, and it inevitably has to evolve to meet new demands. Then a new phase begins, to define new specifications. The specification is implemented and validated and the evolved software is taken into operation, and then has to change and so on.

Since evolution (or maintenance) is a continuation of the development process it should be represented by a spiral model of development and evolution (Sommerville, 2001) (Figure 1 : 4) where the first round represent the initial development. Then the development process continues in the form of evolution.

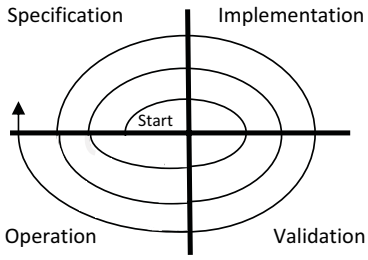


Figure 1 : 4 Spiral model of development and evolution

To meet the threat of decreased software quality as the software evolves, it is essential that change and evolution is placed in the centre of the development process (Mens et al., 2005). This is what Bennett and Rajlich (2000) do. From the *product* point of view, Bennett and Rajlich model the software life cycle in the 'stage model', consisting of five stages:

1. Initial development
2. Evolution
3. Servicing
4. Phase-out
5. Close-down

The initial development results in a first running version, and as soon as the software is deployed the evolution stage takes over. The software will undergo many changes until the ability to evolve is lost. Then the software enters the service stage. The software has become a legacy system and only small changes or services are made to the software. Eventually, no further servicing is possible, and the software will arrive at the phase-out stage where no changes are made to the software. Finally the software will cease to exist. Tailorable systems aim to stay in the evolution stage as long as possible.

To put the process and product approaches together, the first stages in the product life cycle (initial development and evolution and to some extent also

servicing) are embraced by the spiral model, whereas the stages phase-out and close-down occur when the spiral has ceased to spin.

Software evolution activities can be anticipatory or reactive (Bennett and Rajlich, 2001). Anticipatory evolution is based on the idea that it is possible predict, plan and prepare for changes before the need for evolution occurs. Tailoring and software variability (Svahnberg, 2003) and the product line approach (Bosch, 2000) belong to this approach. Reactive evolution means that changes are too unpredictable to be planned and changes have to be made when the need arises. This thesis emphasises the need to combine anticipatory and reactive (unanticipated) evolution.

Summing up

In summary it can be said that when discussing software evolution in software engineering, the main focus is on activities performed by professional developers. Figure 1 : 5 shows how the developer is outside the environment where the user belongs. The developer evolves the system from the outside to deal with the user's changing requests and changes in the environment. In software evolution, the intention of the developer is to evolve the system to meet the user's needs. As shown in Figure 1 : 5 the user deals with the visible part of the software in use and the developer evolves both the invisible and visible parts of the software. Consequently there is a gap between use and development, and users and developers.

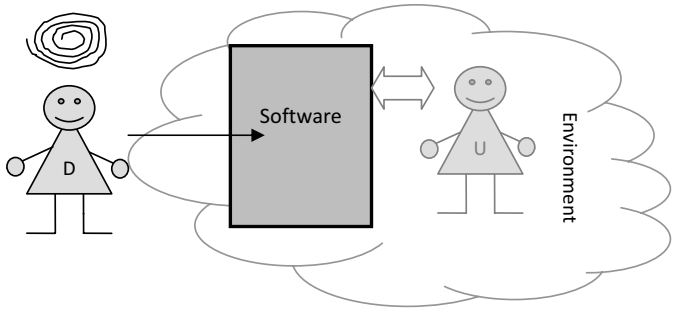


Figure 1 : 5 Software evolution performed by professional developers

1.4.3 Participatory Design

There is no single understanding of Participation Design (PD), but the core principles of PD are that (Sanoff, 2007)

- every participant is an expert in their own field,
- all participants' voices must be heard,
- good design solutions come from the collaboration of diversity composed groups,

- participatory democracy in decision making and
- engaging people in changing their own environment.

In summary, those individuals that have to adapt to the introduced change should be a part of the decision making (Kensing and Blomberg, 1998). Shapiro (2005) claims that if Participatory Design would be used when developing large scale systems in the public sector the failures would be less.

Two concepts that are essential to the successful outcome of PD projects are (Sanoff, 2007):

- The solution is informed by users' tacit knowledge.
- Collective intelligence.

Collective intelligence can be defined as the shared insight of an interacting group where the insight is more insightful and significant than the collective sum of the participants' individual understanding of the problem (Kensing and Blomberg, 1998).

The research concerning PD can be divided into three areas (Kensing and Blomberg, 1998):

- the politics of design
- the nature of participation
- methods, tools and techniques of participation

The *politics of design* is related to sharing power in the workspace, and the introduction of computer systems (Kensing and Blomberg, 1998). Initially there were two trends in PD originating from the politics of design: the Scandinavian (with the UK variation) and the American. The Scandinavian approach evolved out of power sharing or democracy within the workspace while the American line evolved from the fact that the computer-based systems tend to increase management control and therefore there was a need for strategies to facilitate direct worker participation in decisions.

In the area of *the nature of participation* a central concept is that there should be "room for the skills, experiences, and interests of workers in system design.." and that such a setting will "…increase the likelihood that the systems will be useful and well integrated into the work practices of the organization" (Kensing and Blomberg, 1998, p. 172). It is important that there is mutual learning and an understanding between the participants, both users and developers. In Participatory Design the participants alters between being experts or novices in a cycle dependent of what discussions and tasks is going on in the group (Farooq et al., 2005). The mutual learning process is good, but at the same time it is essential that the user representatives preserves their vocabulary and professional identity (Olsson, 2004).

The basic requirements for participation are (Clement and Besselar, 1993):

- access to relevant information,
- the possibility to hold individual opinions and views of the problem,
- participation in making decisions,
- access to suitable participatory development methods and
- alternative technical and organizational arrangements.

The extent of the participation can range from the users being limited to supplying designers with access to the users' skills and experience, to the users being considered valuable since their interest in the design solution is recognized. In this type of setting the users take part in the analysis of the requirements, the evaluation and selection of technological components, the design and prototyping as well as the organizational deployment (Kensing and Blomberg, 1998). The challenge is to find a balance of commitment and useful result (Letondal and Mackay, 2004). If the participating users experience the involvement to be hard it affects the result. However, if there is low-responsibility to participate in for example workshops it will affect the result as well.

Tools and the development of tools are an essential part of PD projects. The *techniques* utilize informal ways of exposing the relationship between the work and the technology. There are many *tools and techniques* to be used in a PD project ranging from techniques for analysing the work to tools for use in system design (Kensing and Blomberg, 1998). The tools and techniques can be used in different phases of the development cycle or iteration. Examples of tools and techniques are (Muller et al., 1993):

- Ethnographic Methods (Kensing, 2003)
  - *Purpose:* understanding users' work activities
  - Visiting the workplace to understand "the members' point of view".
- Contextual Inquiry (Kuniacsky, 2003)
  - *Purpose:* understand the users' work through inquiry, helps the users articulate their work practice.
  - Interviews where users are experts, the control is shared during the inquiry, shared meaning is created and reflection and engagement are important.
- Card Games (Muller et al., 1994)
  - *Purpose*: analysis of task and critique of design.
  - Cards represent events within the system, a workplace event or a user action.

- Future Workshops (Löwgren and Stolterman, 2004)
  - *Purpose*: users elucidate problems and create a vision of the future.
  - Three phases: critique, fantasy and implementation.
- Mock-ups (Ehn and Kyng, 1991)
  - *Purpose*: give the users possibilities to imagine the future by experimenting with new design proposals.
  - Inexpensive representations of the systems.
- Prototyping
  - *Purpose*: achieve a familiarity with the tool.
  - Cooperative activity involving both users and developers.

  Types:
    - Collaborative (Bødker et al., 1993)
    - Cooperative (Muller et al., 1994)
    - Storyboard (Muller et al., 1998)
    - Video (Bauersfeld et al., 1992)

Participatory Design is not a method but an approach, but PD embraces several *methods* that take a comprehensive view of PD. Some examples are (Kensing and Blomberg, 1998):

- MUST - a conceptual framework of the design process (Kensing et al., 1998)
- Contextual Design – with focus on early design activities (Beyer and Holtzblatt, 1997).
- Cooperative Experimental Systems Development (CESD) – user participation through the whole development process (Grønbæk et al., 1997).
- Work-oriented Design – field studies in combination with case-based prototypes (Blomberg et al., 1996).

Summing Up

In short, Participatory Design (PD) means that the users who will be affected by the new or changed IT-system have to participate in the decision making concerning the design of the software. Figure 1 : 6 shows how the developer is partly inside the user's environment. This is symbolic. PD ranges from designers participating in the users' world (e.g. ethnographic methods) to users participating in the design activities (for example the use of Mock-ups). As shown in the pictures both roles are equally visible, which means that they are equally important. PD activities deal with the work process and visible parts of

the software, such as the user interface. Consequently the gap between use and development, and users and developers, is partly bridged through collaboration.

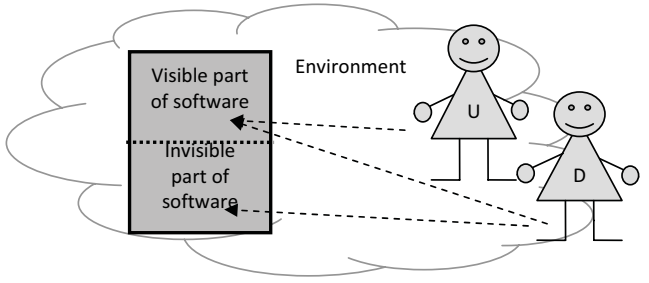


Figure 1 : 6 Participatory Design

1.4.4 End-User Tailoring, Software Evolution and PD Crossover

The division of the evolution process into specification, implementation, validation and operation is valid for tailoring too, but perhaps in a more informal way, as it is the end user who makes the change, alone or in cooperation with colleagues. The tailoring process can also be represented by a spiral model. The spiral model of tailoring starts after the initial round, which means immediately after the system is deployed. For each tailoring attempt there is a new rotation in the spiral. The tailoring process can continue until all tailoring capabilities are exhausted. Then the evolution through tailoring ceases temporarily.

Some researchers in the tailoring community have already related tailoring to software evolution. For example, Anders Mørch describes how end users can evolve a general tool, Basic Draw, into a domain-oriented design tool (Mørch, 1997). Even though the main focus is on how such a tool can be achieved, Mørch talks about evolution, and states that tailoring "supports application evolution by a set of tools that are integrated into a generic application" (Mørch, 1997, p.1). In another paper Mørch (2002) puts tailoring in the perspective of natural evolution and he introduces new concepts and techniques for software evolution.

Fischer also combines software evolution with tailoring, or as he calls it modifiable software or under designed software (Fischer, 2003). He calls this type of approach meta-design as the software is designed to be designed. The conceptual framework *seeding, evolutionary growth, and reseeding* (SER) (Figure 1 : 7) process model (Fischer et al., 2005) supports meta-design. SER encourages designers to conceptualize design activities as meta-design so that users can be active participants. After a period when tailoring has taken place, the software will have deviated from what can be regarded as good design and the software will need to be restructured or reseeded (Fischer et al., 1994)

As pointed out on page 4, there are two ways of adapting software to changing requirements: the end user adapts the software or the software engineer adapts it. Accordingly there are two ways for the users to influence the design of the tailorable software: *directly* or *indirectly*.

- The users *directly* shape the design while performing some tailoring activity (a). In other words. the participation takes place at *use time* and
- *indirectly* when participating in the cooperative design process to develop new tailoring capabilities (b). In other words, the participation takes place at *design time*.

Both ways can be regarded as Participatory Design. The first way of influencing the design (a) can be seen as a PD practice, while the second one (b) does not differ from common ways of regarding PD, during design time. The second approach makes use of the different PD tools and techniques.

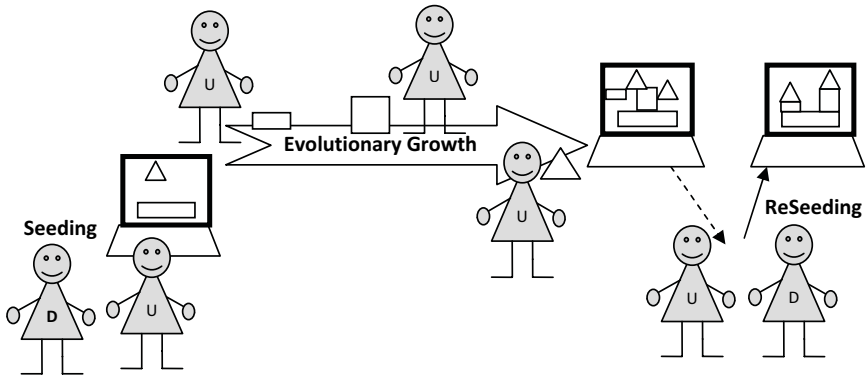


Figure 1 : 7 Seeding, evolutionary growth and reseeding (Fischer et al., 2005)

Muller et al. (1993) have proposed a taxonomy of PD practices and placed them in a two dimensional diagram. The x-axis represents the position of the activity in the development cycle or iteration (e.g. early or late in the iteration) and the y-axis represents who participates with whom in what (e.g. if the designers participate in the users' world or the users participate in design activities). If we position the two types of participations (a and b) in the diagram, (a) implies that the tailoring activity is put in the upper, right corner of the diagram, since it occurs late in the development cycle, in fact after the software has been deployed. The indirect influence of the design (b) means that all the different activities shown in the diagram can be used in the collaboration (except for the tailoring activity). Muller et al. do have a tailoring activity in the diagram called customization, but as tailoring in the context of this thesis is so much more than customization, the customization activity is replaced with simply 'tailoring activity' (Figure 1 : 8).

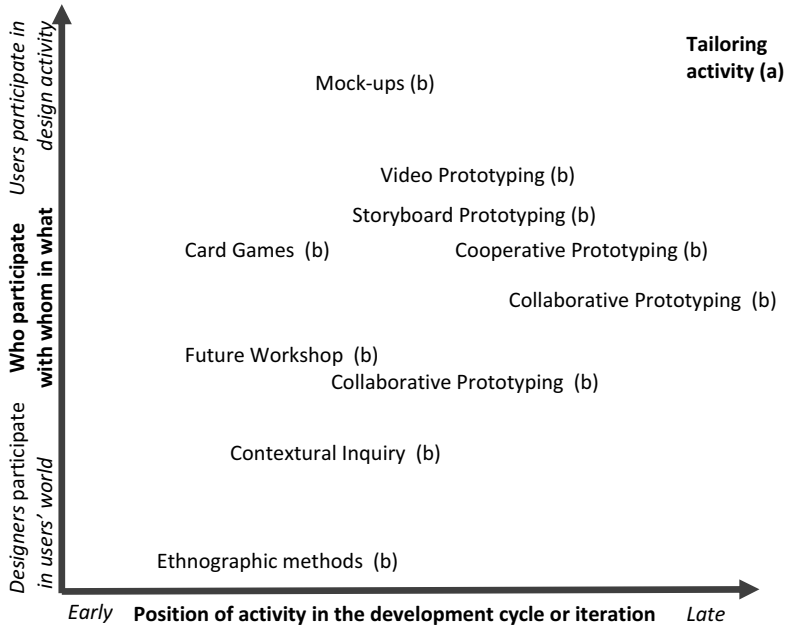


Figure 1 : 8 Participatory Design Activities (freely from (Muller et al., 1993))

Summing Up

End-user tailoring, software evolution and Participatory Design are interweaved (Figure 1 : 9), and when carrying out tailoring activities in this context, combinations of the areas result in different kinds of activities. For example:

- Refactoring to restore the tailorable software. (Fischer et al., 1994) (tailoring + software evolution) (AB in Figure 1 : 9)
- Collaborative tailoring activities between users and tailors (tailoring + Participatory Design) (AC in Figure 1 : 9)
- User participation in software evolution projects (software evolution + Participatory Design) (BC in Figure 1 : 9)

The intersection between all three areas (ABC in Figure 1 : 9) involves a combination of tailoring activities, collaboration between participants and software evolution activities performed by professional developers. It is this combination that is discussed in the thesis, in terms of the development of tailoring capabilities to extend the life of the tailorable software.

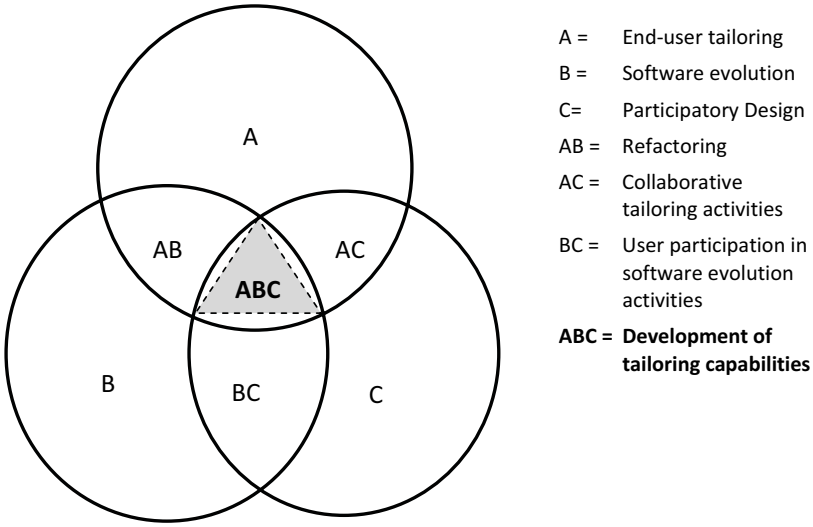


Figure 1 : 9 Intersections between the areas discussed in the thesis

1.5 Research Approach

This chapter starts by introducing Cooperative Method Development (CMD), which is the overall research approach used. Within the CMD approach, Design Research is applied, which is presented in Section 1.5.2. Since fieldwork, the creation of prototypes and evaluation are essential parts in the design research applied, the research within these areas is described in Sections 1.5.3, 1.5.4 and 1.5.5. This section ends with a discussion of the validity of the research (Section 1.5.6)

1.5.1 Cooperative Method Development

Software development is a social activity and thereby influenced by social aspects. This thesis is based on the belief that in order to improve software, it is essential to understand social and cooperative aspects of the work practice. It is also important to start from the practitioners' point of view, as their work situation has an impact on the company's success. Therefore the Cooperative Method Development (CMD) approach is applied as it combines qualitative social science fieldwork, with problem-oriented improvements. CMD is developed within the UODDS[2] research group.

[2] UODDS (Use Oriented Design and Development of Software). The group changed name to U-ODD (Use-Oriented Design and Development) in 2005.

In this section a brief description of CMD is given. For a complete explanation see (Dittrich et al., 2007). CMD addresses two main questions (Dittrich et al., 2007):

- How do software development practitioners tackle their everyday work, especially the cooperation with users around the design of software?
- How can methods, processes, and tools be improved to address the problems experienced by practitioners?

The CMD research process is modelled as evolutionary cycles divided into three phases:

Phase 1 - Understanding Practice: The research begins with empirical investigations whose aim is understanding practices and designs from a practitioner's point of view, based on their historical and situational context, and to identify aspects that are problematic from the involved practitioners' point of view (Figure 1 : 10).

Phase 2 - Deliberate Improvements: The results of the first phase are then used in the deliberation phase, as an input for the design of possible improvements. Suggestions for improvements are based on a combination of existing research in the discourse and domain knowledge in the company (Figure 1 : 10). This phase can be implemented in different ways. In the research presented in this thesis a Design Research approach has been chosen (Section 1.5.2). This phase also contains initial evaluations of created artefacts.

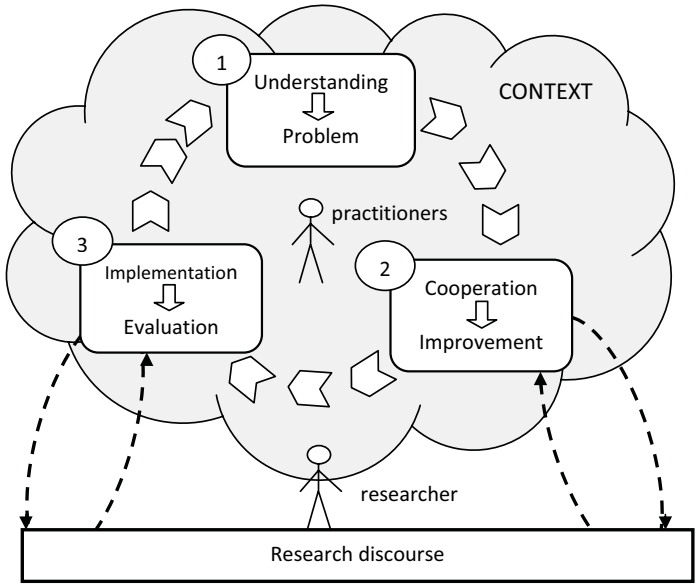


Figure 1 : 10 Cooperative Method Development

Phase 3 - Implement and Observe Improvements: The improvements are implemented. The researchers follow these method improvements as participatory observers. The results are evaluated together with the practitioners involved. This phase is also partly based on the knowledge in the research community (Figure 1 : 10)

The CMD approach also involves some guidelines (Dittrich et al., 2007) for performing research. The guidelines are:

- Focussing on shop floor software development practices.
- Use of ethnomethodological and ethnographical inspired empirical methods complemented with other methods when appropriate.
- Taking the practitioners' perspective when evaluating the empirical research and improvements.
- Improvements involving the practitioners.

These guidelines permeate the research presented in this thesis.

The thesis is based on four projects presented in Section 1.6. Table 1 : 1 summarizes how the CMD approach is applied in the different projects.

CMD	Project 1	Project 2	Project 3	Project 4
Phase 1	x	X	x	x
Phase 2	x	X	x	x
Phase 3	(x)		x	future work
	(x)= in another project			

Table 1 : 1 Implemented phases of the Cooperative Method Development approach.

1.5.2 Design Research Applied

The research methodology adopted within phase 2 of the CMD approach may be termed a design research approach, as the projects started out by defining the research question based on business needs and unexplored issues in the research discourse. Design research has been discussed in several papers, among others Nunamaker et al. (1991), March and Smith (1995) and more recently Hevner et al. (2004). Design research in general can be divided into five process steps:

1. *Awareness of problem* that can come from various sources, such as from industry or from other disciplines, but the findings must add knowledge to the research field.
2. *Suggestion* is closely connected to step one. In this phase a tentative design is achieved.
3. *Development* means that the tentative design is implemented.

4. *Evaluation* of the prototype according to implicit and explicit criteria in step one.
5. *Conclusions* are drawn and if the prototype is not good enough the process continues with step one once again.

Figure 1 : 11 relates the five steps of design research to CMD.

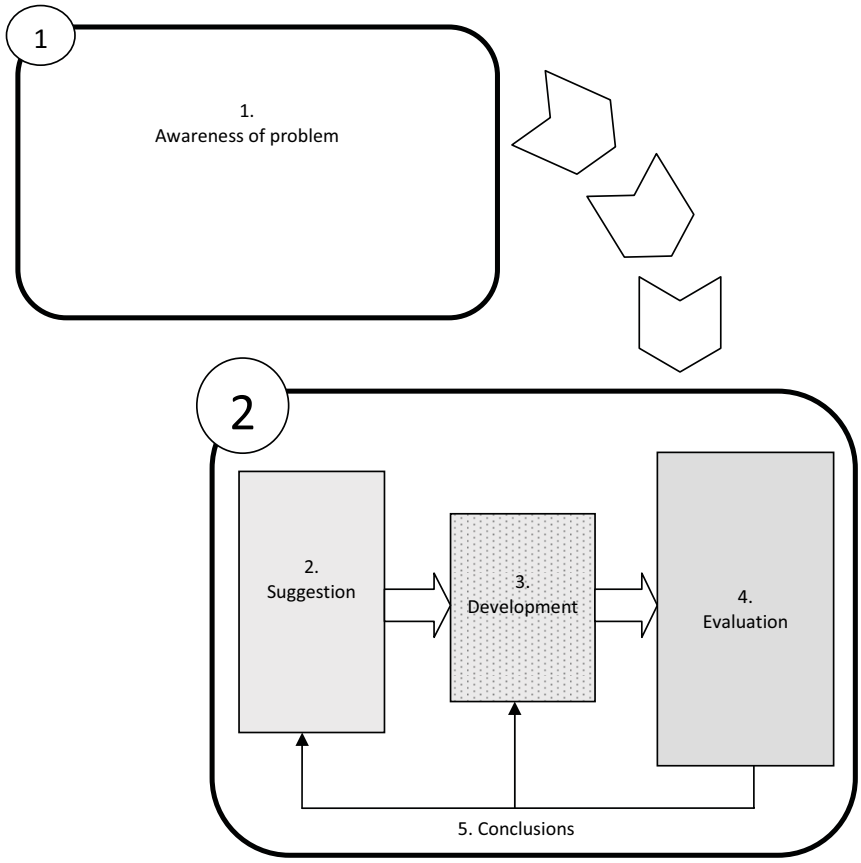


Figure 1 : 11 The five process step of design research

The design approach applied in the studies differs from the general view of design research, in that the goal for the evaluation was not limited to evaluating the quality of the prototype as a technical prototype, e.g. the aim was not to evaluate a comprehensive set of functional and qualitative requirements to be able to improve a specific prototype. The general view is that design research is concerned with how well a prototype works, but the output that design research should produce differs from community to community (Association of

Information Systems, <http://www.isworld.org>). In the projects presented here both 'how' and 'why' the prototype works is important. In other words, issues such as user knowledge, collaboration, and organizational aspects are also considered in the evaluation.

Hevner et al. (2004) emphasize the need for combining design research and behavioural science to "….ultimately inform researchers and practitioners of the interaction among people, technology, and organizations that must be managed if an information system is to achieve its stated purpose, …" (Hevner et al., 2004, p. 76). Cross-fertilization between design and behavioural research is applied in the projects presented in this thesis.

The chart in Figure 1 : 12 visualizes the applied design research. The chart is inspired by a diagram from Hevner et al. (2004). The notation in Figure 1 : 12 relates the applied approach to CMD. The approach follows the five steps in the general view of design research shown in Figure 1 : 11.

The numbers in what follows refer to Figure 1 : 12. The project starts with establishing the research question in terms of the industrial partner's needs (1b) in consensus with what is interesting from point of view of the research discourse (1a). Field studies and document studies (I) are applied, to elicit the needs generated by the research question (2a) together with the business needs (2b). Based on the outcome of the field studies, a prototype is built. To build the prototype, applicable knowledge (2c) is used. The prototype is designed together with users (II). To be able to evaluate the prototype it is assessed (3) either by researchers or by experiments in a setting close to the real world where users try out the prototype. The method used is close to field studies and the outcome is in the form of verbal protocols (III). The evaluation can be of three types: evaluation against requirements (A) (e.g. if the prototype satisfies the different requirements), evaluation of technical issues in the prototype (B) (e.g. *how* the prototype is implemented and what are the advantages and disadvantages compared to other implementations) and evaluation of environmental effects of the prototype (C) (e.g *why* the prototype works as it does, or in other words, what social impact the prototype implies). The evaluation design is based on established methods from the research discourse (4). The outcome from the evaluation generates new knowledge that is added to the knowledge base of the discourse (5a). In addition, the evaluation provides the industrial partner with findings that can be of use when designing similar systems (5b). The cycle is then complete and a new project taking advantage of the knowledge generated (5a), can begin.

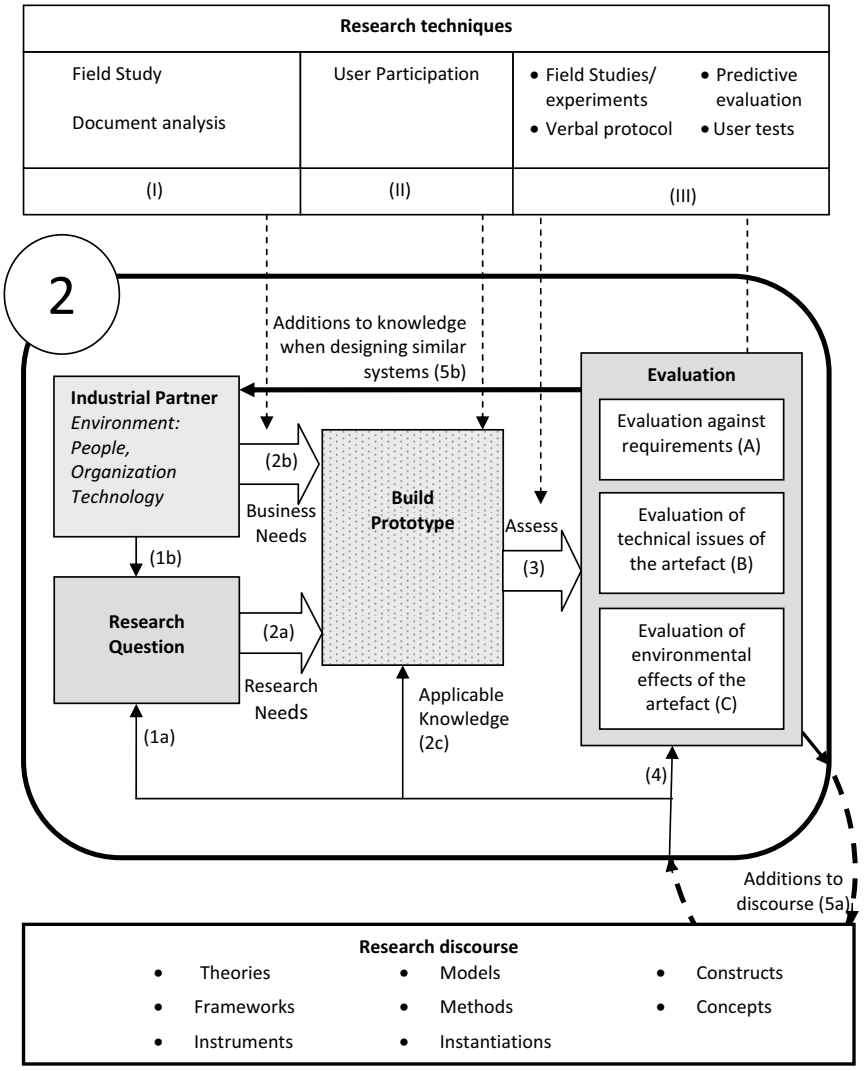


Figure 1 : 12 The research process

Table 1 : 2 shows how Design Research is employed in the different projects.

	Project 1	Project 2	Project 3	Project 4
Field work	-	Research Studio	Telecom Operator	Telecom Operator
Prototype	Contract Handler	ActionBlock System	EDIT	Toolkit
Evaluation	A	A, B	A,B,C	A
	Chapter Two	Chapter Three	Chapters Four & Five	Chapters Six to Ten

Table 1 : 2 Applied research approach in Phase 2

As the table shows there was no field work in Project 1. The reason for this is that the requirements elicitation had been done in a previous project modelling the same system.

1.5.3 Field Work

The projects begin with field work that is inspired by ethnography, which means that researchers enter the work environment with a probing and explorative attitude, instead of trying to find quick answers to predefined, detailed questions (Löwgren and Stolterman, 2004). The field work aimed at investigating the work practice and investigating which requirements there were for an identified problem. The main activities during this phase were participant observations and interviews of users and developers, since thorough field work requires both observations and interviewing (Gerson and Horowitz, 2002). Gerson and Horowitz (2002), elegantly express the possibilities of interviews and observations: “Whether the method is interviewing or observation, direct engagement in the social world focuses the sociological eye on the interaction between structure and action – in how people are embedded in larger social and cultural contexts and how, in turn, they actively participate in shaping the worlds they inhabit.”

There are different kinds of observations, from participant observation to structured observation. Participant observation means that the observer tries to be a member of the observed group, whereas the observer in structured observations takes the position of the ”pure observer” to be able to quantify the behaviour. The observations performed during the projects presented here are participant observations. One advantage of observations is that they are direct. It is possible to get to know people’s views, feelings and attitudes by watching what they do and how they do it and by listening to what they say. It is, however, time consuming (Robson, 2002).

This initial phase also included interviews of users and developers. Interviews can be fully structured, semi-structured or unstructured (Robson, 2002). Fully structured interviews use predefined questions, often in a predefined order. In unstructured interviews the researcher has an area of interest, and the conversation is allowed to develop during the interview and take any direction within that area. Semi-structured interviews are in-between structured and unstructured interviews. Predefined questions are used in semi-structured interviews, but the wording and the order may be changed, and questions can be omitted or added (Robson, 2002). All three types of interviews have been used in this research approach. The advantage of interviews is that they are flexible and provide quick answers to research questions, but interviews are also time consuming, even though they are faster than observations. The preparation and the supplementary work take time (Robson, 2002).

The danger of participant-observation is that the observations may be influenced by the interaction between the observer and the observed (Sánchez-Jankowski, 2002). Observations are also filtered by the researcher's experiences, expectations and interests. One way of confirming that the researcher has perceived the work practice in a correct manner, that the result of the observations is reliable, is to go back to the field and let the participants read the notes from the observations (Ely et al., 1993). This was done in the projects.

The field studies also involved document studies of specifications and manuals of existing systems.

1.5.4 Prototype

When a rich picture of the problem is assembled, a prototype or artefact is built that is an approach to solving the problem. The prototype is designed to fit together with existing technology and systems at the workplace. The prototype is designed in cooperation with end users and developers, in workshops at the workplace. The preliminary design of the prototype is presented there. This may result in changes in the design.

The prototypes presented in this thesis make use of different techniques and implement different solutions. Two of the prototypes (Chapter Two and Chapter Tree) are more or less proof-of-concept prototypes whereas the third prototype (Chapter Four) can be called a case-based prototype, a prototype containing real domain-specific data, addressing the work of a particular set of practitioners in a specific environment (Blomberg et al., 1996). The artefacts presented in Chapters Six to Eight are paper based PD tools.

1.5.5 Evaluation

As mentioned above, the research approach embraces the idea of cross-fertilization between design and behavioural research. Therefore, in the evaluation, we chose to use the prototype as a mediating artefact to discuss not only technical issues but also cultural and social factors in the organization that

influence the experienced quality. Using the prototype as a mediating artefact is also something that differentiates the research approach from other design research approaches.

Evaluation is done on both a system level (evaluation type A and B in Figure 1:7) and a use level (evaluation type B and C in Figure 1:7).

Quinn Patton describes two pure types of evaluation designs consisting of different evaluation methods (Patton, 1987):

- Pure hypothetical-deductive approach to evaluation:
- Experimental design, quantitative data, and statistical analysis
- Pure qualitative strategy:
- Naturalistic inquiry, qualitative data and content analysis

The different methods of evaluation can be combined to produce mixed forms of evaluations. Experimental design, qualitative data collection and statistical analysis can for example be combined (Patton, 1987). Which evaluation design to choose depends on what the stakeholders want to know, the purpose of the evaluation, the funds available and the interests of the researchers (Patton, 1987).

The evaluation design applied in the research approach presented here can be said to be a mixed form: experimental design, qualitative data collection and content analysis.

Evaluation against requirements (A)

Most software is based on comprehensive, preferably well-defined requirements. Building a prototype in a research project narrows down the set of implemented requirements to those that are most important to allow exploration of the research question. In a research project, evaluating the prototype against predefined requirements means ensuring, from a technical point of view, that the prototype really has the potential to conform to the rapidly changing business needs. By evaluating against requirements, the evaluators determine whether the prototype implements a possible solution for the stated problem. This is done through group discussions between the researchers involved. In addition, stakeholders at the workplace can also perform evaluation against requirements. In order to get a broader view of how well the prototype implements the requirements, it is preferable to perform both evaluations.

Evaluation of technical issues of the prototype (B)

Evaluation of technical issues can be done at two levels: how the prototype is implemented and which technical features the prototype provides for the end users.

How the prototype is implemented involves issues such as the understandability, performance and complexity of the prototype's construction

in comparison with other related implementations. This type of evaluation is performed by researchers. Which technical features the prototype provides is an issue for both researchers and other stakeholders. The researchers can decide whether the prototype is intended to implement a feature e.g. the researcher *has* implemented the feature. The other stakeholders, above all the end users, can decide whether they feel the feature is implemented in a satisfactory way. End users can carry out user tests in order to decide whether the feature is implemented satisfactorily. The difference between this type of evaluation in comparison to evaluation against requirements is that issues outside the requirements are considered.

Evaluation of environmental effects of the prototype (C)

A goal for the evaluation is also to find out what is required to realize quality in use when employing tailorable systems in a rapidly changing environment. It is only the users of the system who can decide if quality in use is achieved and it is not only dependent on the prototype itself but also on social factors. This type of evaluation is done through user tests. An evaluation paradigm that can provide rich and nuanced data is used in a specific setting to achieve a rich picture of the obstacles and possibilities of the prototype. Observation and verbal protocols in a setting close to the real world are employed. The ideal situation is to test the prototype in a real world setting, but most often it is impossible to perform user tests in the real environment. Factors such as open-plan offices that make it impossible to video tape the evaluations, or work processes involving money make it inconvenient or impossible to test the prototype in the real environment. The evaluations often have to be done in an environment as close as possible to the real environment, which means that user tests can be seen as experiments.

The prototype implements a process at the workplace and during the evaluation the users use the prototype as a replacement in the work process. In this way it is possible to discuss obstacles and possibilities, with the case-based prototype as a mediating object. Accordingly, the users try out the prototype in a setting close to the real-world environment with real-world data, whilst they ‘talk aloud’ (Ericsson and Simon, 1993, Robson, 2002) to express how they comprehend, perceive and understand the prototype. The ‘talk aloud’ technique is utilized if two users sit together and discuss with one other (Preece et al., 2002). In this way the conversation becomes smoother and the material becomes richer. When the number of end users participating in the project is small, a researcher can act as a ‘sparring partner’ for the end user. The researcher acts as a participating observer, prompting the end users to talk about what they experience. In this way it is possible to compensate for the fact that the users evaluate the prototype individually and not together. In addition, it is possible to penetrate issues of interest that otherwise may remain undiscovered.

The evaluations are recorded on video and audio tape, and after the evaluation sessions the tapes are transcribed, coded, categorized and analyzed.

1.5.6 Validity

Validity and reliability of qualitative research is connected to how the research is performed. Robson (2002) lists some criteria showing what is *required of good research*. We believe that the research approach described in this thesis fulfils these criteria.

- The data are collected through *multiple data collection techniques* (observations, interviews, workshops, discussions and document studies).
- The research has focused on the *participant's view* and the researcher has been the data collector in relation to the participants.
- Participant observations, semi-structured interviews and workshops are established methods in ethnographical studies, e.g. are part of an *existing tradition of enquiry*.
- The research started with an *aim of understanding* how tailorable systems can be used and can stay sustainable in rapidly changing environments.
- Good quality research should also *reflect the complexity of real life to be believable*. The methods used are employed because of their ability to provide a nuanced, complex set of data. The nuanced data are then used as a basis for prototype/tool construction.
- A *rigorous approach to data collection and analysis* is taken. All field work is documented either by notes or recordings. How the analyses are made is documented and a research diary is kept, containing thoughts and records of actions.

Even if the criteria for good qualitative research can be said to be fulfilled, there are threats to validity that have to be addressed. Robson (2002) describes actions to make to reduce the threats.

- By *peer debriefing and support,* researcher biases can be avoided.

Peer debriefing and support has been used in several constellations, mainly together with members from the U-ODD[3] research group.

- By *negative case analysis* (looking for instances that disconfirm the theory) research biases can be reduced.

[3] Use-Oriented Design and Development

During peer debriefing and support one of the researchers often acted as the 'devil's advocate' posing questions like 'What is this good for?', 'Can it be done another way?' etc.

- By *prolonged involvement*, reactivity and respondent biases can be avoided.

In the projects done in collaboration with partners outside the university, the research has involved being stationed at the studied workplace. The involvement with the partners has lasted between six months and one and a half years. Prolonged involvement can also be a source of research bias as the researcher may identify himself as being a part of the studied company. This threat has to be dealt with by for example negative case analysis or peer debriefing and support.

- *Triangulation* can be used to increase the rigour of the research.

Data triangulation has been used in the form of participant observations, interviews, workshops and studies of different kinds of documentation.

- By *member checking,* the obtained data can be verified.

The results of the analysis are verified by confirming the results with the participants. The verification was done either by presenting the results to the participants or letting the participants read the material. The same procedure was used to verify observations. Additionally, sub-results were presented at meetings together with representatives from the research partners.

- Through an *audit trail* (keeping full record of the research activities) the results become traceable.

Interviews and workshops are audio recorded. Notes were taken during observations, due to the restrictions on audio recordings in the open-plan office. The end user evaluations are video and audio taped and after the evaluation sessions the tapes were transcribed, coded, categorized and analyzed. Tape recording has three advantages compared with other qualitative data (Silverman, 2001), for example tapes can be replayed and they are also public records, which improves the reliability of the study. To video-record the evaluation adds another dimension to the audio tapes, as it is possible to see how the user acted as well as hearing what was said in a specific situation. By transcribing the tapes it is possible to illustrate the conversation in a way that makes it possible to recall not only what was said but also, for example, pauses, overlaps, hesitations and enthusiasm. To determine which initial categories to use, two or more researchers read through the material to discover interesting issues. The preliminary categories were established and the researchers coded the material individually. The material was colour coded, which means that the observations or responses are collected into groups of similar topics, and the groups are given a symbol or a code (Robson, 2002), in this case a colour. The

different coding sets are checked for correspondences and eventually the categories are brought together and the coded material is further analyzed.

Generalization of research results is always an issue. Qualitative research is often performed in a specific setting with a rather restricted number of participants who express their own points of view. However, this does not rule out qualitative research from possessing some kind of generalizability. A study can provide some degree of theoretical insights that can be transferred to other areas. This type of generalization is called analytic or theoretical generalization (Robson, 2002) as opposed to statistical generalization.

The fact that the same results can be observed in several projects (see Table 1 : 3 and Table 1 : 4) speaks in favour of the possibility of generalizing the result of this thesis beyond the specific settings of the projects. It is probable that the result is valid for companies other than telecom operators, if the company has support systems or business systems that have to *adapt quickly or temporarily to comply with changes in the environment* to remain competitive.

It is likely that the result is valid for different kinds of data intensive systems, as well as reflective systems with tailoring qualities, that is systems that depend on surrounding systems that change. In data intensive systems it is likely that tailorability is needed and it is essential to consider users, tailors and developers as equally important as it is probable that the amount of data will lead to new demands on the tailoring capabilities. In reflective systems, collaboration between all three roles is also essential because it is impossible to anticipate all changes in surrounding systems and the tailoring capabilities must therefore be extended.

It can also be expected that companies where there are layered business related changes may gain advantages from applying the results of this thesis. That is, companies where changes in business related tasks affect other tasks that in their turn depend on sufficient software support.

Even developers of embedded systems may find it helpful to consider cooperation as an issue when designing new systems. For example, an embedded system where the results are applicable is the Billing Gateway (Dittrich et al., 2006). The Billing Gateway is a system that sorts data records of phone calls and distributes the records to billing, statistics and fraud detection systems. The Billing Gateway makes it possible to tailor the sorting algorithms in accordance to, for example, new fraud indicators (Dittrich et al., 2006). How useful the result is for other tailorable embedded systems depends on the frequency and type of changes.

In companies in an environment with less frequent needs for change, cooperation between users, tailors and developers might not be that important.

Tailoring activities individualizing the systems to better fit the way individual employees use the software, or cosmetic tailoring, such as changing the appearance of the desktop, is not an area that is included in the results.

Individualization does not directly affect the company's competitiveness and therefore it is less important if the employee has to wait for changes.

Another case that may not be embraced by the result is when the need for change is so comprehensive that large parts of the system have to be reconstructed, but it is not likely that such major changes are as frequent as minor adaptations.

1.6 Outline and Project Outcomes

This section outlines the rest of the thesis and relates the different chapters to the different projects. The projects were driven by the research questions and each project generated new questions that drove the next project and so on. Some of the questions that were generated are left for future work.

This section elucidates the outcomes and contributions from the projects while the next section, Section 1.7, discusses the overall contributions of the thesis.

The thesis is based on four projects and eight papers, presented in seven chapters (Chapters Two to Eight). The thesis is divided into two parts. Parts I and II contain four papers each. Part II also contains two additional chapters that bring together the rest of the chapters in the second part. Figure 1 : 13 and Figure 1 : 14 show how the chapters are related.

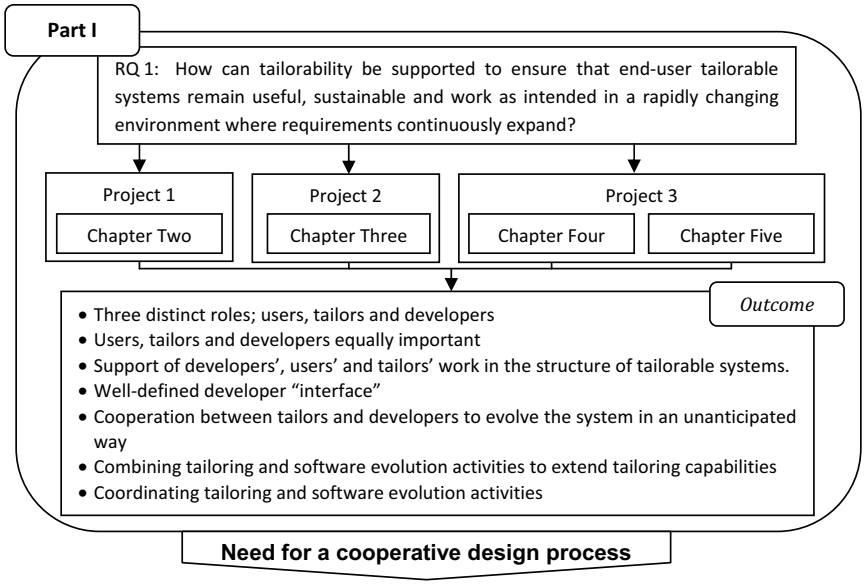


Figure 1 : 13 Overview of research questions and chapters in Part I

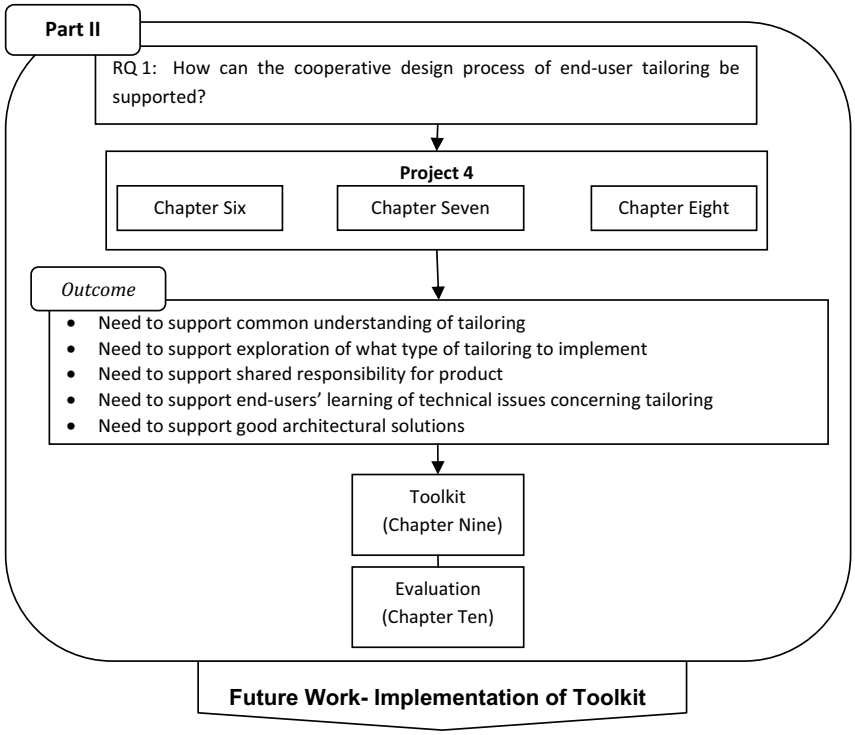


Figure 1 : 14 Overview of research questions and chapters in Part II

The initial research question (RQ1) initiated and drove the first three projects of the research, while Project 4 was driven by RQ2.

1.6.1 Part I – Cooperation

The first part consists of four chapters based on three projects elaborating research question one (RQ1: How can tailorability be supported to ensure that end-user tailorable software systems remain useful and sustainable and work as intended in a rapidly changing environment where requirements continuously expand?). The question generated several sub-questions during the research:

RQ1-a: Is tailoring enough to deal with expanded requirements?

RQ1-b: Is it possible to observe empirically the need for combining tailoring and software evolution, and the need for supporting developers' interaction with tailorable systems?

RQ1-c: From a technical point of view, how can cooperation between users, tailors and developers be supported?

RQ1-d: Is there a need for cooperation between users, tailors and developers in business environments?

The relation between the projects, chapters and research questions is shown in Figure 1 : 15.

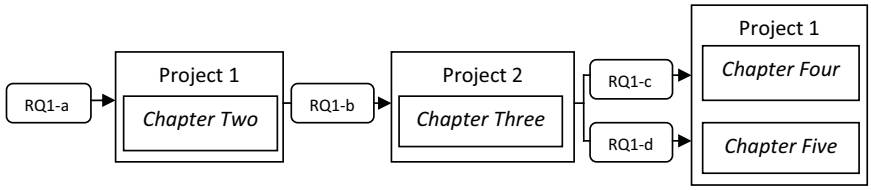


Figure 1 : 15 Relationship between the chapters in Part I

Project 1

The origin of the project was a student project. The goal of the student project was to build a flexible prototype intended for handling payments for a telecom operator. The aim was to explore how the payment system could make use of a very flexible database created during a previous research project (Diestelkamp, 2002). The result was a rather complex prototype that was difficult to manage. The complex prototype raised the question of how to make a similar application according to the same requirements but with an architecture that was clearer and more apparent. It is on these premises that Project 1 started in cooperation with U-ODDS[4] research group at Blekinge Institute of Technology. The aim of Project 1 was to explore if the metaobject protocol idea could be used for tailorable systems whilst at the same time making the structure of the application clearer. The project resulted in a prototype called ContractHandler. The project lasted from January to September 2001.

Chapter Two (page 57) describes a tailorable prototype, and how it was used to test the possibility of using the Java reflection API as a means of implementing tailoring. Tailorability was achieved by using the metaobject protocol idea. This means providing two interfaces to the application (base level and meta level interfaces) allowing the manipulation of the base level through the meta level interface (Kiczales, 1992).

Answers

The project answers the question "RQ1-a: Is tailoring enough to deal with expanded requirements?" The answer to the question is *no*, since it was clear that changes will eventually be needed that are not facilitated by the tailorable system. Tailorable systems are by nature designed to support use and tailoring, and since it was possible to anticipate unanticipated changes the prototype was

[4] UODDS (Use Oriented Design and Development of Software). The group changed name to U-ODD (Use-Oriented Design and Development) in 2005.

designed to facilitate such changes too. It had to be easy for the developer to add new java classes and thereby expand the tailoring capabilities.

Question

Finally, the project brought up a question:

- Is it possible to observe empirically the need for combining tailoring and software evolution, and the need for supporting developers' interaction with tailorable systems? (RQ1-b)

Project 2

Project 2 was performed in cooperation with a research centre in Malmö (Interactive Institute AB, The Space and Virtuality Studio[5]). The research team at the studio consisted of artists, engineers, industrial designers, software developers, hardware designers, etc. who worked in an open-plan office. The research team was, among other things, exploring the area of ubiquitous computing in terms of design processes. They were developing a system of ubiquitous and intelligent building blocks or physical interfaces, such as tag readers, digital cameras, loudspeakers, lamps, buttons etc (jointly called ActionBlocks). What was needed was a way to connect the different devices into different kinds of configurations dependent on the situation and on specific requirements. This is the starting point for Project 2. The author was stationed at the studio two to three days a week from January to June 2002. The aim of the project was to make a prototype that made it possible to easily connect physical devices together in different configurations. The work resulted in a prototype of an ActionBlock system.

Chapter Three (page 73) presents how different architectural paradigms were combined in a ubiquitous computing environment. The system was required to be able to deal with extremely unpredictable use scenarios. Different user roles and their usage of and perspectives on the system were used as a starting point for architecture design, in order to provide different levels of flexibility. By explicitly discerning and equalizing the importance of the three roles and by analyzing their use, interaction and perspectives on the system, it was possible to support the different roles by different approaches towards architecture.

Answers

The project provided an answer to the question "RQ1-b: Is it possible to observe empirically the need for combining tailoring and software evolution, and the need for supporting developers' interaction with tailorable systems?" and the answer is *yes*. Throughout the project the persons involved could clearly be

[5] Interactive Institute AB, The Space and Virtuality Studio ceased to exist in December 2003

observed to be divided into three roles: users, tailors and developers. What could also be observed was the close cooperation between users, tailors and developers. When observing the cooperation between the roles it also became apparent that it was necessary to regard the roles as equally important, to avoid the negotiations becoming biased.

Questions

The project raised two obvious questions:

- From a technical point of view, how can cooperation between users, tailors and developers be supported? (RQ1-c)
- Is there a need for cooperation between users, tailors and developers in business environments? (RQ1-d)

Project 3

Project 3 was done in cooperation with the same telecom operator that was indirectly involved in Project 1. The project started in October 2002 and ended in Mars 2004. Periodically, the author was stationed at the company two to three days a week. During the time between Project 1 and Project 3, the telecom company had invested in making the payment system tailorable by the end user. The system handling the data was however inflexible and could only handle specific data sets. This limited the flexibility and revealed the need to tailor the communication paths and data flow between different systems as well. What was needed was a tool for end users to make it possible for them to tailor communications between different distributed heterogeneous data sources. Therefore, the goal for Project 3 was to explore the possibilities and obstacles of providing the end users with such a tool. The physical result of the project was a prototype modelling the process of handling unpredicted extra payments. The prototype was called EDIT (Event Definer for Infrastructure Tailorability).

Chapter Four (page 93) presents a tool that allows end users to manage system infrastructure. The prototype provided an interface for the system infrastructure. It was shown that it is possible, through small means, to provide end users with opportunities to manage system infrastructure. This could be achieved by structuring the application in a way that clearly separated use, tailoring and further development of the tailoring capabilities.

Chapter Five (page 105) describes how the prototype was populated by real business data and used as a mediating artefact in the evaluation. The users tried out the prototype by carrying out one of their ordinary business tasks whilst they ‘talked aloud’ to express their experience of the situation. The outcome of the evaluation covered the areas of technical support, user knowledge, and organizational and cooperative issues.

Answers

Since Project 2 took place in a rather experimental environment it was interesting to see if the findings from the project were valid for business systems used on a daily basis. Since the task performed in the project in the telecom company is dependent on data from several surrounding systems that in their turn are dependent on one another, there is a need to coordinate tailoring activities and evolution activities that are performed in surrounding systems. This coordination inevitably requires collaboration between users, tailors and developers. The cost of the workload for the developers must not exceed the benefit for the users and tailors, because then the profit gained from tailorable systems would be lost. This suggests that users, tailors and developers should be viewed as equally important to satisfy when designing tailorable systems.

The answer to the question "RQ1-c: From a technical point of view, how can cooperation between users, tailors and developers be supported?" is: *by providing a distinct way for developers (as well as users and tailors) to interact with the system and by structuring the tailorable systems so that the concerns of the different roles are distinctly separated.* The prototype also implements a graphical interface for developers.

The answer to the question "RQ1-d: Is there a need for cooperation between users, tailors and developers in business environments too?" is *yes*.

Questions

The project raised many interesting questions that are out of the scope of this thesis, but which will act as input for future research.

- How should a tailorable system be structured to support users, tailors and developers and the cooperation between them?
- Can the separation of concerns and a division into three parts be regarded as a guideline for how to structure tailorable systems?
- Do the different types of tailoring require different system structures?
- How should the different interfaces be designed?

There was one question generated by Project 3 that was more distinct than others:

- How can the cooperative design process of end-user tailoring be supported? (RQ2)

The question was explored in Project 4 and the focus developed in the direction of how to support the cooperative design process when developing new tailoring features.

1.6.2 Part II – Support

During Project 4, when exploring the question "RQ2: How can the cooperative design process of end-user tailoring be supported?" four sub questions were developed:

RQ2-a: How can the communication that allows end-users, tailors and developers to reach a common understanding of tailoring be supported?

RQ2-b: How can the discussion and exploration of what type of tailoring to implement be supported?

RQ2-c: How can the learning of end-users in the technical design process be supported?

RQ2-d: How can the selection of good architectural solutions for tailorable software be supported?

These questions resulted in four different artefacts aiming at supporting the design process. In Chapter Nine the artefacts are put into context and the tools in the toolkit are given shape. In Chapter Ten the tools are evaluated in an initial expert evaluation. The relationship is shown in Figure 1 : 16

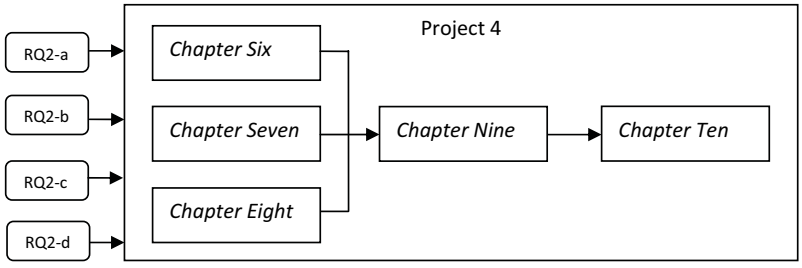


Figure 1 : 16 Relationship between the chapters in Part II

Project 4

Project 4 was done in cooperation with the same telecom operator as in Project 3. The project started in November 2004 and ended in February 2007. Once or twice a week during this period, the author performed participant observations during different types of project meetings, actively participated in design meetings and took part in discussions with users. A couple of months after the deployment of the new system a workshop was held to get an overview of factors that have influenced the creation of the system and to explore how the final system worked. The project also contained a set of interviews with users and developers. The interviews were based on the observations made during the development project. The aim of Project 4 was to explore the possibilities to support the cooperative design process of end-user tailoring and to construct tools that could assist the participants in development projects when end-user

tailorable software had to be modified to provide for extended tailoring capacities. The concrete result of the project was a toolkit that can act as a base for discussion, facilitate mutual understanding and make it possible to make informed design decisions.

Chapter Six (page 125) presents a categorization of end-user tailoring that considers both a user and a system perspective. When cooperating with industry we have experienced a need to systemize tailorability to be able to understand and discuss the phenomenon better. To be able to make informed decisions of what kind of flexibility to implement it is important that users and developers have a mutual understanding of what tailorability is. The categorization is intended as a support for discussions to reach such an understanding.

Chapter Seven (page 151) presents a matrix to support discussions between users and developers concerning what kind of tailorability to build into the software. The matrix contains attributes representing different types of tailoring as attribute values. The attributes represent organizational, business and technical issues to consider and can be used in a dialectic process to balance the human-centeredness and the technical solution.

Chapter Eight (page 162) presents a selection of usability patterns that are of vital importance for the success of end-user tailorable software, that also have architectural impact, and therefore should be addressed early in the design process. A subset of software design patterns suitable for end-user tailorable software was also selected. These patterns are aimed at providing support for technical design discussions when the group is more mature in terms of using patterns. The chapter also describes a pattern structure for patterns of end-user tailoring design.

Chapter Nine (page 193) is aimed at making the four artefacts from Chapters Six to Eight useful. All of the artefacts aim to be a means of communication, but to be useful they have to be made available for participants in a development project. The fact that they have to be packaged in a form that makes them available resulted in a toolkit which is described in the chapter.

Chapter Ten (page 217) presents the evaluation of the toolkit. The toolkit is intended to go through three design loops and three separate evaluations. The toolkit is currently in a prototypical state, but is intended for development and improvement in the two forthcoming design loops. It is the first evaluation, where the toolkit was evaluated by an expert team, which is presented in this chapter.

Answers

The answer to the question “RQ2-a: How can the communication that allows end-users, tailors and developers to reach a common understanding of tailoring

be supported?" is that *a categorization is a good start to begin to discuss and come to a mutual understanding of a phenomenon such as end-user tailoring*.

The answer to the question "RQ2-b: How can the discussion and exploration of what type of tailoring to implement be supported?" is *to base the discussion in some attributes that are associated with tailorable software and try to pinpoint which factors in the environment and in the system influence what is required for the tailoring capability. A matrix populated with values of the attributes can help in that discussion by guiding the team towards what tailorability to consider.*

The answers to the two questions "RQ2-c: How can the learning of end-users in the technical design process be supported? and "RQ2-d: How can the selection of good architectural solutions for tailorable software be supported?" is *by making it possible to combine a pattern approach with user participation*.

Questions

Two of the tools presented in Part II involve patterns and when working with patterns in the context of end-user tailoring some additional questions appear that must be left for future research:

- Can other specific patterns for end-user tailoring be distinguished in existing software?
- Which relationships exist between usability patterns and software patterns for end-user tailoring?

Furthermore, another question to ask is

- How does the toolkit work in practice?

The full answer will not be stated in this thesis, but the toolkit has been evaluated by an expert panel (Chapter Ten). An evaluation in a real setting will be left for future work.

But additional questions also arise:

- Is there a need for other types of tools in the cooperative design process?
- Should the toolkit be supported by software?
- Which alternative implementations of the toolkit are there and in which situations should they be used?

1.6.3 Summing Up

In Part I the need to combine tailoring and software evolution activities is confirmed in all cases. The first three first projects verify a need for collaboration between users, tailors and developers to provide for both anticipated and unanticipated evolution.

To summarize, the projects contributed the following:

- Three distinct roles, users, tailors and developers, were observed. (A)
- When designing tailorable systems, it is necessary to consider users, tailors and developers as being equally important to satisfy. (B)
- There is a need to support developers' work with tailorable systems in use. (Understood that tailors and users can be expected to be supported by the system.) (C)
- The need was observed for a well-defined developer interface or a well-defined way for the developer to evolve the system. (D)
- A need for cooperation between tailors and developers to evolve the system in an unanticipated way was confirmed. (E)
- There is a need to combine tailoring with software evolution activities performed by professional developers. (F)
- A need for coordinating tailoring and software evolution activities was observed. (G)

Table 1 : 3 summarizes the results from Part I. The table shows in which project the results occur.

Part I: Support of tailorability in Software Evolution	Project 1	Project 2	Project 3
Three distinct roles: users, tailors and developers (A)		x	x
Users, tailors and developers equally important (B)		x	x
Support of developers' as well as users' and tailors' work in the structure of tailorable systems. (C)	x	x	x
Well-defined developer "interface" (D)	x	x	x
Cooperation between tailors and developers to evolve the system in an unanticipated way (E)		x	x
Combining tailoring and software evolution activities to extend tailoring capabilities (F)	x	x	x
Coordinating tailoring and software evolution activities (G)			x

Table 1 : 3 Outcomes from Project 1,2 and 3

The fourth empirical project (Part II) confirmed the need for a collaborative design process and revealed a need for tools that could support the process. The project also acted as a basis for inspiration for the construction of the toolkit to support user participation in the design process.

The contribution of the project is four artefacts intended to:

- Support a common understanding of tailoring (G),

- support shared responsibility for the product (H),
- support exploration of which type of tailoring to implement (I),
- support end-users' learning of technical issues concerning tailoring (J) and
- support good architectural solutions (K).

The support is achieved by

- a categorization of end-user tailoring in a way that may be a useful means of communications in industry (Chapter Six),
- a matrix capturing characteristics for the different categories of tailoring and that can be used to elucidate different dilemmas concerning the implementation of new tailoring capabilities (Chapter Seven),
- a selection of vital usability patterns with architectural impact that can act as a gateway to using other types of patterns (Chapter Eight),
- a selection of design patterns suitable for end-user tailoring (Chapter Eight) and
- a pattern structure that supports both users and developers, since it provides for the possibility to enhance both a level of overview and details. (Chapter Eight).

The intention of the artefacts or tools presented in Part II is that they should act as a foundation for a comprehensive understanding of tailoring, stretching from what is needed, to how the requirements can be employed to engage users in the software system. At the same time the architectural structure of the software must not be jeopardised. This is done by providing

- a concrete toolkit that can be used in different phases of the cooperative design process of end-user tailoring to encourage user participation.

Table 1 : 4 summarizes the results from the three chapters. The table shows in which chapters the results occur.

Part II: Support of the Cooperative Design Process	Chapter Six	Chapter Seven	Chapter Eight
Support of common understanding of tailoring (G)	x	x	x
Support of exploration of what type of tailoring to implement (H)		x	x
Support of shared responsibility for product (I)	x	x	x
Support of end-users' learning of technical issues concerning tailoring (J)			x
Support of good architectural solutions (K)			x

Table 1 : 4 Outcomes from Project 4

In the next section the contributions from the projects are translated into the areas of end-user tailoring, software evolution and Participatory Design.

1.7 Contributions to Tailoring, Software Evolution and PD

End-user tailoring brings together the areas of software evolution and Participatory Design. In this section the overall contribution in all three areas will be presented and the cooperative design process of end-user tailoring will be delineated.

1.7.1 End-User Tailoring

When tailoring is discussed in literature, the focus is mainly on how end users perform tailoring or how tailorable systems should be designed. The developer's role is only briefly touched upon. For example Stiemerling (2002) states that Human Computer Interaction efforts often focus on optimizing interfaces for non-programmers and that this effort often has "the nice side-effect of making life easier for programmers as well" (Stiemerling, 2000, p. 33). *This thesis states that professional developers are as essential as users and tailors for tailorable systems in a rapidly changing business environment,* and to make the tailorable system work as intended, the activities of the three roles have to be coordinated.

The user and system perspectives are fundamentally different but this is a positive factor, presupposing that there is collaboration between end users and developers, because collaboration between different competences widens the boundaries for what is possible to do with a tailorable system. Nardi (1993) points out that end users with different skills cooperate when tailoring and she states that "…software design should incorporate the notion of communities of cooperative users…" which "…makes the range of things end users can do with computers much greater" (Nardi, 1993, p. 122). *By extending the cooperation to involve professional developers, 'things the end user can do with computers' may even increase.*

The contribution here is to include the developer in tailoring activities by supplying new tailoring capabilities. To achieve this, the tailoring capabilities can be extended in two ways:

- By providing a developer's interface where the developer can easily create a new tailoring capability (prepared development of tailoring capabilities (Figure 1 : 17)).
- By extending the tailoring capabilities without the assistance of an interface (unprepared development of tailoring capabilities (Figure 1 : 17)).

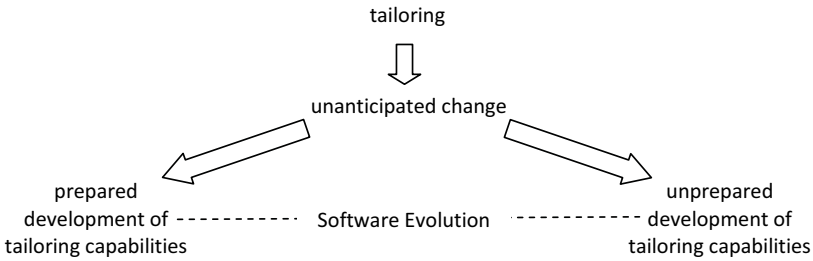
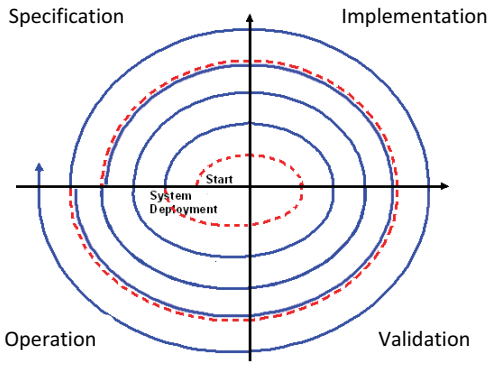


Figure 1 : 17 Two types of development of tailoring capabilities

1.7.2 Software Evolution

Software evolution performed by professional developers is more suitable for unanticipated changes. The projects show that it is easy to imagine situations where the need for unanticipated changes arises. The tailoring capabilities will eventually reach their limit and the capabilities will have to be extended to make it possible for the tailors to continue to tailor the software. The more rapidly the business environment changes, the sooner the limit could be expected to be reached.

Both tailors and developers evolve a tailorable system but they have different objectives for the evolution. Tailors evolve the system to be able to perform a task. The tailors perform ‘task driven evolution’. Developers, however, evolve a system to make it possible for the system to be a useful tool for the end users. The developer performs ‘system driven evolution’. For ordinary software systems that do not have any tailoring capabilities, both task driven and system driven evolution are the responsibility of the professional developer. For task driven evolution to be outsourced to the end users, the evolution has to be supported by the system, which means that the boundaries of what can be achieved are narrower than for software evolution performed by professional developers. Accordingly, tailorable systems in a rapidly changing business environment have to be combined with software evolution performed by professional developers. The continuous evolution of tailorable systems in combination with software evolution can be represented by the spiral model in Figure 1 : 18.



dashed line: activity performed by professional developers

continuous line: activity performed by end users

Figure 1 : 18 Spiral model of evolution of tailorable business systems

The similarities between Fischer's approach presented in Section 1.4.4 (seeding evolutionary growth and reseeding) and the one taken in this thesis is that users and developers collaborate in seeding, the users (tailors) evolve the software, and there will naturally come a time when there is a need to reseed the software, when it has deviated too much from what could be regarded as good design. The standpoint in this thesis is that there will also be need for extending the tailoring capabilities when new requirements arise (Figure 1 : 19) which is not considered in Fischer et al.'s approach..

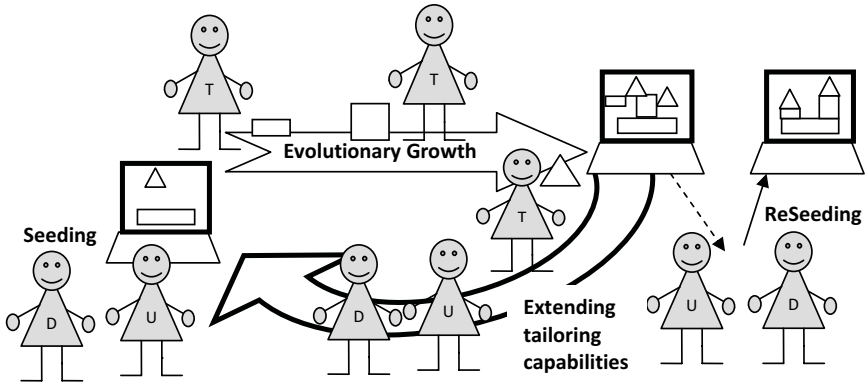


Figure 1 : 19 The approach in the thesis in terms of SER.

The contribution here is to embrace the users and tailors in the development of the tailoring capabilities and thereby state that software evolution in the context of end-user tailoring is performed in two steps: firstly, develop the tailoring capabilities and secondly, the tailor evolves the software by adjusting it to the task in hand. Since the users and tailors carry out the second step of software evolution it is essential that they also are a part of the first step so that they understand the underlying design decisions that set the boundaries of the software flexibility.

As both of the evolutionary activities are dependent on one another, it is necessary to coordinate the tailoring with the systems evolution that is performed by professional software developers.

1.7.3 Participatory Design

End users mainly see computers as tools that facilitate their work (Nardi, 1993). Nardi states that what motivates end users to make changes to the system is the need to change the system to be able to perform a specific task. End users want to make changes to systems as long as the changes are motivated by the task to be done. The end users have the ability to change the system, as they possess the domain knowledge needed for creating the applications they want, as well as the motivation to get their work done quickly and accurately (Nardi, 1993). But the users are also motivated to participate in development projects as this makes it possible for them to make their voice heard in the decision making regarding how the future software must work, which is essential for the users' work task.

As pointed out in Section 1.4.3, Participatory Design activities often deal only with the visible parts of the software. This thesis extends the notion of Participatory Design to include even activities concerned with invisible parts of the software (e.g. architecture). The reason for this is that the tailors are also designers, and to be able to make good design decisions during tailoring it is essential that the tailor knows the boundaries of the software. For the tailor to experience quality in use it is also important that the tailors know which design decisions underlie the boundaries. In the light of this reasoning the invisible side of the software shrinks and the visible parts expand.

Unanticipated change has to involve the developer, and unanticipated change can be dealt with in two ways (Figure 1 : 20), either by

- extending the tailoring capabilities through a prepared interface, or by
- extending the tailoring capabilities without support, in other words from scratch.

In both cases the tailor takes over the evolution after the new tailoring capability is in place and the tailor and the user already collaborate (Mackay, 1990). Therefore it is essential to involve all three roles in the evolution of the tailoring capabilities of the software. Such involvement embraces technical design decisions, and to make it possible for the users and tailors to participate in the technical design process a toolkit is proposed (Figure 1 : 20) in the second part of the thesis.

The contribution here is to involve the users and tailors in the technical design process by introducing tools that support the mutual understanding and learning that make it possible for the users and tailors to actively take part in technical design decisions and thereby influence the decisions that affect the boundaries of the tailorable software. The tools can be regarded as PD techniques aiming for the technical design process of end-user tailorable software. The tools are

structured in a systematic but yet flexible way that should facilitate reproducing successful collaborations.

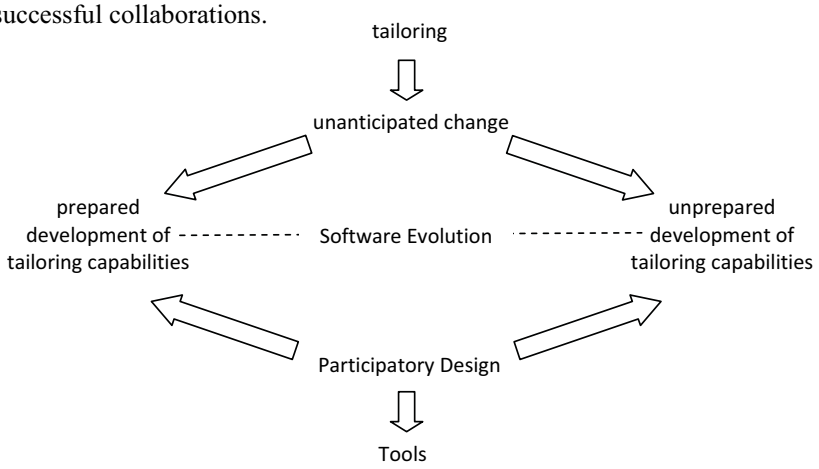


Figure 1 : 20 Two types of development of tailoring capabilities and PD

1.7.4 The Cooperative Design Process of End-User Tailoring

End-user tailorable software embraces two types of activities: use and evolution. Evolution can in turn be divided into sub activities: end-user tailoring, prepared development of tailoring capabilities and unprepared development of tailoring capabilities (Figure 1 : 21). These three evolution activities are included in the cooperative design process of end-user tailoring. The activities add to the previous levels as shown in Figure 1 : 21. When the software does not fulfil the needs the next level is used, if that it not sufficient the next level has to be considered.

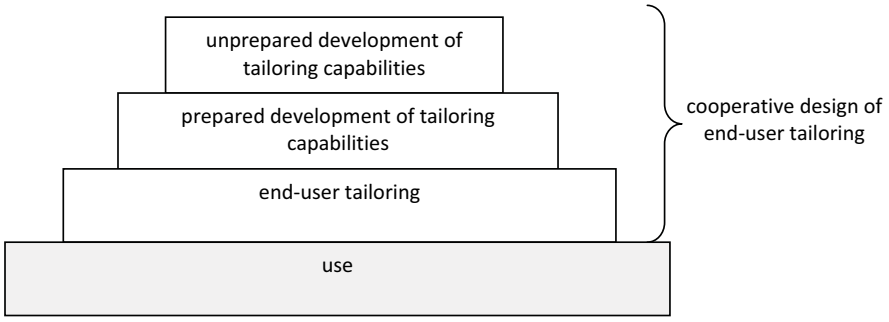


Figure 1 : 21 Components contained in the cooperative design process of end-user tailoring

End-user tailoring is implemented by the tailor and the software is equipped with the tailoring capabilities needed. The tailor is supported by an interface to facilitate the changes. The evolution is done in collaboration with users.

Prepared development of tailoring capabilities is implemented by the developer in response to unanticipated changes. The software is prepared so that the tailoring capabilities can be extended quickly. The developer is supported by an interface to facilitate the addition of tailoring capabilities. The evolution is done in collaboration with users and tailors.

Unprepared development of tailoring capabilities is also implemented by the developer. The difference from the above is that the software is not prepared for the activity. The evolution is done in collaboration with users and tailors and the toolkit presented in Part II can be seen as a substitute for an interface, as the toolkit supports the process to facilitate collaboration between users, tailors and developers.

Figure 1 : 22 visualizes the three interfaces, one for each role (user, tailor and developer) together with the tailoring activity that is not supported by an interface namely the unprepared development of tailoring capabilities. As shown by the spiral over the participants heads they all cooperate to evolve the software.

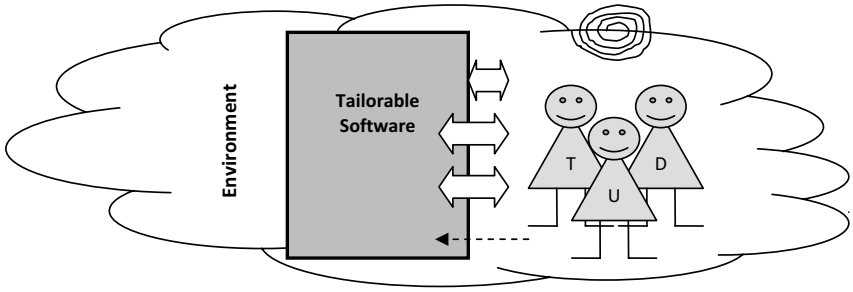


Figure 1 : 22 Cooperative Design of end-user tailoring

1.7.5 Summing Up

In summary the contributions of each of the three areas, tailoring, software evolution and Participatory Design, are:

End-user tailoring: The developer belongs to the collaborative team of end-user tailoring and the developer's contribution to the collaboration is to extend the tailoring capabilities to meet unanticipated needs.

Software Evolution: Software evolution is accomplished in two steps in the context of end-user tailorable software. In the first step the developer extends the tailoring capabilities and in the second step the tailors modify the software to give the user the ability to perform the desired task.

Participatory Design: Tools to support the cooperative design process are introduced to make it possible for the Participatory Design activities to embrace technical design decisions regarding for example the structure and architecture of the new tailoring capabilities.

The overall contribution of the thesis is to merge elements from the three different areas and to describe how all of the areas belong in a cooperative design process of end-user tailoring to keep the software sustainable. In addition the contribution is to propose a toolkit that can support the process.

1.8 Conclusion

Software evolution can take place through tailoring. The different projects showed that tailoring alone is not always enough to deal with expanded requirements. Tailoring has to be combined and coordinated with software evolution activities performed by professional software developers when tailoring capabilities have to be extended. To coordinate end-user tailoring and software evolution activities requires continuous cooperation between users, tailors and developers to bring together different competences to make a joint effort to evolve the system in the desired direction.

Some common results were more or less visible in the different projects.

- Three roles could be clearly distinguished: user, tailor and developer.

Tailors are often also end users, but are more skilled in handling the system and are thereby able to tailor the system to fit new or altered tasks better.

- The task related evolution done by tailoring could be anticipated to a certain degree, but in a rapidly changing business environment the tailoring capabilities will rather soon reach their limits.

Then system related software evolution has to be done to extend the tailoring capabilities so that the tailor can continue to evolve the system. Tailoring should be combined with software evolution performed by professional developers.

- Collaboration between users, tailors and developers was observed, as was a need for coordinating tailoring and software evolution activities
- In the empirical studies, there was an awareness of the competencies of colleagues, and the differences were used for collaboration.
- The design of useful, sustainable, tailorable system should support use, tailoring, and ordinary software evolution.

The prototypes showed how it is possible to facilitate all three roles' relation to, and interaction with, the system.

- User participation should be supported in all phases of the development process.

- The toolkit explores how to support the cooperative design process by involving the end-users in the technical design process.

The empirical studies also revealed a need to support:

- the creation of a common base of understanding
- a learning environment to make it possible for the users to understand technical decisions and their consequences for use
- a learning environment that makes it possible for users to participate in the development project on equal terms
- shared mental models
- agreements of trade-offs
- that all parties in the development project participate in the decision making

Accordingly, tailoring activities performed by end users and software evolution activities involving professional developers have to be coordinated for users to be able to experience lasting quality in use and to keep the business competitive. This concerns end-user tailorable systems in a rapidly changing business environment. It follows that an answer to the initial research question (RQ1) "How can tailorability be supported to ensure that end-user tailorable software systems remain useful and sustainable and work as intended in a rapidly changing environment where requirements continuously expand?" is that there is need for a continuous, close cooperation between the parties for the software to adapt to expanding requirements irrespective of whether the tailoring capabilities are sufficient for the required changes, or need to be extended.

In conclusion, the thesis merges elements from three different areas, tailoring, software evolution and Participatory Design, and states that the areas all belong in a cooperative design process of end-user tailoring to keep the software sustainable. The answer to the second research question (RQ2) "How can the cooperative design process of end-user tailoring be supported?" is to provide for appropriate PD techniques or tools that encourage learning and promote a democratic decision process.

The overall contribution of the thesis is to describe the cooperative design process of end-user tailoring and to suggest a toolkit to support it.

1.9 Future Work

This thesis emphasizes the importance of supporting tailorability in software evolution to achieve sustainable systems that adapt to extended requirements. The support is made possible by collaboration between end users and software developers. The collaboration is enriched by the fact that end users and developers have different perspectives on evolution, but the collaboration also

has to be supported by the design and implementation of the tailorable system. In Part I (Section 1.6.1) some questions arose that should be answered in future research.

How should a tailorable system be structured to support users, tailors and developers and the cooperation between them?

The three prototypes show that it is possible to support users, tailors and developers, but is it possible to finds some general principles of how tailorable system should be structured to support users, tailors and developers and the cooperation between them?

How should the different interfaces be designed?

The evaluations of the prototypes built in the different projects also resulted in technical findings concerning interfaces or interaction points for the different actors (user, tailor, developer). How these interfaces should be designed to support different actors, and in extension the collaboration between the roles, might also be an issue for further research.

Can the separation of concerns and a division into three parts be regarded as a guideline for how to structure tailorable systems?

All three prototypes explicitly implement separation of concerns for the three roles. It is a question for further research whether this type of architecture can be regarded as a guideline for how to structure tailorable systems for a rapidly changing environment.

Also in Part II there are some loose ends to follow in future research. The most noticeable thing is that the toolkit must go through additional evaluations and improvements. The question is:

How does the toolkit work in practice?

The toolkit should be evaluated in an experiment close to a real world setting and then be tried out in two or three real projects with different maturity in terms of user participation, so that the results can be compared and improvements can be made with different target groups in mind. The question to answer is:

Which alternative implementations of the toolkit are there and in which situations should they be used?

And of course

Is there a need for other types of tools in the cooperative design process?

There is also an open question of how to implement the tools in the organization.

Should the toolkit be supported by software?

Should the toolkit remain low tech, or should the tools be encapsulated in a cooperative IT-system, or should the toolkit be both physically and digitally

represented, so that the participants work with physical objects but reflections and decisions are collected and stored digitally, similar to the experimental setting in Chapter Three?

Another track to follow is to investigate different instances of tailorable software to explore possible architectural patterns specific to end-user tailoring.

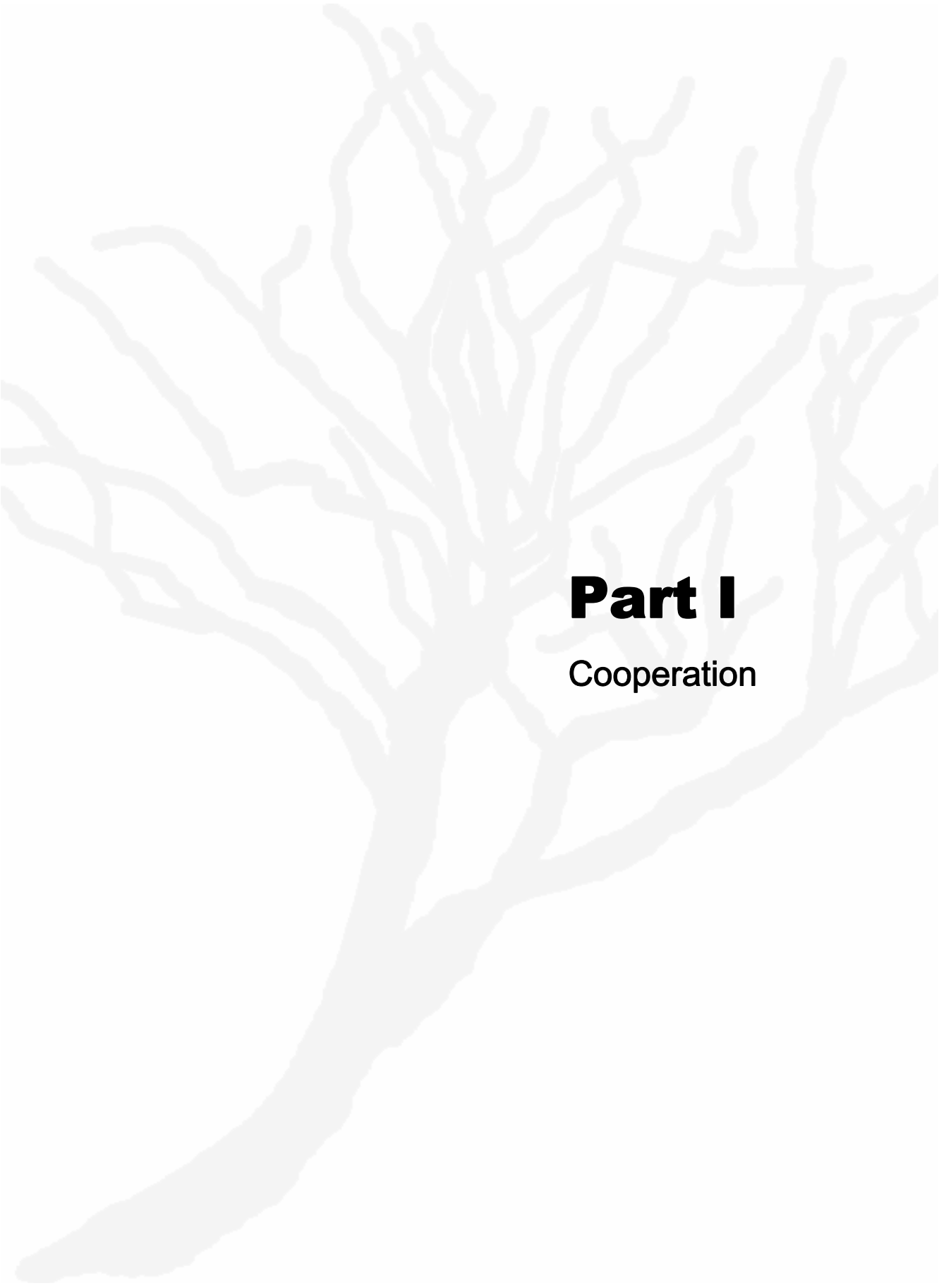
The question is:

Can we distinguish specific patterns for end-user tailoring by studying existing software systems?

Another pattern related issue that would be interesting to explore further is the relationship between usability patterns and software architecture patterns. Research efforts in this direction have been made (Bass and John, 2001, Bass and John, 2003, Folmer and Bosch, 2003., John et al., 2004, Juristo et al., 2003), but not in terms of end-user tailoring. The question is:

Which relationships exist between usability patterns and software patterns for end-user tailoring?

Accordingly there are many questions left to answer, and that is of course what research is all about, to pose questions.



Part I

Cooperation

Chapter Two

Paper I

Chapter Two

Using Metaobject Protocol to Implement Tailoring
Possibilities and Problems

The 6th World Conference on Integrated Design & Process Technology, 2002

Olle Lindeberg, Jeanette Eriksson, Yvonne Dittrich

This article is based on an experiment in using the Java reflection API[1] as a means of implementing a tailorable system. The background and idea behind the experiment was a research project in which we and two industrial partners collaborated. The goal of the project was to investigate a means of developing flexible, adaptable and modifiable software systems. The system that the prototype was modelled on is an application used by one of the research partners that is a telecommunication operator. It was possible to anticipate the type and structure of some of the changing requirements and for them tailoring (Henderson and Kyng, 1991) is a possible way to make the system modifiable. To read more about the project see (Dittrich and Lindeberg, 2002). The other partner had developed a meta-model database system. During the research project we developed several prototypes to test how to make a tailorable system using this database. At the same time a normal system development was carried out at the company. The resulting system has only limited tailoring capabilities. The need to make the software adaptable was instead satisfied by making the software easy to modify. This was achieved by making the software in components, which with only a little programming effort can be assembled in new configurations.

There were several reasons why the meta-modelling database tested in the prototypes were not used in the system. Here we will take up only one of these: the system seemed to become too complicated when tailoring was added to all other requirements. This is an example of a general problem; when you add tailoring capabilities to a system this often makes the system more complicated: not only do you have to construct the tailoring interface but the basic program may also become more complicated. To avoid this we constructed the prototype using ideas based on the metaobject protocol (MOP) approach (Kiczales *et al.*, 1993). The prototype described here is a combination of the MOP approach and the components used in the system development mentioned earlier. This combination results in the meta-programming approach; this prototype is less complex than those implemented earlier on in the project

[1] <www.sun.com/j2se/1.3/docs/api>

The rest of the chapter is structured as follows (Figure 2 : 1).We start by giving a sketch of what a metaobject protocol is, and more particularly, what it is in Java. We then give a description of the software architecture of the complete system (Section 2.4). The prototype implements only part of the system. Following the software architecture are the design and implementation of the prototype (Section 2.5). Finally, some conclusions from the prototype are drawn.

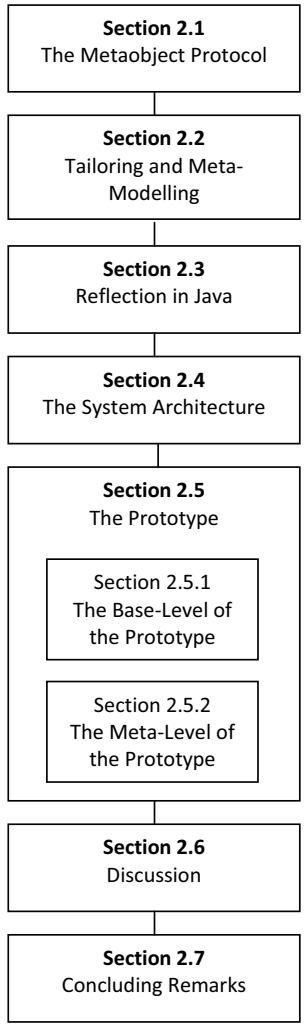


Figure 2 : 1 Overview of Chapter Two

2.1 The Metaobject Protocol

The metaobject protocol approach originates from the CLOS programming language in which it is possible to change program behaviour by interacting with the runtime system through a metaobject protocol (Kiczales, 1991).

The metaobject protocol is based on the idea that one can and must open up programming languages so that the developer is able to adjust the language implementation to fit his or her needs. This idea has subsequently been generalized to systems other than compilers and programming language. Kiczales (1992) argues that the metaobject protocol concept can be used as a general principle for abstraction in computer science. The idea is that any system that is constructed as a service to be used of client application (as for example an operating system or a database server) should have two interfaces: a base-level interface and a meta-level interface (Kiczales, 1992). The base-level interface gives access to the functionality of the underlying system and through the meta-level interface it is possible to alter special aspects of the underlying implementation of the system so that it suits the needs of the client application. The meta-level interface is called the metaobject protocol (MOP). Simply put, a MOP is a set of rules by which to manipulate and communicate with metaobjects.

- A MOP shall consequently:
  - Provide extended control over the behaviour of the system.
  - Have a clear division between the base-level and meta-level interface.

2.2 Tailoring and Meta Modelling

We have adopted a different approach towards the metaobject protocol. The idea of the metaobject protocol approach has inspired us to transfer the concept to end-user tailorable software. In most systems the end user has no access to the implementation of the program; in our approach the end user is given the opportunity to alter or tailor the software should the need arise. Our aim is to give the user the opportunity to add components to the program in a controlled way which does not require any programming. To do this we use a dual-interface: a traditional base-level program and a meta-level program that provides tailoring for the base-level program.

The distinction between a computational base level and a tailoring meta level is a useful one in a tailorable system. In the same way as in a metaobject protocol, the base-level implements what the system normally does. At the meta level you can change what the base level does. The two levels are also often separated in the user interface with a separate tailoring interface. The same separation may exist in the internal design.

Perhaps the obvious way to do this is to let the base-level program be controlled by meta-data which stores the choices the user has made when tailoring. If the tailoring possibilities affect a large part of the program, the base-level program may become littered with tests for the value of the meta-data. If the tailoring is complicated the result may be that the base-level program looks more like an interpreter of the meta-data than a straightforward program. We call this method of implementing tailoring the *meta-data approach*.

The alternative way to implement a tailorable system, the *meta-programming approach*, is closely linked to the metaobject protocol approach. With the meta-programming approach the base-level program is a normal program which performs the normal computation only. When the system is tailored by the meta-level this is implemented by changing the base-level program. To be able to do this we must be able to change (or at least add to) the program during execution. In Java this is possible since new class libraries may be loaded and linked during runtime. Another question is, "where is the meta-level description of the current configuration of the system stored?" In the meta-data approach the meta-level can inspect the meta-data to see how the program is configured; it is the meta-data that will be changed during tailoring. In the meta-programming approach the base-level does not need any meta-data. The radical solution is to take away the meta-data from the meta-level too. This means that it is the base-level program itself that is the meta description of the current configuration. This is the method we have chosen in the prototype.

We have used Java to implement the meta-programming approach. When tailoring activities changes the program the changes are implemented by compiling new class libraries (this is done by the compiler in JDK). The new class libraries are loaded and linked in during runtime. To obtain information of the program we have used the - rather weak - reflection abilities in Java reflection API.

2.3 Reflection in Java

Tailoring will change the program; the latter does not know in advance what the changes will look like. To discover what a new class contains, we need reflection capabilities. In a computational system reflection is the capability of an object to, for example, "reason about and act upon itself" (Maes, 1987).

There are two types of reflection: *introspection* and *intercession* (Rivard, 1996). The purpose of introspection is to acquire information about the program itself and to use that information within the program. Intercession goes further. It allows the program to alter its own behaviour. Different programming languages have different reflection capabilities. Languages such as Lisp or Smalltalk have both introspection and intercession, while Java is basically introspective only. Java's meta-model is shown in Figure 2 : 2. In Java, every class has a meta description which is represented by an object; an instance of the class "Class." This object is a metaobject. While ordinary objects describe

the world, metaobjects describe the ordinary objects. In other words, the metaobject is an object that contains information about the ordinary object (base object) (Golm, 1997). The metaobject may control the execution of the base object (Zimmerman, 1996). The metaobjects together with the ordinary objects are part of a meta-model.

The reflection API in Java provides information about modifiers, methods, instance variables, constructors and the super classes of a particular class. It allows you to create an instance of a class although you do not know the name of the class until runtime. It is also possible to invoke a method on an object without knowing the name of the method during coding.

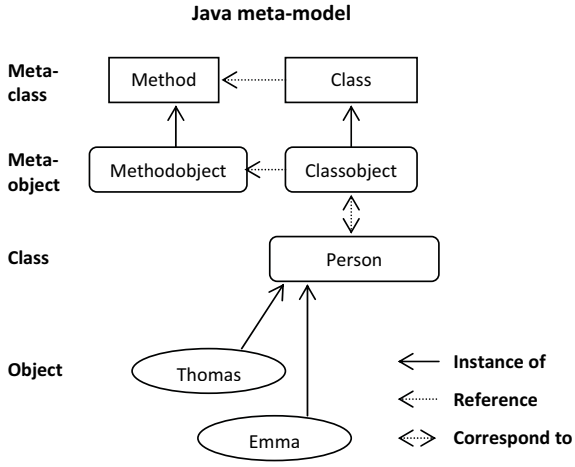


Figure 2 : 2 A part of the Java meta-model

Accordingly, it is java.lang.reflect which makes it possible to inspect the content of the class. However, in Java 1.3, the Dynamic Proxy API was introduced making it possible to alter the behaviour of an object in runtime. We have not used the proxy concept but instead alter the program by adding new classes using the compiler in JDK.

2.4 The System Architecture

The prototype was produced to test the use of MOP in implementing tailoring. The prototype is a partial implementation of a system described in this section. It is necessary to have some understanding of the whole system to understand the design of the prototype. The system is used for computing certain payments[2]; these payments are triggered by certain events. The receiver of

[2] To protect the business interests of our industrial partner we can only give an abstract description of the system: it is our opinion that this does not affect the conclusions we draw.

money and how much should be paid are decided by what contract(s) are valid for the event.

The system architecture is described in Figure 2 : 3. The system can be regarded as two loosely connected parts: the transaction handler and the contract handler. The transaction handler application manages the actual payments and also produces reports while its database stores data about the triggering events, payments and historical data about past payments. (1)[3] The data describing the triggering events is periodically imported from another system. (2) To compute the payments, the transaction handler calls a stored procedure in the contract handler's database. (3) The event is matched with the contracts; several hits may occur. Some of the contracts cancel others; others are paid out in parallel. We call the process of deciding which contracts to pay 'prioritization'. (4) The result is returned to the transaction handler. (5) The actual payments are made by sending a file to the administrative system.

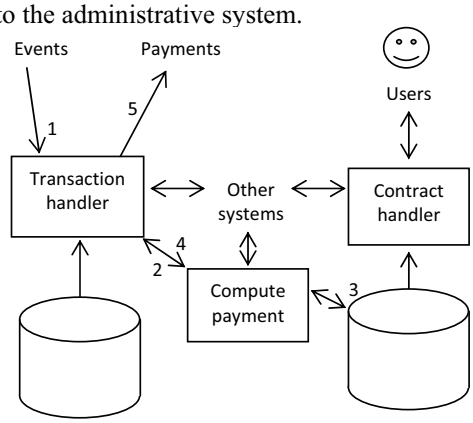


Figure 2 : 3 The system architecture

An important complication in the data model is the categorization of the values on which some of the conditions are based. The categorization is dependent on other systems, making interaction with the latter essential both when the transaction handler matches events with contracts and when a user wants to use categories in a contract.

The contract handler administrates contracts, or rather formal descriptions of contracts, in a relational database. The interface enables the user to enter new contracts and search for old ones. When entering new contracts, the input is checked to ensure the integrity of the data. The main parts of the contracts are:

- Identification of the contract and version control.
- Some flags controlling who receives the money.

[3] The numbers refer to figure 2.

- Conditions determining if the contract is valid for an event or not.
- A payment table deciding the amount to be paid.

The first two parts are common to all contracts; it is the conditions and the payment table that differ.

In order to make the system adaptable to future changes a conceptual model that facilitates a meta-model description of the system is needed. The purpose of the system is to compute payments according to the stored contracts. Each event that triggers a payment has a set of parameters. Today there are only two kinds of events, though several other types of events are under consideration for the future. In the contracts a condition is meaningful only if the transaction handler can evaluate it when payment is due. This leads to the concept of event types: a payment is triggered by an event, and all contracts belong to a particular event type. Each event type has a set of attributes associated with it that limits what conditions a contract belonging to it can have. In the existing system there are a number of contract types that are used for different purposes. From the system's point of view these contract types differ in two significant ways: which conditions you can add to the contract and how the contracts influence each other (if several contracts match the same event one may inhibit the others, or all may be paid out).

Already during the design discussions we constructed a conceptual model with four levels of abstraction (see Figure 2 : 4). The actual data that is stored describes the contracts the payments are based on. The contracts are of several *contract types* which form the base level of the abstraction hierarchy. Some contract types has nearly the same parameters but are used for different purpose in the use of the system; this gives the next level, *contract_group*s. At the top level of the abstraction hierarchy are the *event_types* where we group together contract and payments related to the particular event which triggers them.

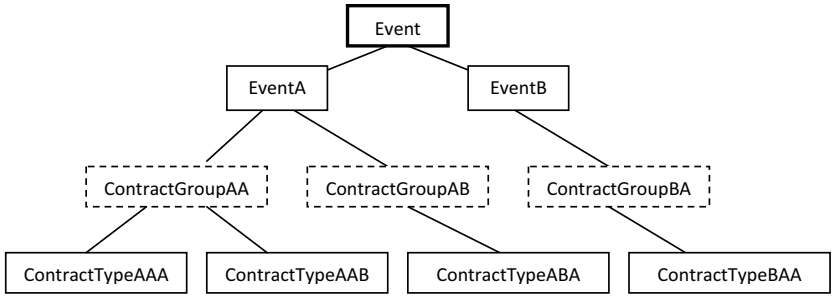


Figure 2 : 4 Type hierarchy

In an object-oriented implementation the actual contracts would be objects belonging to the concrete classes in the bottom line. The remainder of the classes would be abstract.

2.5 The Prototype

The reason for producing the prototype construction was to investigate the feasibility of using Java's meta-programming possibilities to construct a tailorable system. This can be seen as an example of an explorative prototype (Floyd, 1984). We wanted to gain an understanding of the complexities related to this approach. The prototype does not implement the whole system but only the contract handler application. Functionality is reduced, especially the parameters using categorization of values are simplified to simple values, the primary reason for this being that it allowed us to build a prototype without any communication to other systems; in this way development of the prototype was greatly simplified.

The prototype is divided into two levels, the meta-level and the base-level. Two catalogues, one storing contract type and the other parameter classes implement the connection between the two levels. In the meta-level of the prototype, the new contract types are created and stored in the contract type catalogue. In the base-level the same classes are used as part of the program. The parameter class catalogue is used by the meta-level to know which parameters exist and by the base-level as part of the program.

Inheritance, together with the meta representation and the inner structure of the contract types, is essential to the prototype. A simplified model, similar to the conceptual model described in the section about the system architecture – but leaving out the group level – is the basis for the prototype implementation. It resulted in the class hierarchy presented in Figure 2 : 5. The events are super classes to the contract types. In the conceptual model an Event has a set of parameters and the contract type is made up of a subset of these parameters. This is not possible in Java; instead there is a specification that defines the set of parameters for the contract types in the Event classes. Some parameters are compulsory for all contract types belonging to an Event; they are put in the Event so that by inheritance they are present in all the contracts, e.g. all contracts must have a contract id.

One problem is that the contracts should be stored for a long time; all contracts ever entered into the system are kept to preserve its history and ensure that old payments are traceable. This is a problem when a change is made in a contract type as the system must still be able to store and display old contracts according to the old type. For this reason, all contract types from the beginning are defined as both an abstract class (e.g. ContractAB) and a concrete subclass (ContractAB_1). When a minor change is made to a contract type this may then be done by making a new concrete class, as ContractAB_2 in Figure 2 : 5.

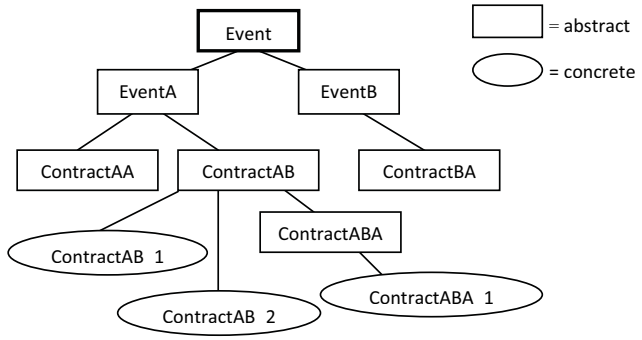


Figure 2 : 5 Inheritance hierarchy for the contract types

2.5.1 The Base-Level of the Prototype

A contract is essentially a collection of parameters. In the system in use some of the parameters are very complex and some even collect values from other systems. This makes it natural to represent every parameter by an object. Most of the methods in the contracts are implemented using delegation to the parameters. For the contracts' three main methods - checking, storing and displaying themselves - there are corresponding methods in the parameter classes. This is a vertical design where one class takes care of one type of parameter through the whole program instead of the more normal three-layer architecture (interface, logic and storing). This design makes it very easy to add new parameter classes to the system.

When the end user wants to create a new contract, i.e. create an object from a contract type, all of the concrete classes are fetched from the contract type catalogue, their names are presented and the end user chooses which contract type to create a contract from. Then a contract is created which has parameter objects without values. The object displays itself by delegating to the parameters. The same principle is used for storing and checking errors. When the user has put values in all slots and wants to store the contract, the error check is delegated to every parameter object. The parameter object checks that the value has the right format and is within the given limits. When a value is incorrect, the slot is marked and the user has to put in a new value. Not until all values are correct, are the values set in the empty contract. The primary problem with the delegation principle is that it is inadequate where parameters are in some way dependent on each other. It is possible for a parameter to access another parameter within the same contract by using a parent reference that all parameters have.

Following is a summary of how to create a new contract:

- The prototype collects the contract types from the contract type catalogue.
- The end user selects a contract type to make a contract from.
- An empty contract is created from the contract type, the display method of the object is called and the parameters display themselves to the user.
- The user puts in values for the parameters.
- The prototype checks the values; when these are correct they are set in the empty contract. A contract is created.
- The contract is stored in a similar way.

2.5.2 The Meta-Level of the Prototype

The contract types are created in the meta part of the program. When a user wants to create a new contract type all existing contract types are displayed. This is done by collecting all the class files from the contract type catalogue in which they are stored. The end user chooses what contract type he wants to have as super class for the new contract type. To make it easier for the user to make a decision as to what contract type is the most suitable, the parameters and the methods of the contract type are also displayed. Java reflection API provides the necessary methods for this.

The next step is to collect all possible parameters for the new contract type. To find the set of all possible parameters the program collect all classes in the catalogue dedicated for parameter classes. All parameters may not be used for all Events. This is achieved by putting a filter in the class describing the Event. For example, if a parameter 'xyz' is not valid for Event type B a method that acts as a filter is placed in the abstract class EventB (Figure 2 : 5).

Thereafter all possible parameters for this Event type are shown to the end user for him to select from. The parameters that are inherited are automatically selected and cannot be deselected. To find which parameters are already present in the selected contract type the program looks into the class of the contract type and its super classes with help from java.lang.reflect.

The meta-level of the program is constructed as a meta-model which is implemented as classes. The contract types correspond to objects of the class Metaobject. Our metaobject is in a way the same thing as the classobject in Java. We constructed our own version because a classobject cannot exist without a corresponding class. This means that it is not possible to create a new class from a classobject; as a result we could not use the classobject alone for our purposes. Another factor is that it is important to be able to handle the metamethods in a special way. The relationship between Java's meta-model and our extended meta-model is shown in Figure 2 : 6.

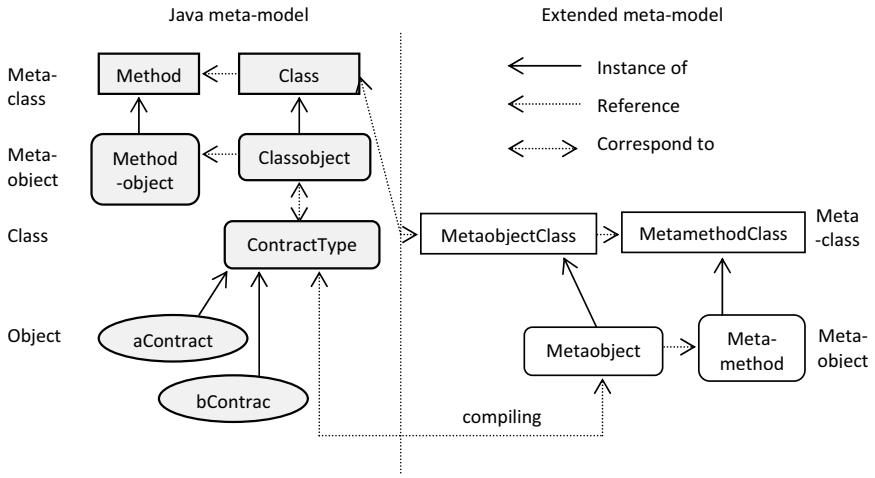


Figure 2 : 6 Meta representation

In our extended meta-model a metaobject is an ordinary Java object, but it contains a description of a contract type and thus corresponds to a specific contract type. The MetaobjectClass is the class of metaobjects. The MetaobjectClass is a description of a general class. When the MetaobjectClass is instantiated the fields acquire values. The fields are references to metamethod objects, metafield objects and metaconstructor objects, e.g. the MetaobjectClass has a field of the Metafield type (the metamethods and the metaconstructors are excluded to simplify the example). The MetafieldClass has the field's *name* and *type*. When the MetaobjectClass is instantiated a metaobject and a metafield object are created and the fields in Metafield acquire their values. The metaobject has a reference to the metafield object and the latter has a reference to a parameter. If the contract type is to have a parameter named *aCustomer*, the metaobject has a metafield object with an instance variable *name* with value "aCustomer" and an instance variable *type* with the value of "Customer". (Figure 2 : 7).

When the user has made his or her choices as to which parameters the contract type is to contain, the class ContractHandlerMOP creates the metaobject according to the input values. From the metaobject the source code for the new class is generated. The java source code is then compiled and a class file is produced. The file is stored in the contract type catalogue.

The ContractHandlerMOP is a class that handles the metaobject. All access to the metaobject goes via the ContractHandlerMOP. The class also restricts what can be done to the metaobject. This can be used to implement business logic controlling what contract types can be created.

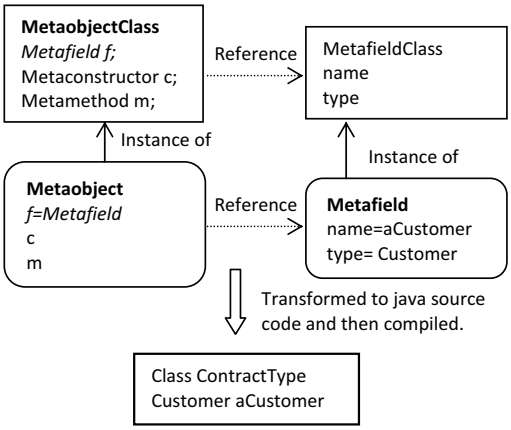


Figure 2 : 7 An example

Following is a summary of how to create a new contract type:

- The prototype collects the contract types from the contract type catalogue and displays the names of the contract types and their parameters and methods to the end user.
- The end user selects a contract type on which the new one is to be built.
- The contract type is inspected and the parameters of the contract type are displayed. All the parameter classes are collected from the parameter catalogue and filtered by the Event type; the result is displayed for the user.
- The user chooses the parameters for the new contract type.
- The program constructs a corresponding metaobject with its metafield, metamethod and metaconstructor objects.
- The metaobject is translated into Java source code.
- The Java source code is compiled and the resulting class file is stored in the contract type catalogue.
- The contract type can be used by the base-level of the prototype.

2.6 Discussion

During the project three prototypes were implemented along with the system that is in operation today. The last prototype is the one that is described in this article. The other two prototypes were for the contract handler (with essentially the some functionality as in the prototype described) and for the "compute payment" function respectively. These two prototypes were constructed with the help of the meta-model database that was the starting point of the project. They

are examples of the *meta-data approach* mentioned in the Section 2.2 above, for a description of these prototypes see (Lindeberg and Diestelkamp, 2001). There are parallels between using a meta-model database and the MOP prototype. In the meta-level of the program the meta database structure and the object structure of the program are inspected respectively. The difference comes in the base-level part: the meta database prototypes were both complicated and slow since it had to inspect the database to establish the structure of a contract type; it also had to inspect the database to see how a parameter looked.

When we compare the earlier prototypes with the one described in this article the latter is less complex (it has taken less time to develop it). The interesting question is why this is the case and it is important to see if we can draw any general conclusions from this.

One of the reasons why meta-programming was so convenient in the example described here is that it is not the functionality of the program which is changed by the tailoring interface but the model of the data in the program. The base-level program has the same functionality in spite of the alterations. If the tailoring had aimed at extending functionality, for instance, with the aid of macro capabilities, the task would have been complicated in the meta-programming approach. An interesting question is if there is a complementary principle here: when tailoring changes functionality use meta-data approach and when tailoring changes the data model uses the meta-programming approach. Our results seem to point in this direction.

Another advantage of the MOP prototype is the loose coupling between the meta and the base part of the program and between the contract types, the parameters and the base-level. This makes the base-level part simpler. By separating the meta- from the base-level we were able to use standard software, which means that at least the base-level is maintainable without any special competence in MOP.

Jet another advantage of the MOP approach is the opportunity it presents to handle unanticipated changes by hand-coding objects. There is always a limit to how far we can get with tailoring since the latter only takes care of anticipated types of changes and there will always be changes in the requirements which cannot be anticipated. In the MOP prototype, hand-coding contracts or new parameter classes can handle some such changes. This goes beyond normal tailoring activity and is part of the evolution and maintenance of the system. The advantage of the MOP approach described here is that it is easy to mix hand-coded and automatically constructed objects.

A new contract type can be coded by hand and put in the contract type catalogue. It will be used in the same way as contract types constructed within the program. Such a hand-coded contract type can be modified later by regular tailoring.

One example of this could be a contract type where two parameters depend on each other; if one parameter has a value the other must also have one. We can implement such an example by first using tailoring to let the system construct the contract type without any check between the parameters. Then a programmer can modify the code by adding the constraint between the parameters to the checking method in the contract type. Should we subsequently wish to make a small modification in the contract type, by adding a parameter, for example, this can be done using the normal tailoring interface.

In the same way it is possible to add new parameter objects by simply placing the compiled parameter class in the parameter catalogue. The parameter class is then ready to be used in the usual way by the program; no other code in the system needs to be changed but the new parameter class must obviously be hand-coded by a programmer. The new hand-coded contract type or parameter class must follow the pattern for how a contract type or a parameter class has to be structured. We believe this possibility of mixing hand-coded and automatically generated objects is a general advantage of the meta-programming approach.

One of the reasons that tailoring was not implemented in the real system development that was part of our research project was that the automatically generated user interfaces would not have been of an acceptable quality. This is a problem that occurs whenever tailoring is used to generate user interfaces. The meta-programming approach enables the user to alleviate the problem by making hand-coded interfaces for the contract types that are in the system right from the beginning so that they have good user interfaces. When new contract types are subsequently added by tailoring, less user-friendly interfaces will result; this may be acceptable, and in our case study it would have been an option since the alternative is to handle payments by hand.

2.7 Concluding Remarks

It has been interesting to try out the possibilities in Java for carrying out meta-programming. Our overall conclusion is that the metaobject possibilities available in Java are a convenient way for implementing tailoring in special-purpose applications.

A question we have only touched on in this paper is if it is worth the trouble to make an application tailorable as opposed to being merely “easy to change”. The answer to this question lies in the future: what types of requirement changes will arise? Would the prototype have been able to handle them? After all, the efforts to make software adaptable only pay off if they are used.

Chapter Three

Paper II

Chapter Three

An Adaptable Architecture for Continuous Development

User Perspectives Reflected in the Architecture

The 26th Information Systems Research Seminar, IRIS 2003

Jeanette Eriksson, Peter Warren, Olle Lindeberg

In this article we report from a design study that was performed at the Space and Virtuality studio at the Interactive Institute AB[1] in Malmö. The research team at the Space and Virtuality studio is exploring how information technology can support different design processes, such for example art projects. In a design process the end users can experiment with different material, media, situations, interactions etc. The intention with the design process may be to explore different possibilities to interact with media or it can be purely artistic. For example discover what artistic expressions you can get by combining different sounds or learn how to combine light and digital projections to create a desired effect in a room. The requirements for computer support change dependent of the design task but also during the process itself.

The persons engaged in the research team have different kind of competences and they cooperate intimately to achieve new design settings. The research team at the Space and Virtuality studio is also exploring the area of ubiquitous computing (Weiser, 1991) in the context of design processes. They are developing a system of ubiquitous and intelligent building blocks, called ActionBlocks (ActionBlocks are comparable with Phidgets (Greenberg and Fitchett, 2001), but ActionBlocks is more of an interaction metaphor. ActionBlocks is used to structure the interaction, which makes it possible to build functional prototypes fast. ActionBlocks are input and output devices (tag readers, digital cameras, loudspeakers, lamps, buttons etc.) that can be used in different projects exploring the interaction between the end user and digital media.

To be able to understand the complex functionality and quality requirements for the upcoming ActionBlock system we have worked closely together with the future users. We have participated in workshops, in different projects and discussions concerning the use and functionality of the future ActionBlocks system. The work made us aware of the close cooperation within the project groups. When some new kind of IT-support is needed in a project the participants discuss and negotiate the functionality, the quality, the look and the

[1] http://www.interactiveinstitute.se

delivery time and a solution is finally reached. The frequency of this type of work makes it desirable to have a 'library' of different ActionBlocks that can be assembled easily and quickly into different applications with different functionality. This brings about a need for some kind of flexible architecture that supports this kind of dynamic use.

During the work it became visible that different persons interested in the future system have different views of ActionBlocks and how they ought to work. The views were connected to what intentions the person has with ActionBlocks and what activity the person focus on. The intention could be to use a system that facilitates the interaction with digital media or it could be to design different situations or possibilities to interact with the system, in other words, to set up different environments for interaction. But it also turned out that the activity to design or develop the ActionBlock was a main concern for some persons. From the discussions and workshops it is possible to distinguish three different roles, the end user, the interaction designer and the ActionBlock designer. The three roles use the system in different ways.

A flexible system is important for all the roles, but it has different meaning. Flexibility for the end user is that he can change the use. To the interaction designer flexibility is to be able to assemble the system he desire on his conditions. While the ActionBlock designer regard flexibility as a way of changing or develop the software in a convenient way. All three roles contribute and use the flexibility and in this way they all play a role in the evolution of the system. The development takes place continuously and the maintenance will be a part of the development. All three roles are part of the development process and thereby they are all equally important. Thereby all situations of use are important to consider when designing the system.

In conversation, an interaction between humans, context, assumptions and other things has to be taken for granted to be able to concentrate on some particular aspect, action, topic or objective (Robinson, 1999). This can be said to be true for the interaction between humans and computers too. Different aspects are important for the user depending on the situation of use. This means that a user disregard some aspects of the system because they are of minor importance for his specific use of the system. This is what makes different kinds of users have different perspectives on a system. Apparently, the three roles have different requirements of and perspectives on the system and this raised the question if it is possible to construct an architecture that reflect these different user perspectives and fulfil the user requirements that go beyond the user interface, for example how the ActionBlocks communicate. The adaptable architecture we present in this article accomplishes this.

Many research groups experiment with different kinds of sensors and environments concerning ubiquitous computing and they suggest various approaches towards the infrastructure. We have studied several related systems that are concerned with making everyday life easier by adding computational

support in the background and/or bringing the physical world into the virtual. We have studied Multiple trivialities that is a project performed at the Space and Virtuality Studio[2] Web presence (Kindberg *et al.*, 2000), Appliance Data Service (Huang *et al.*, 2001), Interaction spaces (Winograd, 2000), The Weather Alarm System (Jacobsen and Johansen, 1999), RFID Chef (Langheinrich *et al.*, 2000), Informative things (Barret and Maglio, 1998), Invisible interfaces (Want *et al.*, 1999), Hive (Minar *et al.*, 1999) and JINI (Waldo, 1999). When studying the different systems we have looked for implications that different kinds of usage is taken care of in the architecture. The studied systems has architectures that are suitable for one or two of our user roles, none fits all three roles. In the descriptions of the systems we have not found any discussion about how to make an architecture that fits different user perspectives and roles. Our own work was driven by the question if it was possible to construct a system architecture that fit the requirements of all three roles. We wanted to show that it is possible to reflect user roles in system architecture and use it as a platform for system design.

Throughout the work we have used scenario-based design that is a well-known and accepted design representation (Carroll, 1995). We have used scenarios to envision different situations of use and just like in (Caroll, 1998) we see requirements as statements of situations of use. The use of scenarios makes it possible to visualize even cognitive aspects like expectations, goals and former experiences. In this article we also use scenarios to extend the comprehension. We let the opening scenario in the ActionBlock section extend into the further sections to clarify how the different roles are supported by the architecture.

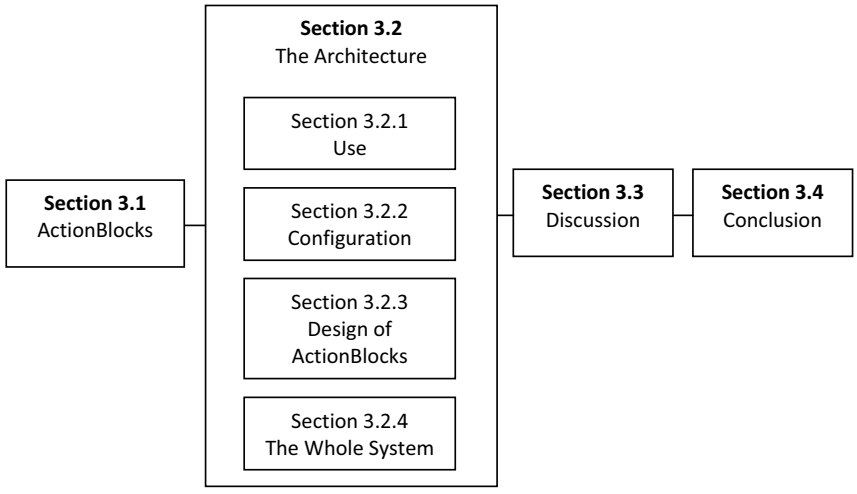


Figure 3 : 1 Overview of Chapter Three

[2] < http://w3.tii.se/project.asp?project=104>,

The rest of the chapter is structures as follows (Figure 3:1). We first start with a scenario that visualizes what ActionBlocks are. After that we present the architecture and how it reflects the three user roles. We also exemplify the architecture by parts of the proof of concept prototype that implements the architecture. Then we discuss our result and present some arguments for why the architecture is appropriate for all three users. Finally we make a conclusion of the work.

3.1 ActionBlocks

The primary idea with ActionBlocks is to be able to make experimental designs fast, primarily the ActionBlock concepts is for making it possible for the interaction designer to experiment with different designs. Basically an ActionBlock is a physical device that interacts with its environment and function as an input and/or output device for the rest of the system. An ActionBlock may be almost any electrical device: a tag reader, a digital camera, a video camera, a projector, a button, a lamp, a loudspeaker etc. An ActionBlock can be regarded as a part of physical interface to a ubiquitous computer system. An end user can by manipulating a physical thing (that is part of an input ActionBlock) make the system react and the system can cause an action in the real world, the action is done by an output ActionBlock. The intention is that systems of different ActionBlocks may easily be constructed to support interaction with digital media. When this work was performed only a few ActionBlocks were constructed, but the intention is that the ActionBlock family will expand. To clarify the use of ActionBlocks let us take an example. Suppose we have the following simplified scenario:

Carl is a teacher at the Interaction Design program at Malmö University. Every year the master students have an exhibition to expose what they have done during their studies. This year Carl that works part time at the Interactive Institute AB wants to enrich the visitors' experiences of the exhibition by making them a part of it. His idea is that every exhibitor provides the visitors with items that are associated to a film, an image or a sound that have some relation to the exhibitors work. When a visitor, let us call him Jan, visits an exhibitor he chooses an item that appeal to him. He can collect several items from different exhibitors. At a central place in the exhibition hall Jan can make his own multimedia show by putting his items on a table with a tag reader hidden beneath it. When an item representing a film or an image is put on the table the film is shown on a big screen and when an item associated to a sound is put on the table the sound is exposed for everybody to hear. This is possible because the items are tagged with small electronically tags.

To realize his idea Carl need several ActionBlocks: one tag reader for the central table, one tag reader for each exhibitor, one projector and a loudspeaker set. Carl also needs some software to support the application. He needs a piece of software that takes care of the exposure of the different media. Carl then asks Minna, the ActionBlock designer (the programmer) at the studio, to construct

that software. When the software is ready Carl can configure his system by fetching all the needed ActionBlocks and connect them to the network. The ActionBlocks is then shown in an interface along with the available software. He draws lines between the different unites to put them together. When he is finished he saves the configuration and when it is time to use the system he activates the system. The necessary software is downloaded to the different parts of the system and the system is ready to be used.

In the scenario three different persons interact with the ActionBlocks system, Jan, the end user, Carl, the interaction designer and Minna the ActionBlock designer that supply the technical solutions. The three has different concerns about the system. Jan wants the system to be easy to use and understand, but it also has to be interesting to use it. It shall enrich his experience. Carl wants the system to be flexible so that he easily may alter it if he wants to change the interaction with the system. Minna wants the system to be easy to maintain and develop further. The different roles also have different perspectives on the system. Even if one person can act in all three roles, they can be regarded as distinct ways to interact with the system. In many other scenarios the ActionBlock designer has no role at all; if the interaction designer can construct the needed system by assembling and connect ActionBlocks that already exists.

As implicated in the scenario above the participation in the work with ActionBlocks lead to an identification of three different roles (end user, interaction designer and ActionBlock designer) that are stakeholders in the future ActionBlocks system. The roles have emerged from discussions with people at the Space and Virtuality Studio. People that mainly work with software development, interaction designers and people that represent the user view were involved in these discussions. The persons that shared the user's view stated there has to be a correlation between the user role and the functionality and appearance of the ActionBlocks.

3.2 The Architecture

By analyzing the different perspectives on the system the basic concept of ActionBlocks was refined into a more structured concept which is represented by the puzzle in Figure 3 : 2. At a conceptual level an ActionBlock is an artefact that exists in both the physical and the digital world. ActionBlocks consist of a computational (intelligent) part and a physical part (Figure 3 : 2). The physical part contains a physical item and an action. The action is what happens when the physical item is manipulated. The physical item is the part of the ActionBlock that the user can touch and see. The computational part contains both hardware and software. The computational part consists of a symbolic representation of the physical item, a logic that makes the computation and a digital representation of the action, an event. The physical item and the action have its place in the real world, but to achieve an action the computational part is needed.

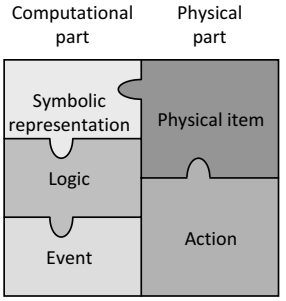


Figure 3 : 2 Conceptual model for ActionBlocks

When an action is made on a physical item the computational part comes in use. The symbolic representation contains basic functionality dependent of the type of ActionBlock and translates the signals from the physical item and transfers the data to the logic. The logic work on the data and then the event is activated. An ActionBlock that is creating an action react in the reverse order. The different roles focus on different parts of the ActionBlock depending of their perspective.

A proof of concept prototype[3] that simulates different ActionBlocks assemblies was made to test the validity of the concept. The implementation resulted in a loose coupling between the three perspectives in the architecture as shown in Figure 3 : 10. The prototype is built in Java. XML and JXTA (a set of peer-to-peer protocols (Mason, 1996) are used for descriptive documents and communication. Bellow some parts of the prototype are presented to exemplify how the architecture can be implemented. The description of the prototype is disregarding the specific JXTA dependent issues. This is done to be able to keep the description quite simple to highlight the main issues. The prototype worked well but it does not contain a physical interface. The prototype showed us that it was possible to adapt the architecture to different user roles and still get a complete comprehendible system.

3.2.1 Use

When Jan arrives to the exhibition hall he moves around the exhibition and at one place he picks up a smooth stone that he puts on a tag reader nearby and suddenly he can hear the sound of children's laughter when they are throwing stones in the water. It remembers him of the summers in his childhood. He put the stone in his pocket and moves along. At another place he picks up a sample of seaweed that seams to go well with the stone. Eventually he arrives to the

[3] For an extended description of the prototype see Jeanette Eriksson, Interaction Views in Architectures for ActionBlocks, Master theses in Computer Science, 2002, Blekinge Institute of Technology (http://www5.bth.se/fou/cuppsats.nsf/1d345136c12b9a52c1256608004f0519/4ac2f2585884b670c1256c1a00433023/$FILE/ ToEachHisOwn.pdf)

central table and he puts the seaweed on the table. The projector immediately shows a film of a man fishing at a calm lake. The only sound that can be heard is some birds singing. Jan then puts his stone at the table and the children's laughter is heard and the film clip gets a totally new expression.

For the end user the power of ubiquitous computing lays in direct and simple interaction. It is essential that there is a direct coupling between his manipulation of the physical object and the system's response. The physical part of ActionBlocks is in focus (Figure 3 : 2). The system should also be robust, if one part is missing or defect only that part of the system should stop working. This perspective and requirement can be supported by a pure peer-to-peer structure (Figure 3 : 3).

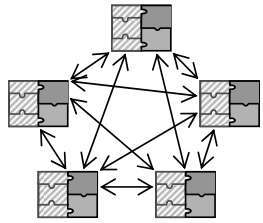


Figure 3 : 3 Pure peer-to-peer architecture

In use there are some ActionBlocks that the user interacts with and some ActionBlocks that produce actions. Cheap web servers, called TINI[4] integrate the ActionBlocks. The ActionBlocks may be combined in various combinations and it is the logic that makes the combination possible.

Every type of ActionBlock has a specific functionality. To be able to speak of ActionBlocks in a more general way we introduce transmitters and receivers. A transmitter is an ActionBlock that has as a task to send tag id or other types of translated signals to a receiver that performs an action. An ActionBlock may act as both a transmitter and a receiver at the same time if such a circumstance occurs. A camera might be such an ActionBlock. The camera can be controlled from for example a tag reader and then it may send the pictures to a projector. The gist of the example is that the ActionBlocks know by themselves what to do. They do not have to take help from some central server that holds all the information.

When Jan, the end user in the scenario interacts with the system the constellation of ActionBlocks may look like in Figure 3 : 4. The projector in the example might not be able to store the films because of lack of memory space. It has to make a request to some type of database. Several ActionBlocks might share this database. This database is not really regarded as an ActionBlock due to the definition of ActionBlocks as a unit that contains an intelligent and a physical part. At least it does not have a physical part that the end user comes in

[4] http://www.maxim-ic.com/products/microcontrollers/tini/

contact with. But this type of unit has to exist and to be able to speak of this type in a general way it goes under the name of responder.

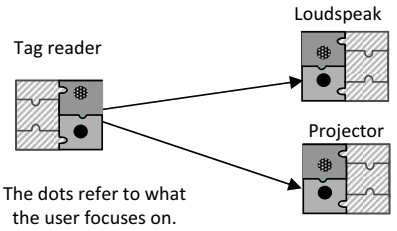


Figure 3 : 4 ActionBlocks in the exhibition hall

What happens within the ActionBlock when it is used can be exemplified by how it is done in the prototype. When Jan puts a tagged object at the central tag reader the signal from the tag reader is translated by the symbolic representation (that also contains basic functionality of the tag reader). The data is transferred to the logic in a XML file. In this case the logic adds the tag readers name to the data and transfers it to the event module that handles the communication and it sends a JXTA message, containing the XML file, to the loudspeaker set. The event module takes care of the message and forwards the data to the logic. The logic works on the data and dependent of the tag id and the tag reader name a decision is made of what sound and sound level to expose. This information is added to the XML-file. The result is sent to the symbolic representation that transforms the data to signals that the physical loudspeaker understands and the sound is exposed.

3.2.2 Configuration

To be able to use the system, the use has to be proceeded by a configuration. The participant in this phase is the interaction designer. The interaction designer is the person that set up the connection between the physical item, the computational part and the action. He connects physical objects with software units. He configures the system. For him all three components in the ActionBlock have the same dignity (Figure 3 : 2). He sees the parts as components that can be combined into ActionBlocks that fulfil his requests. He thinks of software and physical item as one unit, an ActionBlock. The interaction designer also designs the interaction between the ActionBlocks and between the system and the end user. The system should make it easy to design system with direct coupling between the manipulation of the physical item and the action. It is practical to think that the interaction designer configures the system by a drag-and-drop interface (Figure 3 : 5).

When the interaction designer configures the system he chooses what logic to use for which ActionBlock and when he saves the configuration it registers what logic and ActionBlocks the configuration require. The file is then used when the system is activated.

To the interaction designer it is also appealing that the assigned logic really resides in the intended ActionBlock, because it corresponds to how he handles the ActionBlocks while configuring the system. Peer-to-peer architecture also has the advantage that the system is scalable and it is easy to join the network. It is just to start a peer (Minar et al., 1999) and then the peer itself takes care of the communication with other peers. These things are important from the configuration perspective.

To be able to set up a system it is important to the interaction designer to be able to easily obtain a complete list of existing and connected ActionBlocks to be able to configure the system and assign logic to the ActionBlocks. Such a list can be obtained by a distributed service that may request the ActionBlocks in the network for a description of them, and in this way be able to show them as icons in the interface.

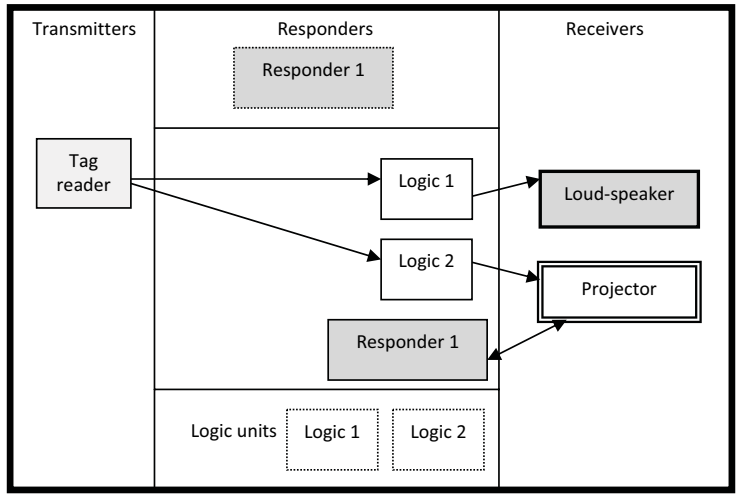


Figure 3 : 5 Schematic interface to configure a system.

It is possible to configure the system even if the ActionBlocks are not connected to the network. This can be done by the fact that all the ActionBlocks are tagged. To make it possible to show the ActionBlocks in the interface, tag id together with a description or image of the ActionBlocks has to be obtained in a database. If the tag is known, the icon is shown. Then the system can be configured in the same way as before. If the ActionBlocks are disconnected while configuring the system the architecture may be regarded as client-server[5]

[5] The distinction between a server and a distributed service might seem a little bit unclear. The main issue is that the distributed service may be located on several places and isn't exclusively serving a set of dependent clients. But at some time slot a computer that answers a request acts as a server to the computer that sends the request despite which functionality the computers has in the network.

architecture while the application requires information about the different ActionBlocks and logics, which can be requested from a computer that supplies that service. But the computer with the service may also be a peer in the system while the architecture may be regarded as a peer-to-peer architecture with distributed service (Figure 3 : 6). The interaction designer is gained by a peer-to-peer architecture with distributed service.

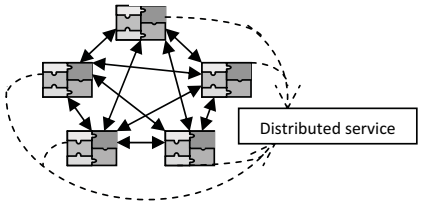


Figure 3 : 6 Peer-to-peer architecture with distributed services

The prototype does not implement the graphical interface that the interaction designer uses to assemble the ActionBlocks into a system. The intention is that when the interaction designer creates an application the ActionBlocks, logic etc that it is going to consist of is saved in a XML file (1) (The numbers refer to the numbers in Figure 3 : 7. In the prototype this file is written manually for the different assemblies.

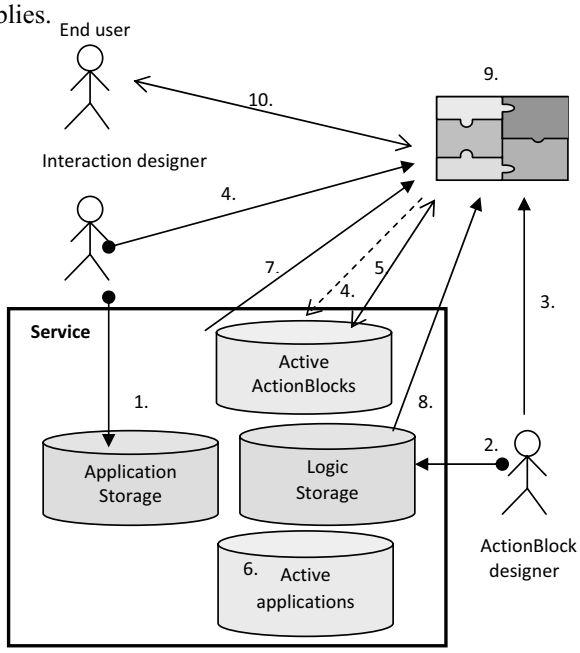


Figure 3 : 7 Functionality of the prototype

But the setup is also a part of the configuration. At setup the system gets ready to be used by the end user. When the interaction designer makes his intention to

start an application known to the system (4) a start message is sent to the distributed service. The distributed service continuously discovers which ActionBlocks that are active (5) so when the start message arrives the distributed service can examine the XML document that contains the application description to explore if there is a match between the required ActionBlocks and the active ones. If so and there also is another application running the use of ActionBlocks is coordinated (6). An application may allocate an ActionBlock, but it is also possible for two applications to share ActionBlocks. This is defined in the XML file describing the application. When everything is checked and it is possible to start the application, a message is sent to the ActionBlocks that are going to be a part of the application. The message is an XML document containing information about who to communicate with (7). The symbolic representation (Figure 3 : 2) contains an XML document that defines the name, type, functionality etc. for the ActionBlock. In this way the ActionBlock already knows if it is a transmitter or a receiver. The next step is to upload the required logic to the intended ActionBlock (8). The system is now ready for the end user to use the application (10).

3.2.3 Design of ActionBlocks

The ActionBlock designer is the person that develops the software to support for basic functionality for new or altered ActionBlocks. The ActionBlock designer therefore focuses on the software and what actions it can generate. He focuses on the computational part. To him the physical item and the action are represented in the computational part. It is the ActionBlock designer that has the control over what actions that can be associated to a physical item because it is he who constructs the software. The ActionBlock designer designs ActionBlocks. The ActionBlock designer also handles the evolution and maintenance. From his point of view an ActionBlock may consist of only the intelligent part and the action. The ActionBlock designer is aware of how the software shall be used but for him the physical appearance is of minor importance. For the ActionBlock designer it is of importance that the development and the maintenance are easy to perform. It has to be easy to update and administrate new software versions. It also has to be easy to overview available software. With this approach it is convenient to make all substantial computation at one place, at a server. From the ActionBlock designer's point of view the clients are input devices that make requests to the server that is a computer that serves the client with the required service (Capron, 2004). For example, supply different ActionBlocks with information about which to communicate with or required logic. It is also much more easy to maintain and update a system with client-server architecture (Huang et al., 2001) where many of the resources are kept at one place. It is also easy to keep a list over existing clients (Minar et al., 1999). From the ActionBlock designers perspective a centralized architecture, client-server, may be preferable as architecture for ActionBlocks (Figure 3 : 8).

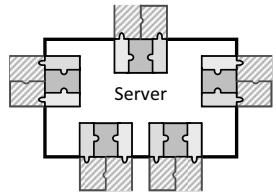


Figure 3 : 8 Client-Server architecture

Let us take a look at Minna's task in the scenario. When Carl asks Minna to make some new logic for the exhibition she has to construct two pieces of software: one for the loudspeaker set and one for the projector. The projector has never been used in an ActionBlocks system before and therefore there is a need for some general software in it.

What Minna has to do is to equip the projector with an event module and a symbolic representation. The architecture for the ActionBlocks define that the part called Event (Figure 3 : 2) handles the event generated by the user and the events from the network. That part is the same for all ActionBlocks, so that piece is just to upload to the projector. The symbolic representation has to be constructed. This representation is divided into two parts. It contains the software that handles the basic functionality for an ActionBlock, for examples signals from tags has to be translated to a form that the logic can understand. This part is specific for each kind of ActionBlock (tag reader, button, projector etc.). The other part is a description of the ActionBlock, what type it is (receiver, transmitter), what sort it is (tag reader, projector), name, id etc. When this is done Minna turns to constructing the logic for the projector and the loudspeaker set. Minna has to make her new software adapt to a predefined interface to the event module that handles the communication. The use of ActionBlocks differs during time and no project is similar to another therefore it has to be easy to put different logics together. They will be software components. All the components have to adapt to a general interface. Let us take a closer look at the projector logic. What is needed? The projector is going to show images and films. One component for displaying images and one for displaying films are required. But to be able to make the decision what type of media to display a third logic unit is needed, a part that holds the information about which tag is associated to which file (Figure 3 : 9), the *direction guide* (this part might get assistance from a responder, but we disregard that possibility for now). The same principles go for the loudspeaker and the spotlight switch. While reusing the logic in another combination the *direction guide* will be different.

If there is a need for new software the ActionBlock designer makes and test the required software and saves it in the “logic storage” (2). If basic software (for example symbolic representation) is required in an ActionBlock the ActionBlock designer uploads that too (3).

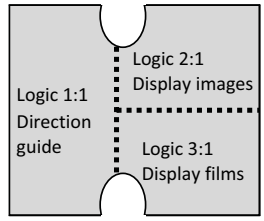


Figure 3 : 9 Logic for the projector

3.2.4 The Whole System

We have seen that different architectures are preferred dependent of the user role. From the discussion above a concept for the system architecture combining different architecture paradigms, to support all three roles, is derived. The architecture can on one hand be seen as an example of requirement driven design where we first analyze the three roles different requirements on the system and then construct an architecture that fits the requirements. On the other hand the architecture starts from the concept of ActionBlocks which essentially is taken as an on forehand given starting point for the whole design.

The concept of the architecture is on a rather high level because its flexibility on different levels. The details in the architecture are determined by the specific use situations and the implementation. The architecture is a manifest of the design decisions that equalizing the different roles and to focus on one role at a time and then combine them. To keep the concerns separated sub concepts are developed for each user role. The different sub concepts fit well together and form a merged concept for an architecture (Figure 3 : 10) that will adapt to the different roles when interacting with the system.

When the user use the system the ActionBlocks communicates directly. The ActionBlocks can be of different kinds and numbers. The interaction designer uses the system by putting tagged ActionBlocks on a tag reader and trough a configuration interface. The *manager* manages these units and the data is collected in databases. When the ActionBlock designer uses the system he constructs new software (logic) and uploads it to the *logic storage*. As shown in Figure 3 : 10 the three roles interact with different parts of the system and it is only at setup, performed just before the end user is going to use the system that the different parts are connected. This accomplishes a low coupling between the parts.

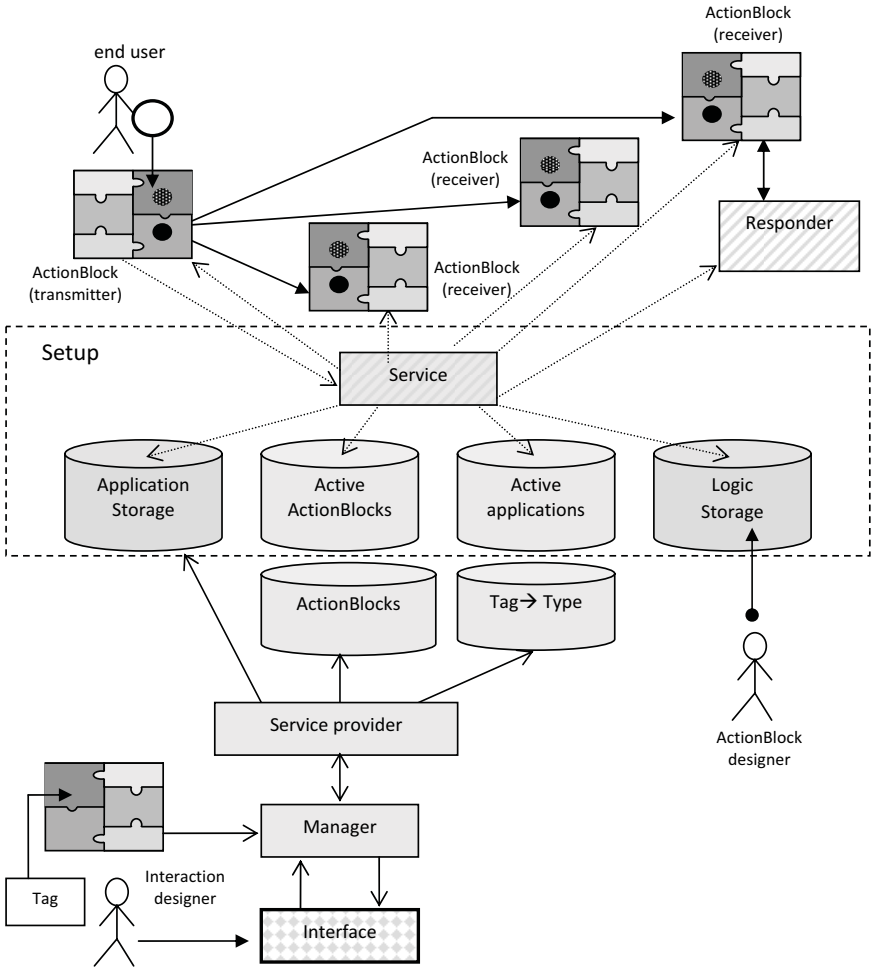


Figure 3 : 10 The whole system

3.3 Discussion

Our goal was to construct an architecture that will support the three roles in an adequate way. We have achieved this by combining several architectures that correspond to different interactions with the system.

The abstraction into three roles is done to be able to focus on three different interaction perspectives or relations to a computer system. In this case especially the ActionBlocks system. The advantage with this approach is that it is easier to grasp the most important issues for each role. This makes it possible to treat all interaction perspectives equal and to give the roles what they want and need.

The end user wants a system were there is a direct coupling between the manipulation of a physical item and the system's response. He also wants the system to be robust. When one ActionBlock ceases to function the system continues to work. A pure peer-to-peer architecture will fit the end user. An advantage with pure peer-to-peer is its robustness because if a peer breaks the network may continue to exist. This together with the fact that the computational part of the ActionBlock really is within the ActionBlock corresponds to the end-user perspective. When an ActionBlock cease to function it is due to visible items, either the tag reader itself or the connection is broken. But there are also some disadvantages with peer-to-peer. It is hard to keep a list over existing peers in the network because all peers hold their own lists and they may not be complete. It may also be a problem to administrate different versions of software components (Minar et al., 1999). The disadvantages concern the configuration and ActionBlocks design. The interaction designer is also interested in having the ActionBlocks doing their own computation because it corresponds to how he handles the configuration, but he is also interested in obtaining a list over all active ActionBlocks. The peer-to-peer architecture has to be extended by a distributed service to meet his needs. The approach opens up for customized solutions were some services can be more or less centralized but the communication between the ActionBlocks is direct.

The distributed service is essential when the interaction designer has a need for trying out and test the functionality of the system while configuring it. A distributed testing service for example can help to trace all communication or can collect testing information from all the peers.

The ActionBlock designer wants the system to be easy to maintain and develop. He also wants to have control over the system. To him it is preferable with a client-server architecture. The ActionBlock designer needs to have total control over the system performance while testing and evaluating the functionality of the developed software. Client-server architecture would supply the ActionBlock designer with this capability. The disadvantages with client-server architecture concern the end-user perspective. When the server breaks, the network ceases to function and there is no personal control over the server. Another disadvantage is that the end user's apprehension of ActionBlock does not correspond to a client-server architecture where the ActionBlocks just are dumb devices that rely on the server.

The testing is an essential part of the ActionBlock designers work and it is in this phase of the development that the ActionBlock designer interacts with or use the system. The initial testing of the software might be done on the ActionBlock designers computer, but the software also has to be tested at the right hardware environment, e.g. at an ActionBlock that ActionBlocks to do their own computation because it corresponds to how he handles the configuration, but he is going to possess the logic. When Minna has developed a piece of new logic and she regards the software as completed she uploads the

new logic to an appropriate ActionBlock. She simulates a receiver or a transmitter on her computer and all communication goes via a server on her computer. In this way Minna controls the communication and may focus on the performance of the new logic. In this situation the architecture is client-server architecture. When the testing is finished she saves the logic at the computer with the distributed service. This line of action provides the ActionBlock designer with some important advantages of a client-server architecture.

By making the architecture adaptable to different situations of use it is possible to support all three roles. You can also say that the architecture changes focus depending on the use. When the end user interacts with the system it acts as a peer-to-peer application. When the interaction designer uses the system it acts as a peer-to-peer application with a distributed service and when the ActionBlock designer interacts with the system it can be regarded as client-server.

The user roles are founded in different kinds of requirements. It is harder to catch and implement cognitive requirements because they are not accountable, but we regard it essential to capture all types of requirements to make the system suit the user roles which makes it possible for the users to disregard some aspects and focus on other more vital tasks. This makes it easier for the users to survey the system. You may question if this approach is applicable in other situations. We advocate that a consistent development process and a flexible system that develops all the time require an approach that takes care of all participants' requirements both task related and softer ones. This approach might be useful in other similar settings were the system continuously evolve, like in tailoring and end-user development.

A disadvantage with the separation of different perspectives is that the architectures may stay separate in the implementation. This approach may lead to separate systems. That is not the intention. There also might be a gap between the user roles, if one is not aware of the fact that the roles can slip into each other. This may lead to a less adaptable system. Another disadvantage is that the system might be more complex if the system designer is not aware of the risk and has not as a goal to make the architecture as simple as possible without giving up the concept.

The scenario sketches a rather limited picture of how flexible the system might be. Let us consider another scenario: When Jan arrives to the exhibition hall he can choose an item that appeal to him from a basket, a nice stone, a ball etc. Then when he sees something he especially likes in the exhibition, an image, a film, a noise, a piece of music he can associate the chosen item to that object. A representation of several objects is kept in the exhibitioner's computer and when Jan finds something he wants to store, an association may be done by putting the item at a tag reader and by choosing the representation in a simple computer interface. Jan associate his stone to his favourite sound by assistance of a Graphical User Interface (GUI). When Jan puts his tagged stone at the tag

reader a message is sent to the computer that gets knowledge of the tag id. A picture on the tag reader representation indicates when a tag is put on the physical tag reader. The available files are shown in the explorer and Jan click on the file he wants to associate to his stone and drag it to the tag reader representation in the GUI. The stone is now associated to the sound of laughing children. If Jan is not satisfied by the result he may do the procedure again.

When the end user on his own associates tags to digital media he in a way alters the configuration of the system. He makes a connection between the tag and a file just like the interaction designer associates different ActionBlocks to each other. The border between the end user's task and the interaction designer's task is not so evident any more. The distinction between the user and the interaction designer get blurred.

The interaction designer already assigns logic to the ActionBlocks. Let us picture that the interaction designer might program the logic by himself in the same interface as he associates the ActionBlocks in. For example by using programming by demonstration and visual before-after rules it is possible for non-programmer to program computers (Canfield Smith *et al.*, 2000). Applying such a method even blur the distinction between the interaction designer and the ActionBlock designer. We have to remember that the roles may be contained in one person. The roles may slide into each other.

3.4 Conclusion

Construction of flexible systems is an effort to extend the usability of the systems. In a flexible system the end user tends to perform tasks that earlier was dedicated for professionals. The development of the software becomes a continuous process that does not end when the end user take the system in use. This is especially apparent in the system described in this article where the end user, the interaction designer and the ActionBlock designer continuously cooperate to evolve the system. The three roles perceive and use the system in different ways. The user designs in use of ActionBlock and he sees the physical part of the system, and he regards it as a working tool. The interaction designer designs the interaction between the ActionBlocks and the user. He sees the system as a building kit that can be used to build tools for the user. The ActionBlock designer designs ActionBlocks and he thinks of the system as software components that can be assembled to make various actions. The differences in how the roles use and perceive the system makes them have different perspectives and requirements on the system. The usability of the system is dependent of how well the systems support the different situations of use and thereby the requirements.

In this chapter we have shown that by explicitly discerning the three roles and analyze their use, interaction and perspectives of the system it is possible for us to focus on the roles one by one and support the different roles by different architectures. This approach means that we equalize the three roles' importance

for the continuous evolution of the system. The different architectures can then be combined into an architecture that satisfy all the roles and that adapt to the different kinds of use. The architecture acts as a foundation for the continuous development of the system.

The prototype implements the concept described in this article and we have thereby shown that it is possible to cater for different user roles even if they go beyond the limit for the user interface and into the underlying architectures. But there is still research left to do to explore how to reflect user roles in architectures in other settings.

Chapter Four

Paper III

Chapter Four

Can End-Users Manage System Infrastructure?

User-Adaptable Inter-Application Communication

WSEAS Transactions on Computers, December 2004

Jeanette Eriksson

The study presented in this article was carried out in cooperation with a telecommunication operator in Sweden. Since this line of business is characterized by fast change the company's information systems must also change rapidly. In such a fast-changing world flexibility is needed in software to prevent the software becoming obsolete. One way to provide this kind of flexibility is End-User Development (EUD). EUD "can be a strategic solution to bridge the productivity gap by allowing end users to directly implement some additional features important to accomplish their tasks" (Paterno *et al.*, 2002). One way of conducting EUD is end-user tailoring. End-user tailoring enables the end user to modify the software while it is being used as opposed to modifying it during the development process (Henderson and Kyng, 1991). Tailoring is also a way to reduce the efforts keeping the system up to date through further development. EUD and tailoring are used in stand-alone applications or in the case of distributed systems with predefined homogenous data sources. Example of such tailorable applications can be found in (Mørch and Mehandjiev, 2000, Stiemerling et al., 1998).

Our industrial partner has some tailorable business systems that communicate with other systems in the infrastructure. As flexible connections are needed in an infrastructure for flexible systems it is a natural progress to provide the end user with the possibility to tailor the communication paths and data flow between different systems; a possibility to manage system infrastructure; whenever necessary in the system tailoring process.

Various functionality needed to manage the infrastructure exists in tools for system integration (EAI; Enterprise Application Integration (Lee et al., 2003)), For example Microsoft BizTalk Server[1], Microsoft Host Integration Server[2], Sun ONE Integration Server[3] and WebMethods Integration Server[4], network management (monitoring the infrastructure) (Subramanian, 1999), component

[1] <www.microsoft.com/biztalk/default.mspx>, accessed January 27, 2008
[2] www.microsoft.com/hiserver/default.mspx, accessed January 27, 2008
[3] <sun.com/software/products/integration_srvr_eai/home_int_eai.html>, accessed January 27, 2008
[4] http://www.webmethods.com/Products/B2B, accessed January 27, 2008

management (if you choose to regard the different systems as components; how the data is structured) (Szyperski, 2002) and report generation (assembling data) (Chan, 1998). Except for report generation; that sometimes supports end users but often need support from developers to fit new data sources; these tools are exclusively designed for system experts not for end users.

Accordingly (I) tools for managing infrastructures are designed for system experts and (II) tailoring is used in stand-alone applications or in the case of distributed systems together with predefined homogeneous data sources. In our case an end-user tool for tailoring communications between different distributed heterogeneous data sources is needed. We therefore performed a design study to explore the possibilities of providing the end users with such a tool using existing standard techniques available at the company. The result was a prototype called EDIT (Event Definer for Infrastructure Tailorability). This paper describes the structure of EDIT and analyses the lessons learned.

The structure of the chapter is visualized in Figure 4 : 1. In the following section we present the problem. In the section thereafter the structure of the prototype is described from the user's, tailor's and developer's perspective. We then discuss the findings and alternative uses of EDIT. Finally we conclude that it is possible to enable the end user to tailor communication between different heterogeneous data sources in a large infrastructure. By the construction of the prototype we show that it is possible to provide a simple solution that take advantage of existing standard technology and that facilitates both use and tailoring and also makes it easy to extend the tailoring capabilities to ensure the system evolves along with the business tasks.

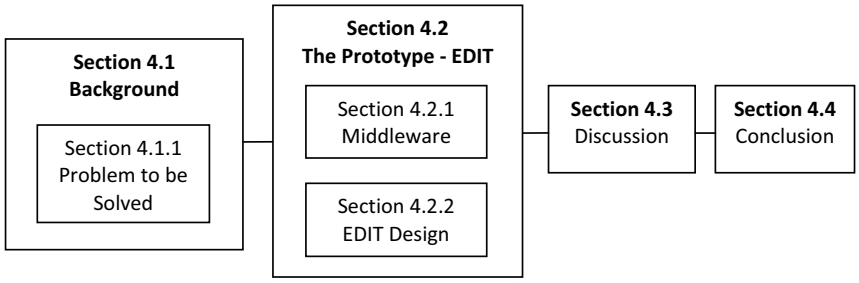


Figure 4 : 1 Overview of Chapter Four

4.1 Background

In the telecommunication business the business environment changes very fast and competition is hard. Telecommunication operators compete by among other things introducing new types of services to the customers and by improving business systems that take care of the business side of the services. But because changes are very fast, it takes a lot of effort to keep business systems up-to-date. To come to terms with this problem, our industrial partner has invested in

making some systems tailorable by the end user (Dittrich and Lindeberg, 2002, Dittrich et al., 2006).

In a previous project (Dittrich and Lindeberg, 2002), a system handling contracts for payments was made adaptable; however, the system communicates with several other systems that are not adaptable, e.g. the system managing payment data is not tailorable. When creating new contracts or types of payments, adaptability is restricted by the fact that the system handling the data can only handle specific data sets. This limits the flexibility and reveals the need to tailor the communication paths and data flow between different systems as well.

In our study the subset of the infrastructure that deals with payments served as an example of system infrastructure. From now on we will refer to this subset as 'the payment system'.

4.1.1 Problem to be Solved

The payment system is used for computing certain payments[5] determined by what contracts are valid; these payments are triggered by specific events. Each event that triggers a payment has a set of parameters (data set). Each event type has a set of attributes associated with it that limits what conditions a contract belonging to it can have. This means that a payment is triggered by an event, and all contracts belong to one of the two existing event types. The data describing the triggering events is periodically imported from another system once a month. The actual payments are made by sending a file to the administrative system.

To make new types of payments, new types of contracts must be implemented; this is done by the end users.

We have just noted that there are only two types of events today, but this is not entirely true. Several payments based on events cannot be handled automatically by the regular payment system. We call this kind of payment 'extra payments'. Extra payments are handled and computed manually but run through the payment system in order to send a file to the administrative system. Extra payments are also made once a month, like the regular payments.

The manual procedure to compute extra payments has until recently worked well but took a lot of time. But the competitive telecom business is forcing the company to come up on a continuous basis with new services; ultimately, other types of extra payments are needed. These extra payments are based on new types of events, which means that new types of data sets are needed. This revealed the need for a tool to define and handle the new events.

To make the event definer/handler as flexible as possible, it must be able to assemble data from different kinds of systems. Experience suggests that it is

[5] To protect the business interests of our industrial partner we can only give an abstract description of the system.

impossible to anticipate how future extra payments will look and which details are needed. As a result, the event definer/handler must be able to communicate with any system in the infrastructure. What is needed is a tool for inter-application communication which can be adapted by the user. It is also essential that the tool allows expansion of the tailoring capabilities so that new data sources can be added.

The main research question in this article is how to structure a tool that makes it possible for end users to manage a large infrastructure and at the same time facilitate both use, tailoring and further development of the tailoring capabilities.

4.2 The Prototype - EDIT

To explore how to solve the problem stated above we developed a prototype called EDIT (Event Definer for Infrastructure Tailorability). EDIT is designed to highlight such issues as how to make it possible for end users to:

- assemble data from different sources
- set up rules and algorithms that will be performed during computation
- map data sets to receiving sources.

It is important that the design is kept as simple as possible to make EDIT as easy as possible to understand and survey. If the design is kept simple, it will also be easier to visualize new ranges of uses and to extend tailoring capabilities. The principle of simplicity also includes use of existing, well known standard techniques.

The users tried out the prototype in a setting close to the real-world environment with real-world data, while the users 'talked aloud' (Robson, 2002) to express their apprehension, perception and understanding of the prototype. One developer working with maintenance of the regular system also evaluated the prototype and gave her opinions on it. Advantages and drawbacks concerning use, tailoring and expansion of the tailoring capabilities were discussed. All employees concerned with the payments participated in the evaluation process. The reactions of the prototype were positive.

The prototype has also been successfully tested to fit in a technical sense into the infrastructure of the company.

4.2.1 Middleware

The company has recently bought a platform which supports integration. A little simplified, the platform consists of integration servers, brokers and workflow servers.

The integration server is the platform's central component at run time and it connects internal and external resources to the platform. The integration server works as the entrance for the systems and applications to be integrated. The

services running on the integration server consist of integration logic that retrieves data from one resource and delivers it to another. The idea is that by subscribing to a service the subscribers shall obtain the information needed whenever new information appears. We had a somewhat different intention when using the platform: We wanted to collect the information when we need it. Instead of passively waiting for the data and then sort out a subset of interesting data, EDIT actively gathers the information when needed. The platform provides EDIT with information about how to get in touch with desired resources and what data is accessible at these resources.

The integration server could not inform us about which resources are available in the infrastructure; we thus had to create a service that makes it possible for developers to publish information about their system. To do so, the developers set up a database view containing data that can be shared with tailorable systems such as EDIT, and then with the aid of a service on the integration server, they publish how to connect to the system and what view to use. We call this service 'publishResource'. The service produces an XML file containing connection data for all published resources. When EDIT wants to know what resources are available, the XML file is fetched from the integration server by means of a service called 'getAvailableResources'. The third service, called 'getMetaData', provides EDIT with meta data from selected resources, e.g. what fields (attributes) can be accessed in a specific database, and what types the fields are.

An advantage of using the integration server on the platform rather than any other server is that the integration server already contains services that can be combined and extended to fit specific requirements as opposed to making the services from scratch. Furthermore, the platform provides a graphical interface to the services, which makes it fairly easy for developers to tailor services.

4.2.2 EDIT Design

EDIT is divided into two parts. In one part the end user can tailor communication and data interchange between systems, i.e. the end user defines the event types. The other part handles the execution of extra payments or events. We can call the parts Event Definer (handles tailoring) and Event Handler (handles use) respectively. The Event Handler is used once a month to run the different extra payments, while Event Definer is only used when someone comes up with a new type of extra payment.

Inevitable there will be a point in time when the tailoring capabilities in the system are not enough to perform a new task. To make durable tailorable systems it is essential that the further development of the tailoring capabilities is made easy.

Figure 4 : 2 shows a simplified picture of EDIT and its connections to other systems. When Event Definer starts, the XML file ('AvailableDB.xml') containing all the published systems is fetched from the integration server. It is

possible for the end user to select some of the published resources to avoid cluttering the interface with uninteresting information ('Publications'). The collection can be altered at any time. This selection is done separately and is only altered when new resources are needed to define a new extra payment.

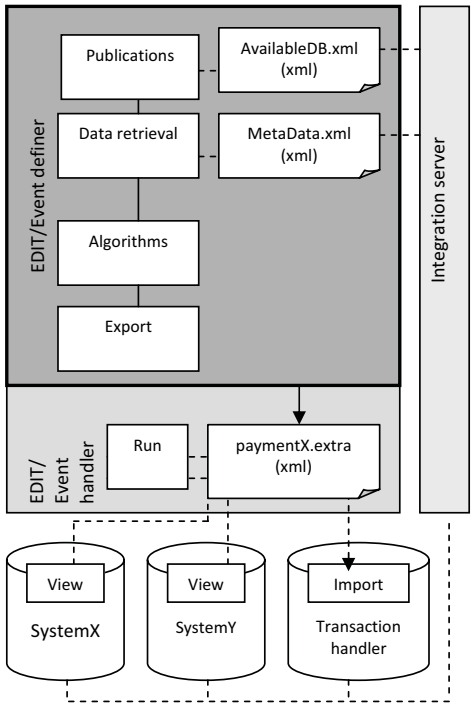


Figure 4 : 2 EDIT

Tailoring

The graphical tailoring interface of the Event Definer has been constructed using different steps. These guide the end user through the process, but can also be used in arbitrary order as the end user chooses. Some steps must be completed once, before the end user can alternate between the steps. The steps are revealed one by one and united consecutively. There are seven steps in the graphical interface:

Step 1: Naming the extra payment

Step 2: Choosing what databases to connect to

Step 3: Choosing what fields to use from the selected databases

Step 4: Setting up criteria for what data to collect from the databases

Step 5: Shows the specified criteria from Step 4 as SQL queries

Step 6: Setting up algorithms of what to do with the collected data

Step 7: Mapping the input table structure with the output table structure

Step 1 simply means that the end user names the extra payment to distinguish between different extra payments. As already noted, when EDIT starts the XML file 'AvailableDB.xml' is fetched from the integration server. In this way, EDIT is always up-to-date with whichever systems are available. The end user has already sorted out which of the published systems that are normally of interest and only these are shown in Step 2.

In Step 2, the end user chooses which databases to use for the present extra payment.

The next step (Step 3) is to choose what fields to use from the different databases; to make this possible for the end user, EDIT has to know what fields there are in the 'views' in the different databases. To determine the structure of the 'views', EDIT calls the service 'getMetaData' at the integration server; the service then calls the system in question and discovers the structure of the view. The information is then collected in an XML file called MetaData*SystemName*.xml (MetaData.xml). This procedure takes place for all the selected databases in Step 2. The structures of the views are then shown in the graphical interface, and the end user can make his or her choices. Even if the steps can to a great extent be carried out in an arbitrary order, for obvious reasons Steps 2 and 3 must be performed once before Step 4 etc. can be performed.

Step 4 is the most advanced step and requires quite a lot of knowledge of the business task and data required. In Step 4, the end user must choose what field is to be the base for the period selection, e.g. there is likely to be more than one field containing dates in the whole collection. The end user must select which date field to compare with when executing an extra payment for a specific month. In Step 4, the end user must also specify how the different 'views' are related to each other, how they are linked together, e.g. SystemX is linked to SystemY by saying that fieldX in SystemX must be equal to fieldY in SystemY. These two tasks in Step 4 are mandatory but setting up selection criteria for what data to collect is optional. The graphical interface makes it possible for the end user to drag and drop the field names in slots and set up conditions for them, e.g. fieldX must be equal to 'HI00' etc.

Step 5 is somewhat similar to Step 4 as it shows the SQL queries representing the criteria the end user has set up for data retrieval. This step exists because it should be possible for the end user to set up more complicated (and unusual) conditions for data retrieval than those which the graphical interface can accommodate.

Step 6 makes it possible for the end user to specify what algorithms are to be used on the collected data. Up to this step, the different displayed systems have been kept separate but now the structure from the different 'views' is assembled to make it possible to combine fields from different databases in the same algorithm. We have chosen to focus on how the end users would prefer to write

the algorithms rather than implementing the algorithms. Therefore how the algorithms are performed will not be discussed here.

Step 7 enables the end user to map the assembled and computed data to a receiving system. This is done by showing the structure of the assembled data views to the left and the structure of the receiving table to the right in the graphical interface; it is then possible for the end user to drag different fields from the left table to the right and in this way map the different sources together (Figure 4 : 3).

Step 7: Export to Database

Event table

Datafield	Type
Code_DB1	VARCHAR
Month_DB1	INTEGER
Date_DB1	DATE
SUM_DB2	INTEGER
Name_DB3	VARCHAR

Export table

Import	Datafield	Type
	ID	VARCHAR
Code_DB1	CodeNr	VARCHAR
Month_DB1	CallMonth	INTEGER
Date_DB1	Date	DATE
SUM_DB2	SUM	INTEGER
	SubSum	INTEGER
Name_DB3	Name	VARCHAR
	PayDate	DATE

Figure 4 : 3 Step 7

All these choices, criteria, algorithms, mapping etc. are finally brought together and arranged into an XML file named according to the name of the payment stated in Step 1. An event type is created. The XML file has the same structure as the interface of Event Definer: specification of data retrieval, algorithms and export.

Use

The XML file produced by tailoring is then used whenever the end user decides to execute the extra payment. The execution of extra payments is carried out by Event Handler in EDIT. The XML file specifies which systems to connect to and what data to collect. The business decision stating the integration server only to house services facilitating integration and not regular data transfer result in that the XML file must specify how to connect to the different systems directly. When an extra payment is executed, EDIT contacts the chosen systems one by one and data is collected from each of these. It is here that the relationship or link between the systems the end user specified in Step 4 comes into use because the collected data from one system acts as input for the data to be collected in the following system, e.g. if there is a link between SystemX and SystemY that says that fieldX in SystemX must equal fieldY in SystemY, and if the data collected from fieldX has the values ‘H001’, ‘H002’ and ‘K666’, only records containing those values in fieldY will be collected from SystemY, and

so on. The collected data is stored in a temporary database and assembled in one table. When the data is collected and assembled in a single table, the events are created, and the actual payment procedure can take place. The algorithms are then applied to the data and the result is displayed to make it possible for the end user to check and correct the result where necessary. By clicking on a button, the end user eventually exports the result to the system handling the payment in accordance with the mapping specification in the XML file. The data in the temporary database is erased when the execution of the extra payment is finished.

Expansion of Tailoring Capabilities

There will always come a time when the end user wants to retrieve data that is not published in an available view. Collaboration between the end user and the developer in question must in such cases work adequately because the developer must update the view, create a new view or publish a new resource to meet the end-user requirements.

We have described how the middleware provides EDIT with information about the surrounding systems; EDIT is not, however, dependent on the middleware because XML files, provided by the integration server, could be produced manually by the developers. The integration platform makes the integration between systems easier because EDIT can easily obtain adequate information about the surrounding systems. One of the great advantages of XML is that both computers and humans can read it easily if it is kept in a simple form. We have tried out this course of action too and even though the middleware greatly facilitates the administration of the system and keeps it up-to-date in relation to surrounding systems, EDIT work smoothly without middleware.

4.3 Discussion

To be able to tailor something the end user must have a general view of what can be achieved. This is one of the reasons why we have chosen a simple design for EDIT. We believe the overview benefits from an understanding of the structure of the application. A simple design facilitates understanding of the structure and is it easy to understand then it is easy to use. This has been confirmed by the user tests.

We have in EDIT focused on a design that makes the tool easy to interact with for both users, tailors and developers which has led to separations of concerns that was reflected in the structure of the application. If we look at the structure of EDIT we can se that providing support for both use, tailoring and further development of the tailoring capabilities is done by distinctly dedicate different building blocks in the system for a specific interaction and by keeping the parts clearly separated. The building blocks are encapsulated in a general shell that is not affected if the building blocks are changed. For example when developing the tailoring capabilities further the XML-file ‘MetaData.xml’ is produced as a result of the developer publishing a new view. When doing tailoring

'MetaData.xml acts as input, but the activity produces a new XML-file 'paymentX.extra' and this file then acts as input for using the system. In other words the different activities only produce output to one part of the system (Figure 4 : 4).

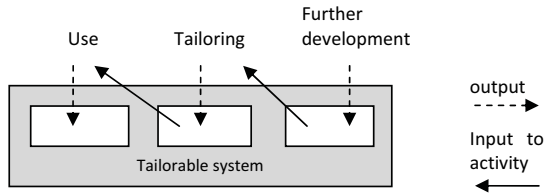


Figure 4 : 4 Division into three parts

This division into three parts can be observed in other tailorable applications too (Chapter Two). The phenomenon occur when it is required to facilitate not only use and tailoring but also further development of the tailoring capabilities. If the division into three parts can be regarded as a guideline of how to structure such a system will be explored in the future.

The division of EDIT into three parts makes it uncomplicated to use EDIT in settings other than those aimed at computing extra payments. As long as the developer directly or indirectly; by assistance of a integration server; provides EDIT with information of how to connect to the data sources any setting involving collecting and manipulating data from different sources is supported.

4.4 Conclusion

The preconditions that have to be fulfilled to be able to answer 'yes' to the question "Can users manage system infrastructure?" are that the end users have solid knowledge of the nature of the task and what data is required to perform the latter, but also a working collaboration between developers and end users when the possibilities for end users to manage the infrastructure must be extended.

We have shown in the example of EDIT that it is possible to enable the end user to manage and tailor communication between different heterogeneous data sources in a large infrastructure. It is even possible to do this in a simple way with small means and at the same time facilitate not only use and tailoring but also further development of the tailoring capabilities.

Chapter Five

Paper IV

Chapter Five

Combining Tailoring and Evolutionary Software Development for Rapidly Changing Business Systems

What is required to make it work?

Journal for Organizational and End-User Computing 19(2) 2007

Jeanette Eriksson, Yvonne Dittrich

End-user development (EUD) is one way to provide a flexibility that allows companies to compete in rapidly changing business environments. Telecommunication provision is one such example of a rapidly changing business area. Telecommunication providers compete by, among other things, providing their customers with new types of services, and as the business changes, the business systems supporting it must also change. One way of conducting EUD is end-user tailoring. End-user tailoring is an activity allowing end-users to modify the software while it is already in use, as opposed to modifying it during the development process (Henderson and Kyng, 1991). End-user tailoring ranges from setting the values of parameters to adding code to the software. Since evolution of software is inevitable (Lehman, 1980) and since tailoring is recognized as a way of reducing the efforts when keeping the system up to date through further development (Mørch, 2002), tailoring could be an alternative to increase the sustainability of software in a rapidly changing business environment.

Tailoring research so far has focused on flexible stand-alone systems. In earlier projects, we too focused on the design of flexible and end-user tailorable applications (Chapter Two). However, interaction with other systems turned out to be a bottleneck, since business systems in telecommunication are part of an IT-infrastructure consisting of heterogeneous data sources. Other research also indicates that software and IT-infrastructures pose new challenges for software engineering (Bleek, 2004). Normally, the data exchange between different systems is the realm of the software developers, but in this article we use the evaluation of a prototype to answer the question: What is necessary to allow *end-users* to tailor the interaction between flexible applications in an evolving IT-infrastructure? Our results support the claim that end-users can even tailor the interaction between business applications. The analysis of a user evaluation of a case-based prototype results in a number of issues to be addressed regarding the technical design, the know-how demanded of the users, and the organizational setting, particularly the cooperation between users and

developers. These issues both confirm and extend existing research on end-user development and tailoring.

The structure of the chapter is visualized in Figure 5 : 1. We start by briefly describing the relevant work practices and business systems of our industrial partner. We then present how our research relates to others' work. In the following section, we describe our research approach in detail and the design of the prototype is presented to provide a basis for the evaluations and discussions. Thereafter, we present the outcome of the evaluation, which points out three different categories of issues that are important when providing end-users with the possibility to manage interactions between applications in an evolving IT-infrastructure. The discussion relates these results to the state of the art.

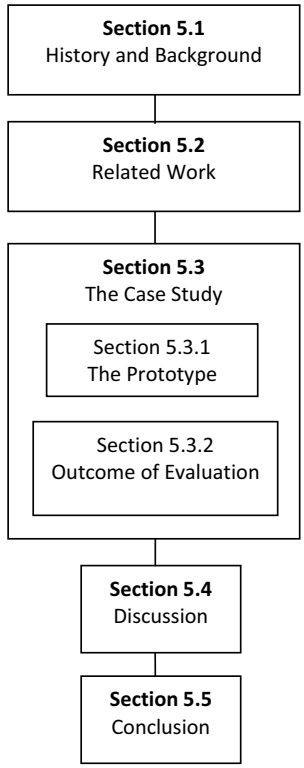


Figure 5 : 1 Overview of Chapter Five

5.1 History and Background

The research reported here is part of a long-term cooperation between the university and a major Swedish telecommunication provider, exploring the applicability of end-user tailoring in industrial contexts (Dittrich and Lindeberg,

2002). The subject of the prototype is part of the telecommunication provider's back office support infrastructure for administering a set of contracts and computing payments according to these contracts. To compute payments, the system must be supplied with data from other parts of the IT-infrastructure. When creating new contract types based on different data, flexibility is constrained by the hard-coded interface to other systems. As a work-around, ASCII files can be created providing the necessary data sets – or events – to compute the payments. The data for these extra payments is handled and computed manually. To compute the data for an extra payment, members of the administrative department first run one or more SQL queries against the data warehouse. The result is stored in ASCII files. Next, the user copies the data from the ASCII files and pastes it into a prepared spreadsheet. When the user has thus accumulated the data, the user works through the spreadsheet in order to remove irregularities. The contents of the sheet are eventually converted again to an ASCII file that is imported into the payment management system. The manual procedure to compute the data for the extra payments has worked well until recently, although it is time consuming. The competitiveness of the telecom business is however continually forcing the company to come up with new services; more and more types of extra payments will be needed. This situation necessitates a tool to define and handle the new data sets or events. To make such event tool as flexible as possible, it must allow the collection and assembly of data from different kinds of systems. Experience suggests that it is impossible to anticipate the structure of future extra payments or which details will be needed. As a result, the tool must be able to communicate with any system in the IT-infrastructure. It is also essential that the tool allow for expansion of the tailoring capabilities, meaning that new data sources can be added. The addition of a new source should be as seamless as possible. Since different system owners and developers are responsible for these systems, it is their responsibility to make new data sources available. Such changes are part of the maintenance of the other systems, and here the limits of end-user tailoring are reached.

5.2 Related Work

The research on end-user tailoring addresses mainly the design of tailorable applications, tailoring as a work practice, and cooperation between users and tailors. Examples of research on the design of tailorable systems are (Mørch, 1997, Stiemerling, 2000, Stiemerling et al., 1998) and Chapter Two and Three. Of these, only two address tailoring in the context of distributed systems. In Chapter Three a prototype that dynamically connects different physical devices (video cameras, monitors, tag readers, etc.) is presented. The tool can be regarded as tailoring the interaction between different intelligent devices. Stiemerling (2000) and his colleague (Stiemerling et al., 1998) show how to build a search tool by using customized Java Beans. The users customize search and visualization criteria. The tailorable search tool is used within a distributed

environment provided by a groupware system. Neither of the distributed tailoring approaches is evaluated by users to explore beyond technical issues of how end-users can manage interaction between applications.

Several researchers have studied how tailoring activities are carried out in work practice, for example (Gantt and Nardi, 1992, Stevens et al., 2006, Trigg and Bødker, 1994). In a study involving tailoring spreadsheets, Nardi and Miller (Nardi and Miller, 1991) identify collaboration between three kinds of users of CAD (Computer Aided Design) systems (1) users who do not program (2) users who acquired the skill to program small macros, and (3) local developers: users having a more or less formalized responsibility for supporting other users and maintaining the macro selection of a group or department.

Carter and Henderson (Carter and Henderson, 1999) invented the expression *tailoring culture* to express the need for organizational support for tailoring. Kahler (Kahler, 2001) also points out that, in order to make tailoring successful, an organizational culture must evolve that supports the development and sharing of tailoring knowledge. Kahler also emphasizes three often coexisting levels of tailoring culture, identified and addressed by different researchers. First there is a level with equal users; people help each other to tailor the software (Gantt and Nardi, 1992) or there is a network of whom to ask when encountering trouble when tailoring the software (Trigg and Bødker, 1994). Second, there is a level with different competencies (Gantt and Nardi, 1992). The third level is a level of organizational embedment of tailoring efforts and official recognition of tailoring activities (MacLean *et al.*, 1990). We will return to this classification in the discussion of our results, as our findings propose the consideration of a fourth level of tailoring culture when implementing and deploying tailoring possibilities in an IT-infrastructure environment.

5.3 The Case Study

Our research approach can be described as a single case study (Yin, 2003) following a design research paradigm (Nunamaker *et al.*, 1991). The question "What is necessary to allow end-users to tailor the interaction between flexible applications in an evolving IT-infrastructure?", addresses the design and deployment of a previously inexistent functionality. In design research, the design and development of a (prototypical) information system can be used both to answer technical questions and as a probe to explore requirements posed by the deployment of the technical possibilities. Hevner et al. (2004) especially emphasize the need for combining design research and behavioural science. The technical design of the prototype is discussed in Chapter Four.

The practical work was conducted during a period of slightly more than one and a half years. Prior research indicates that the collection of data to process the so-called extra payments was a bottleneck both for the users' work as well as for deploying the flexibility implemented in the existing systems. During the initial field studies focusing on the work practice of the business department, we

visited our industrial partner once or twice a week to observe and interview both users and developers. These field studies informed the development of the overall research question and also the design of the prototype.

In the beginning of the design phase, workshops were arranged involving researchers, users and developers. When designing the prototype, one of the researchers was stationed at the company two or three days a week to ensure that the prototype conformed to existing company systems. Field notes were taken, and meetings and interviews were audio taped, during all phases of the case study.

The prototype was evaluated by all three employees involved in the collection of data and computation of the extra payments and by one developer involved in the maintenance of the payment system. These evaluations were video taped. The analysis in the section of this article entitled "Outcome of Evaluation" is mainly based on the latter tapes, but uses the other field material as a background. For secrecy reasons, videotaping is not allowed on the telecommunication provider's premises. We therefore installed the system on a stand-alone computer outside the actual work place. To allow the users to evaluate the prototype realistically, we reconstructed part of the IT-infrastructure in a local environment and populated it with business data, developing our prototype into a case-based prototype (Blomberg *et al.*, 1996). The users were given two tasks. One task was to construct the collection and assembly of data for an extra payment that they implemented regularly (in the manual fashion described above) as part of their normal work. For the second task they had to construct a totally new but realistic payment. The users were asked to *talk-aloud* while performing the task. This method is common when evaluating software in a use context (Ericsson and Simon, 1993, Robson, 2002). The researcher performing the evaluation observed and asked exploratory and open-ended questions to provoke reactions that differed from our expectations. The developer who worked with maintenance of the regular system evaluated the prototype in a workshop, and discussed advantages and drawbacks concerning use, tailoring, and expansion of the tailoring capabilities.

We analyzed the data in a manner that was inspired by grounded theory. A coding scheme was developed with its starting point in the transcripts of the evaluation sessions. The researchers coded the interviews independently from one other and then compared their results. The resulting categories were finally merged into three core categories, that is, design issues, user knowledge, and organizational and cooperative issues. The categorization can be found in the evaluation section.

5.3.1 The Prototype

The prototype is divided into two parts, the Event Definer and the Event Handler (Figure 5 : 2). By using the Event Definer, the end-user can tailor communication and data interchange between systems, that is, the end-user defines the event types for the computation of the above-described extra payments. It allows the user to: define the assembly of data from different sources (Figure 5 : 2a), set up rules for aggregation and algorithms that will be performed on the data when aggregating the data (Figure 5 : 2b) and define how to map data sets to the format required by the receiving application. (Figure 5 : 2c). The Event Definer needs to be used only when defining new types of extra payments. The Event Handler handles the execution of extra payments or events and is to be used once a month to run the different extra payments.

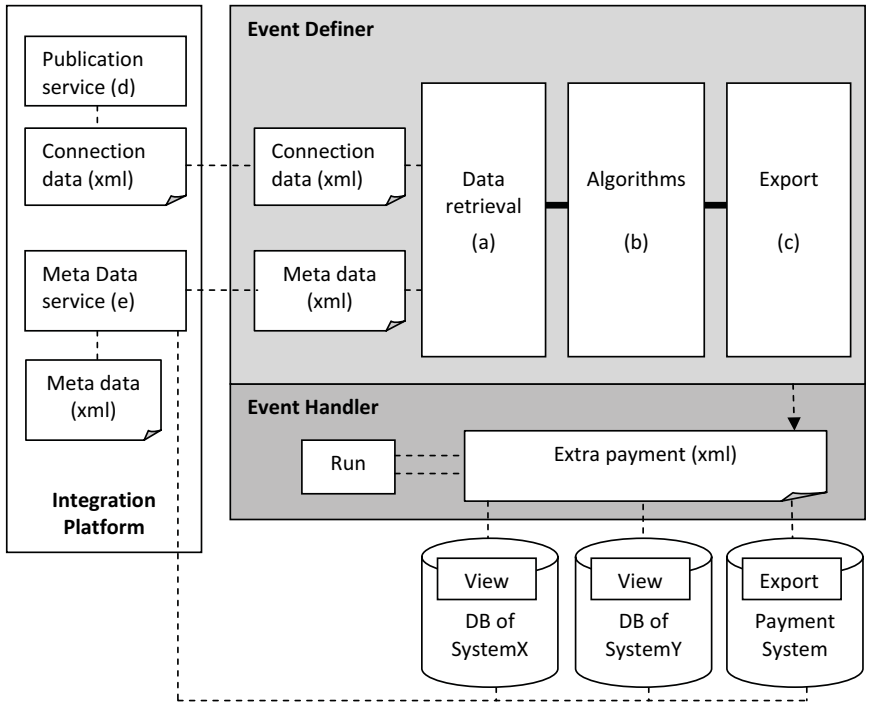


Figure 5 : 2 The connection between the prototype and the surrounding systems

Various solutions exist that provide the functionality needed to manage the connections between applications. These are found in tools for system integration that connect systems, in network management for monitoring the IT-infrastructure, in component management (if you choose to regard the different systems as components) and in report generation for assembling data. These tools are designed exclusively for system experts, not for end-users. A possible

exception is report generation, which sometimes supports end-users but often needs support from developers to adapt it to fit new data sources. We found that none of these approaches was suitable for fulfilling the requirements for a tool for interapplication communication that can be adapted by users. Neither were the approaches suitable for the purpose of exploring what is necessary to allow end-users to tailor the interaction between flexible applications in an evolving IT-infrastructure. For our prototype we used an existing platform that supports integration between the telecommunication provider's back office applications. The integration platform makes it possible to publish events that other applications can subscribe to. We had a somewhat different intention when using the platform. We wanted to collect the information when needed, and we used the platform to provide the prototype with information about how to get in touch with desired resources and what data were accessible at these resources. We created a service (Figure 5 : 2d) on the integration platform that allowed the developers of the different systems to publish information about available data and showed how to connect to the respective database. To do so, the developers must set up a database *view* containing data that could be accessible to other systems (such as the prototype). The service produced an XML file containing connection data for all published data sources. When the Event Definer starts, the XML file is fetched from the integration platform. Yet another service (Figure 5 : 2e) provided the prototype with metadata from the data sources, for example, which fields (attributes) could be accessed in a specific database, and the types of the fields.

Tailoring

The graphical tailoring interface of the Event Definer was constructed to consist of seven different steps. These steps are intended to guide the user through the process, but could also be used in an arbitrary order as the end-user chooses.

Step 1: Naming the extra payment.

Step 2: Choosing which databases to connect to.

Step 3: Choosing which fields to use from the selected databases.

Step 4: Setting up criteria for what data to collect from the different databases, that is, by drag and drop, the end-user chooses which field should be used and the end-user can also specify how the different views should be linked together, for example, fieldX in SystemX must be equal to fieldY in SystemY.

Step 5: Showing the specified criteria from Step 4 as SQL queries, that is, here the user can edit the SQL queries to set up more complicated (and unusual) conditions for data retrieval than can be accommodated by the graphical interface.

Step 6: Setting up algorithms for what to do with the collected data. (partially implemented.)

Step 7: Mapping the input table structure to the output table structure, that is, the end-user can map the assembled and computed data to a receiving system by dragging the fields from the assembled data table and dropping them in a table representing the receiving database.

All these choices, criteria, algorithms, mapping and so forth, were finally brought together and arranged into an XML file (extra payment in Figure 5 : 2).

Use

The XML files produced by the Event Definer are then used whenever the end-user decides to execute the extra payment. The Event Handler contacts the chosen systems one by one and collects the data specified in the XML file. When the data is collected and assembled in a single table it is displayed to the user to allow for checking and correcting the result where necessary. By clicking on a button, it is possible for the end-user to export the result to the system handling the payment data, in accordance with the mapping specification (Step 7).

Expansion of Tailoring Capabilities

There will inevitably be situations where end-users wish to define extra payments based on data that is currently unavailable. If the data and metadata are unavailable, the end-users are unable to perform new tasks. They have neither the authority nor the ability to alter or add views in surrounding systems. In this case the surrounding systems, as well as the tailorable system, have to evolve to meet the additional requirements from the end-users. The developer responsible for the respective system must then (a) alter the system by creating a new view or changing an existing view, so that it contains the required data, and (b) make the changes available through the integration platform. To support the latter, the publication of a new source was supported by a web interface where the developer (also system owner) could fill in the necessary data.

5.3.2 Outcome of Evaluation

The evaluation presented here focuses on issues beyond the technical design and the appearance of the graphical interface of this specific application. It addresses overall design issues for this kind of application, the end-user knowledge necessary to handle such complex tailoring tasks, and organizational issues to deploy such systems in a sustainable way. We have also evaluated the prototype against functional requirements, but the results are not reported here. Individual opinions held by only one or two of the subjects are disregarded in the following presentation.

Design Issues

In terms of technical support we focused on the different interfaces provided by the prototype: the tailoring interface, the deployment interface and the development interface.

The Tailoring Interface

Functionality for Controlling and Testing

All users appreciated the freedom to alternate between the seven steps. They found that the steps provided not only guidance and an overview but also the freedom to alter something performed in previous steps, without losing the overall view. To be able to overview all choices and trace them backwards was one way of providing control. But there was also a need for error control and limitation. The users, especially the beginners, wanted some kind of guidance in order to feel secure.

It became very obvious that the design must enable the end-user to test and control the correctness of the specification of extra payments. Control facilities must be provided to ensure security for the users in their work. Although control and test functionality was important for all users, the attitude towards test and control varied between the users. The better the knowledge of the task, the surrounding systems and possible errors, the less important explicit test and control seemed to be. Following statements exemplifies different attitudes towards control and test functionality:

> When you make an extra payment for the first time you would probably like to make a test run to see that it really works correctly. (user comment, evaluation session, February 24, 2004)

and

> there isn't the same protection as in SystemZ … but to make a more flexible solution, then you can't expect it to be strictly user friendly (user comment, evaluation session, February 24, 2004).

Clear Division between Definition, Execution and the Tailoring Process

When tailoring, the user rises from one level of abstraction to another, higher level. From thinking only in terms of the execution of an extra payment the user had to think in more general terms of what characterizes this extra payment, what kind of data were fetched, what variables there were, and so forth. The users had to think in terms of levels, which is not an easy step to take. We found that a clear separation between execution and the tailoring process helped the users to make this step successfully.

The users also started to discuss the division of labour enabled by a system resembling the prototype. For example, one of the users said:

> I think it is very good because then someone is very familiar with how to make a new extra payment and then all employees in the group can run the extra payment. (user comment, evaluation session, February 24, 2004)

Unanticipated use Revealed to the Tailor

Systems that continuously evolve through tailoring aim to support unanticipated use. The possibilities for unexpected use are inevitably limited by the technical design. To support unanticipated ways of tailoring, the system has to provide additional information of what is possible to do and what the limitations are. In the prototype this was achieved by providing data for the user that is not directly applicable to the type of extra payments that exist today. As one of the users expressed it when seeing the opportunity for one of the export systems to also act as input source:

> This is interesting! It opens up new opportunities. It might be like one extra payment uses another payment as a base (user comment, evaluation session, February 24, 2004).

Complexity

We found that the users preferred more information, rather than a less complex tailoring interface, resulting in more tailoring possibilities. Their opinion was that, as tailoring is not routine work, performed several times a day, it is allowed to take extra time. Then it is better to have a more complex interface providing more opportunities to tailor the system.

The Deployment Interface

Simplicity

One thing that was revealed and worth mentioning is that it seems that the deployment interface should be even simpler than an ordinary user interface. The users expressed the opinion that the tailoring interface and the tailoring process may be rather wide-ranging if that allows for a simpler deployment interface.

One Point of Interaction

The development interface in the prototype was a graphical Web interface where the developer could fill in the data that was to be published about the respective source system. During the evaluation of the development interface, the software engineer emphasized the importance of having one point where changes to the data sources are published. The developer should not be forced to make changes in several places in the application in order to extend the tailoring capabilities.

End-User Knowledge Required for Tailoring

Even previous to the evaluation session we had experienced the high expertise of the users not only regarding their tasks but also regarding the data available in the different databases that are part of the IT-infrastructure. The users acquired the knowledge in order to perform the assembly manually. The communication between different systems is normally hidden from the user in a data communication layer for the separate systems. Our prototype is designed to make exactly this communication tailorable. Its deployment depends on the respective expertise of the users.

Task Knowledge

Business knowledge about contracts and payments provides the base on which the users decided what data to collect. Extensive business knowledge was a prominent feature of the results of the evaluation. The users' reflections on which data to collect always concerned different aspects of the business tasks.

System Knowledge

To map requirements regarding the task at hand and the available data, demands expertise regarding the available data in the different systems. And the users knew where to find the data needed for defining a specific extra payment. The prototype just helped with the exact location of the data, for example it guided the user to which fields to use, by listing the fields with examples of the data they contained. However, the user had to understand the sometimes quite cryptic names and know where to look for specific data.

Error Knowledge

All users were extremely aware of which errors could occur, that is, errors concerning the use of the prototype, the IT-infrastructure and the task. Task-specific errors are particularly important for the end-user to overview since they may cause serious consequences for the company if the errors are not prevented. On several occasions during the user tests the users expressed concern about making errors. They made statements like:

> when you work as we do you must know a little about database management, you have to understand how the tables are constructed and how to find the information. And also in some way understand the consequences of or the value of the payment. In other words how you can formulate conditions and what that leads to. (user comment, evaluation session, February 24, 2004)

Organizational and Cooperative Issues

The system for which the prototype was a test would depend on data published by many different surrounding programs. Each one of these systems is itself the subject of both tailoring and evolution. Both the users, and the software

engineer who evaluated the prototype, addressed the necessary interaction with other system owners and the assignment of responsibilities regarding the publication and updating of the connection information and the kinds of data available.

Publication and Update Responsibilities

During the workshops it became apparent that there is already friction in the coordination between the payment system and the changes in the surrounding systems. When one system in the IT-infrastructure is changed, the changes are orally communicated to the owners of other systems that may or may not be affected by the change. For the prototype to function as designed, it was important that the systems that the prototype was expected to communicate with were visible and accessible. The design of the prototype solved this problem by requiring every change relevant to the prototype to be reflected in the published information. In other words, it was designed so that the respective system owners were responsible for keeping their system visible and showing its current status. As the prototype was dependent on accurate just-in-time information, the evaluation revealed a need for coordination concerning publication and updates of surrounding systems and tailoring activities in the prototype.

Collaboration between Developer and End Users

The fieldwork revealed, and the evaluation confirmed, that it is impossible to know what future contracts will look like. Therefore there will always come a time when the end-user wants to retrieve data that is not published in any available view. In this case the system that can provide the data has to be identified and the respective system owner or developer has to be persuaded to implement a new view of the system or update existing ones, and publish the relevant information.

Another issue related to communication and cooperation between users and developers concerned the decision of how much information to make available for the users to do a good job of tailoring. The users wanted to see as much information as possible, provided it was within reasonable limits. In order to have better control over the execution of the system and to decouple maintenance that would not necessarily impact the communication with the payment system, the developers would rather prefer to restrict the user's options. These two perspectives have to be negotiated.

In this company, cooperation between business units and the IT unit works very well. The users evaluating the system were quite aware of the limit of their own competences and knew when to consult the responsible developers. All users frequently referred to developers when they experienced that something was beyond them. None of them considered the necessary coordination and cooperation to be a serious problem.

Summary of Outcome of Evaluation

The evaluation revealed many issues to consider when making a system that continuously evolves through tailoring work in a rapidly changing business environment. The issues could be divided into three categories regarding design issues, user knowledge, and organizational and cooperative issues. Below, the issues are summarized and listed under the respective category.

Design Issues

1. Functionality for controlling and testing changes has to be integrated into the tailoring interface and there must be sufficient technical support for the end-user to estimate and check the correctness of the computation.
2. A tailorable system has to define a mental model that makes a clear division between definition, execution and tailoring. This mental model must be adopted in the tailoring interface and be shared by users, tailors and developers.
3. The tailoring interface also has to reveal potential for unanticipated use to the tailor. This means, that the information flow must, to a certain extent, exceed what is currently necessary.
4. The tailoring interface can be more complex, provided the tailoring process makes the deployment easier. The tailoring interface is not used as often as the deployment interface and additionally the tailoring itself often involves careful thought.
5. The deployment interface should be simpler than ordinary user interfaces.
6. The developer expanding the tailoring capability should only interact with one clearly defined point in the tailorable system, that is, changes are made at one point in the system.

End-User Knowledge

7. End-users must have sufficient knowledge of how the systems are structured and what the systems can contribute.
8. End-users must have solid knowledge of the nature of the task and what data is required to perform it.
9. End-users must have knowledge of which errors can occur and what the consequences of these may be.

Organizational and Cooperative Issues

10. System owners or developers must be responsible for making their systems publicly available within the company. System owners or developers must also be responsible for updating the systems according to external requirements.

11. The necessity to extend the possibilities for end-users to manage the interaction in an evolving IT-infrastructure requires effective collaboration between the developer and end-users.

5.4 Discussion

Our results are applicable in other areas that are similar to telecommunications, and that depend on an IT-infrastructure for a major part of their business and where the development of new products requires changes in this IT-infrastructure. On one hand, our results confirm existing research: Users ask for additional functionality to guide the tailoring and test the outcome (Burnett *et al.*, 2003). We found that users wished to incorporate control of the tailoring process in the form of an outline, preferably in a step-by-step fashion. They also asked for visualization and test facilities in order to check the impact of the separate steps on the end results. The evaluation of the interface allowing software engineers to expand the tailoring possibilities confirms and expands previous research results addressing the developer responsible for the evolution of tailorable systems as an additional stakeholder whose requirements also have to be considered (Chapter Two, Chapter Three).

On the other hand, the results indicate that tailoring in an IT-infrastructure of networked applications provides additional challenges for the design of the software, the competence of the users and tailors, and the cooperation between users and developers. Changes – independently of whether they are implemented by tailoring or by evolving the software – can depend on and affect changes in other applications of the IT-infrastructure and the interaction between applications. This requires coordination between tailoring and development, and cooperation between the persons responsible for tailoring and developing the different applications. And this, in turn, requires a different set of competences from users and developers. The use of an application such as the prototype discussed here, for example, required knowledge of the surrounding systems and their data structures. Developers as well as users have to understand not only the system they are responsible for but also the dependencies between different systems and tasks. Several researchers have discussed collaboration between users and tailors, but not between users, tailors, and professional developers. For example Nardi and Miller's approach (Nardi and Miller, 1991) differs from ours in that they see local developers and programmers as being skilled users, while we take the concepts a step further and state that there is also a need for collaboration between users and professional developers who can perform programming tasks to extend the tailorable software beyond the script level.

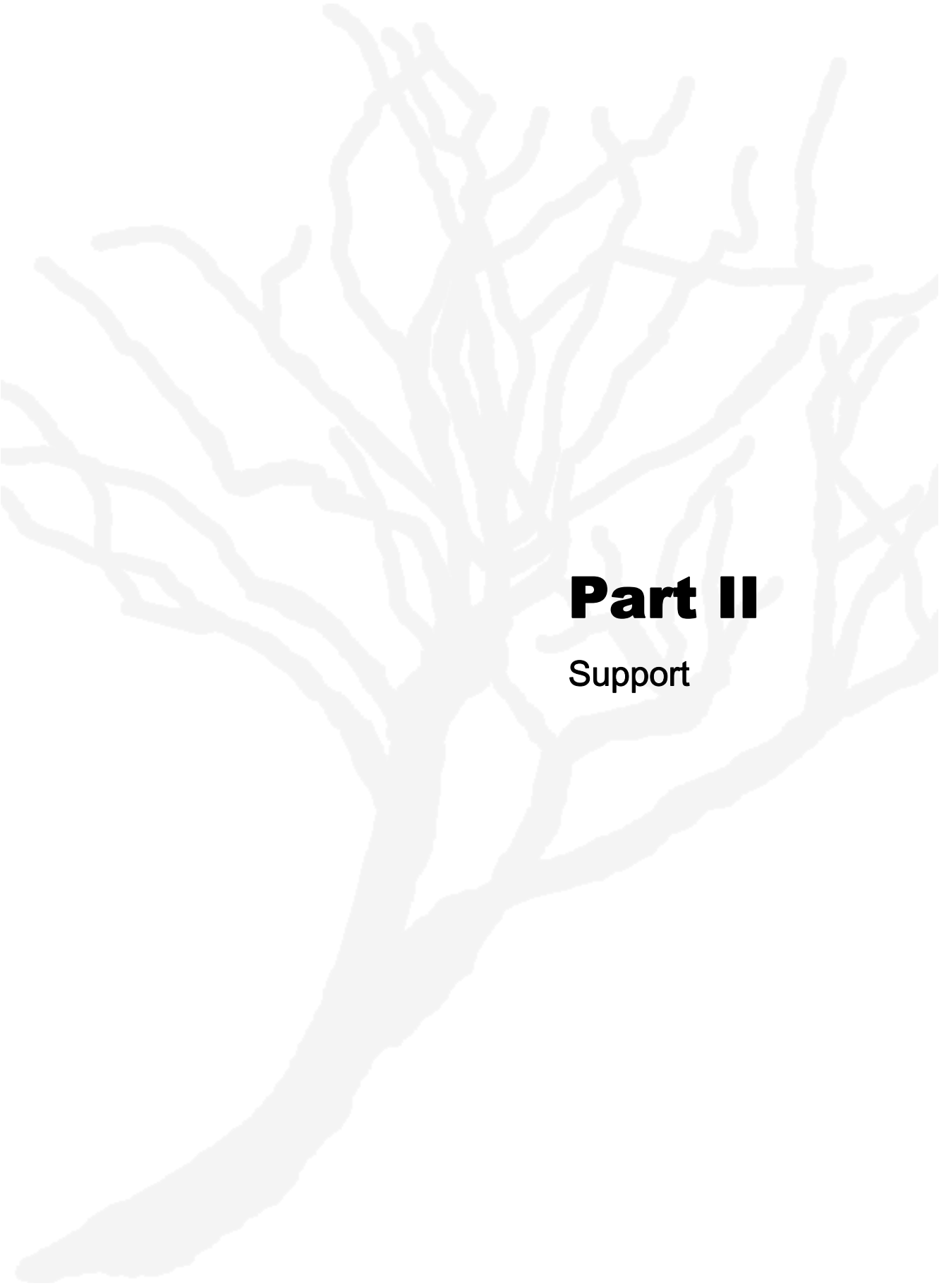
What we claim is that in order to make tailoring really successful, it must be made possible for the tailorable system to evolve beyond the initial intention when building the tailorable system. Kahler's three levels (Kahler, 2001) of tailoring culture – cooperation between tailoring end-users, cooperation

between tailors and users, and the organizational recognition and coordination of tailoring efforts - have to be extended with a fourth level, of organizational support for coordinating tailoring and development activities *involving the cooperation not only between users and tailors but also between tailors and software developers.*

5.5 Conclusion

Allowing end-users to tailor the interaction between flexible applications in an evolving IT-infrastructure requires that the tailoring activities are supported by the design of the system, for example by providing a clear division between execution and tailoring, by revealing potential for unanticipated use, and by supporting single interfaces for changes to the software. It is also essential that the competence of the end-users is sufficient in terms of knowledge of how the systems are structured and what the systems can contribute. End-users must also have substantial knowledge of the task and which errors can occur and what the consequences of these may be. To allow end-users to tailor the interaction between applications in an evolving IT-infrastructure, the organization has to allow for cooperation between users and developers.

The main conclusion of the research described here is that it is possible to provide end-users with the possibility to tailor not only the applications, but if necessary also the interaction between different applications that are part of an IT-infrastructure. The evaluation clearly showed the dependencies between tailoring and the further development of the tailoring capabilities. The evaluation also made it apparent how the different actors were aware of their colleagues' skills and of what each individual could contribute. To ensure a sustainable tailorable system when deploying a system intended to evolve continuously through tailoring, it is necessary to take into account resources concerning various skills and collaboration between users and developers. Without smooth collaboration between the parties an extended fourth level of tailoring culture will not be provided for, and therefore the system will soon become partially obsolete and the competitive advantages provided by the system will decrease dramatically. The results challenge the clear division between software use and evolution on one side and software development on the other side, when developing and maintaining an IT-infrastructure. Collaboration between the end-user and the developer must work satisfactorily in order to achieve tailorable, sustainable software. In other words, in a rapidly changing business environment with continuously changing requirements, such as the one presented in this paper, the tailoring activities have to be coordinated with the software evolution activities.



Part II

Support

Chapter Six

Paper V

Chapter Six

Four Categories of Tailoring as a Means of Communication

Submitted to the Journal of Information Technology, February 2008

Jeanette Eriksson, Olle Lindeberg, Yvonne Dittrich

In a fast changing world, software needs to be increasingly flexible, to support higher reusability and prevent it from expiring too soon. One way to provide this kind of flexibility is end-user tailoring. A tailorable software is modified while it is being used, as opposed to being changed during the development process. Tailoring a system is "continuing designing in use" Henderson and Kyng, 1991, p. 223). It is possible for the user to change a tailorable software by the means of some kind of interface. This means that some design decisions are postponed until the software is up and running. It is the end user who will adjust the program to fit altered requirements through, for example, run-time configuration. Anders Mørch (1995) discusses tailoring in terms of the adaptation of generic software, but tailoring is also applicable to special purpose software. Tailoring is especially well suited for applications used in a rapidly changing business environment. Tailoring can be regarded as a form of End-User Development. In the new paradigm of End-User Development the need for more flexible systems is recognized and the goal is to "empower end users to adapt IT-systems themselves as much as possible, thus letting them become the initiators of a fast, cheap and tight co-evolution between themselves and the systems they are using." (Klann, 2003, p. 5). This is exactly what tailoring enables.

There are several aspects concerning user knowledge, technical issues and business organization that have to be satisfied to make a tailorable system work in the long run (Chapter Five). Among other things, the tailorable software must be supported by a collaboration between developers and users (Chapter Five), since users in the context of tailorable software are to be regarded as co-designers (Fischer, 2003, Fisher and Ostwald, 2002). Tailoring should therefore be looked at from two perspectives, both the user perspective and the system perspective (Stiemerling, 2000), as the user perspective reflects how users work with tailoring and the system perspective elucidates important issues from the developers' point of view.

In our research, we have cooperated closely for several years with a telecom company, and during this industrial cooperation, it has been found that there is a need to look systematically at tailoring, in order to understand the phenomenon

better and to be able to make better informed decisions of the kind of tailoring to adopt in new software. We have found that there is a degree of uncertainty surrounding flexibility and tailoring. This uncertainty is revealed in the discussions of which flexibility is needed and how to implement the flexibility when building tailorable software. The participants in the discussions have different viewpoints and individual experiences of flexibility. This is the case when developers discuss with users, and even when developers discuss with other developers. They do not have a common ground from which to start the discussion. To make software successful it is important that there is a consensus between users and developers on how the software should work. This is especially important when the users will continue the development or evolution of the software at use time. Users and developers must speak the same language. To put it another way, the parties must have a common understanding of the phenomenon to come to an valid agreement (Preece *et al.*, 2002).

With this chapter we want to contribute to putting design for tailorability on the agenda and to systemize tailoring in a way that facilitates design decisions by providing a common base for discussions. What we want to achieve is a categorisation of tailoring that can be useful in communication between users and developers in industry, during the requirements phase of a development project. The result is a categorization that maps the user perspective on tailoring to the system perspective. The resulting categorization consists of four categories of tailoring: customization, composition, expansion and extension. The categorization is found to be applicable in three research cases and the categories are also recognized in industry.

Some categorizations of tailoring already exist (Fischer and Girgensohn, 1990, Mørch, 1995) but they are not explicitly intended for communication between users and developers in the requirement phase. A comparison between our approach and the pre-existing approaches is discussed in Section 6.4.

The rest of the chapter is organized as follows (Figure 6 : 1). We start by discussing how change is perceived from a user perspective (Section 6.1.1) and how tailoring can be accomplished from a system perspective (Section 6.1.2). We continue by presenting a new possible categorization for systemizing and classifying tailoring (Section 6.1.3). Then we present three different tailoring approaches to see if the categorization can be applied to them (Section 6.2). We present an overview of three approaches concerning the kind of flexibility provided, from the users' point of view (user perspective) and regarding techniques for implementing tailoring (system perspective). We look at one of our own examples (ContractHandler, Section 6.2.1) and two other approaches within the area of tailoring and End-User Development, namely Anders Mørch's work with application units (Mørch, 2002, Mørch and Mehandjiev, 2000) (BasicDraw, Section 6.2.2) and the research done within a project concerning the tailorability of CSCW (Computer Supported Cooperative Work) systems (Stiemerling, 2000, Stiemerling et al., 1998) (Search Tool, Section

6.2.3). We have chosen these approaches because they are published and well known and provide a description of both use and design. Additionally, they represent different types of applications. Other relevant approaches exist, of course, for example CHIPS (Wang and Haake, 2000), Oval (Malone *et al.*, 1995) and Click (Rode *et al.*, 2006). After the cases are presented, some interviews performed at a major telecommunication operator in Sweden are introduced (Section 6.3). Both developers and users were interviewed to determine if they could distinguish the four categories of tailoring. The paper ends with a discussion of the results (Section 6.4) and we make a comparison with related work. Finally we draw some conclusions in the Summary (Section 6.5).

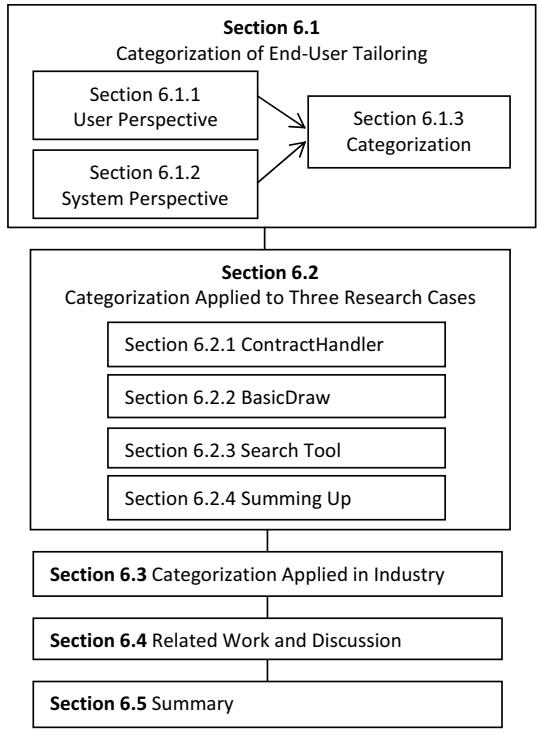


Figure 6 : 1 Overview of Chapter Six

6.1 Categorization of End-User Tailoring

When discussing with people in industry what we here call tailorability, they seldom think of or talk about this kind of software in terms of tailoring; instead they simply call it flexibility. When observing work with tailorable software, or interviewing or discussing tailorable software with people in industry, it emerged that there was confusion in the discussions between participants when

discussing flexibility. The reason for this is that they view flexibility from different perspectives. Flexible software is one thing when using it and a totally different thing when building the software. There were even misunderstandings between the developers themselves. The reason was found to be that the perspective on the software changes seamlessly between a user and system perspective. The developers especially make this shift without thinking about it. The reason is of course that they must take both perspectives into account in order to make good software. The fact that the differences between the two perspectives are considerable, and that the shift in perspectives is unconscious, makes discussions about flexibility very complex. Under such circumstances it is hard to reach a consensus about which flexibility to implement and still be convinced that the chosen type of flexibility is best for the situation. These difficulties motivate a categorization of tailorability that takes into account both a user perspective and a system perspective. The user perspective represents which changes can be made, or the intention of the activity, whilst the system perspective corresponds to how the change is achieved in the system (on a high level).

6.1.1 User Perspective

Tailoring is all about change. From our empirical studies at the telecom company we have observed that it is natural for the user to think of four kinds of change:

1. adjust
2. combine
3. create
4. add

Set Parameter Values

To *adjust* software means small changes such as *setting parameter values.* Let us take a simple example. Imagine a flexible, text based calculator program with the main purpose of calculating the four fundamental rules of arithmetic. The predefined way to write decimal numbers is with a point e.g. 1.2. If the user prefers writing decimal numbers with a comma the user can adjust the program. The program allows the user to write " '.' replace with ',"' and the program will interpret a comma as a point in the future, e.g. 1,2 is translated to 1.2. The software is *customized.*

Link Different Existing Components

The second type of change means to *combine different components* to achieve the right functionality. For example, imagine the same calculator application as above. Say that there are four components, each of which handles one of the four rules of arithmetic. When the user writes 35 '+' 5 '-' 5 two components are

activated one by one. Say that the user wants to calculate a sequence as, for example, 35 '+' 5 '-' 14 '/' 2 '-' 14 six times with the difference that the numbers varies each time. Then the user can temporarily store the pattern of the sequence to avoid writing the signs every time. The user writes "pattern on '+' '-' '/' '-' " and the command make it possible for the user to write 35 5 14 2 14 and get the result 19. In this way the components (+, - and /) are connected together temporarily. A *composition* is made.

Creating a New Component

The third way to change the software is to *create a new component* out of pre-existing components. We continue with the example of the calculator program. This type of change means that the user combines several rules of arithmetic and names the combination, so that it can be used by giving the name as a command. For example, say that the user wants to use the combination + then – often, for example 35 + 5 - 5. Then it is useful to save the combination under the name c1. The user can then use it by writing c1 35 5 5 and thereby get the answer 30. By linking components the software is *expanded* by a new component. It is also possible for the user to use the new component as a springboard for other calculations, for example write c1 35 5 5 '/' 5 and get the result 7.

Insertion of Code

To change by adding is to make a component by *adding code* to the software. For example, the user wants to do something new in the calculator program. The user wants to compute the factorial of 5, e.g. 5!. The calculator does not have this functionality and the user must therefore write her or his own component. This can be done by using the multiplication component recursively and saving and naming it '!'. When the user writes 5 ! the new component first calculates 5*4 by calling the multiplication component. The answer, 20 and (5-2) is then sent to the multiplication component which calculates 20*3 (=60). The component is called once again and this time it calculates 60*2 and the calculation ends and the result of 5! is 120. By inserting code the software is *extended*.

We end up with a classification of change with four types, named: *customization*, *composition*, *expansion* and *extension* (Table 6 : 1).

Tailoring from a user perspective		
Customization	A	Set parameter values
Composition	B	Link different existing components
Expansion	C	Creation of a new component.
Extension	D	Insertion of code.

Table 6 : 1 Tailoring from a user perspective

6.1.2 System Perspective

When we look at change from a system perspective the question is "What happens in the system when tailoring is performed?"

For the users to be able to make changes to the software the software has to be flexible enough to adapt to the changes. Typical ways of achieving flexibility are parameterization, configuration, inheritance, generation and extension (Jacobson *et al.*, 1997). The five ways of achieving flexibility can be conceptualized into five ways of accomplishing tailoring from a system perspective. The only difference from Jacobson's et al. description (1997) is that the involvement of the developer is removed, i.e. the flexibility will be handled by the software itself as a reaction to actions taken by the user.

Parameterization means to set a parameter, whereby the application behaves in a specific way based on the value of the parameter. Rephrased, the parameter is interpreted by existing code to achieve the change.

Configuration means that components are connected to each other. In other words the relationship between the components is redefined by the application.

Inheritance, in terms of tailoring, means to use a component as a starting point and then specialize it to suit the altered requirements. In this case it means that all predefined components, as well as newly specified components, are treated uniformly and can thereby be a base for further tailoring in the future.

Generation can be used to create derived components or relationships between components, which means that code is generated to form those entities.

Extension means adding small attachments to other components. In the context of tailoring it means that the user in some way writes code that is added to the software, thereby changing its behaviour.

Table 6 : 2 summarizes the discussion.

Tailoring from a system perspective		
Parameterization	I	Interpretation by existing code
Configuration	II	Definition of relationships between components.
Inheritance	III	New and predefined components are treated uniformly
Generation	IV	Code generation (optional)
Extension	V	New code is added.

Table 6 : 2 Ways of achieving tailorability from a system perspective

In the next section the user and system perspectives are combined to form categories that take both perspectives into account.

6.1.3 Categorization

The end users have four different ways to manipulate the tailorable software; customization (A), composition (B), expansion (C) and extension (D) (Table 6 : 1), and we have five different ways to achieve adaptability (I, II, III, IV and V) (Table 6 : 2). To be able to form a categorization that takes both the user and the system perspective into account, the two approaches must be unified. Since it is the end users that perform the tailoring we begin with the terminology originating from the user perspective, and we pose a question to be able to match the user perspective to the system perspective:

What happens in the system in the respective cases, when…

…*customization,*

…*composition,*

…*expansion* and

…*extension* is carried out?

As shown in Table 6 : 3, customization (A) does not pose any problem. There is a 1:1 relationship between the user and system perspective.

- The change is interpreted by existing code

But when composition (B) is performed, it means that

- a connection between components is achieved and this connection can either be predefined (II), or it may be necessary
- to generate some code (IV).

This corresponds to two ways of doing tailoring from a system perspective (II and IV in Table 6 : 3).

When expansion (C) is carried out three different things may happen in the software, which can be a source of confusion. When performing expansion (C),

- different components are related to each other (II), but the same thing is done when doing composition (B). The difference is that expansion (C) means that
- a new component is created and that the component is treated in the same way as other components in the software (III). Accordingly the new component can act as a base for new composition or expansion.
- It is also possible that some code is generated (IV) to accomplish this new component.

The same confusion can arise when talking about extension (D) even if the system response is limited to involve two different ways of achieving adaptability. Expansion (D) means that new code is added, but

- the code can be generated by the software (IV) or
- written by the user (V).

User categories \ System categories	I	II	III	IV	V
A	x				
B		x		x	
C		x	x	x	
D				x	X

Table 6 : 3 User and system perspective in combination

By combining the user and system perspectives we end up with a new categorisation (Four-to-Five categorization) which is summarized in Table 6 : 4.

	User Perspective	System Perspective
Customization	Set parameter values	Interpretation of existing code
Composition	Link different existing components	Definition of relationships between components.
		Code generation (optional)
Expansion	Creation of a new component.	Definition of relationships between components.
		New and predefined components are treated uniformly
		Code generation (optional)
Extension	Insertion of code.	New code is added.
		Code generation (optional)

Table 6 : 4 The four-to-five categorization of tailorable software

Customization is the simplest way of doing tailoring. It means that the user sets some values for one or more parameters and those parameters manage the functionality that is used. The existing code interprets the parameters and the corresponding functionality is put in operation.

Composition means that the user has a set of components to choose from and he or she can connect them in specific ways to reach the desired functionality. The software can represent the connection in different ways. It is possible to implement possible connections in advance and when a specific combination is chosen, a particular interface relates the components. But it is also possible for components to be connected by generated code.

Expansion also means that the user chooses components out of a set, but the difference is that the users' combination of components is built into the system as an integrated part. The new component is treated in the same fashion as the

predefined components and will be accessible in the set to choose from next time the software is tailored.

Extension is the category which provides for the highest flexibility. It means that the user writes code that is integrated into the system, either by wrapping up the new code in system generated code or, if written in a predefined way, just adding it to the code mass of the software. The user can either write the code in a high level language or in a visual programming language.

The four-to-five categorization makes it possible to categorize tailorable software by starting to consider what the intention of the user action is. Is it to set some parameters to make the software behave in a specific way? Or is it to combine different components to reach the goal? Or is it that the user needs to make a new component? Can the component be created by assembling pre-existing components or does the user have to write some code? These questions act as an entrance to the categorization. Thereafter we can continue to ask what happens in the system. When a mach is found, the categorization is completed. The strength of the four-to-five categorization is that a match is required. It forces the users and developers to look at the system from both a user and a system perspective at the same time, which makes the discussion of flexibility more distinct.

6.2 The Categorization Applied on Three Research Cases

In this section three research cases are presented and then the new categorization is applied to them one by one to see if the applications can be described by the categories. We examine three research cases; one case of our own (ContractHandler, Section 6.2.1), one from Anders Mørch, (BasicDraw/KitchenDesign, Section 6.2.2) and one case performed by Stiemerling and his colleagues (Search Tool, Section 6.2.3). The overall differences between the approaches are shown in Table 6 : 5.

Type of software	Stand-alone application	Distributed application
Special purpose software	ContractHandler	Search Tool
Generic software	BasicDraw	

Table 6 : 5 Differences between the three research approaches

6.2.1 ContractHandler

The prototype presented in this section is an experiment that uses the Java reflection API as a means to implement a tailorable system. The background to and idea behind the experiment was a research project in which we collaborated with two industrial partners. The goal of the project was to investigate means of developing flexible, adaptable and modifiable software systems (see (Dittrich and Lindeberg, 2002)). The system the prototype was modelled on is an

application used by one of the research partners, a telecommunication operator. It was possible to anticipate the type and structure of some of the changing requirements, and tailoring was a possible way to make the system modifiable.

The system is used for computing certain payments[1] which are triggered by certain physical events. The receiver of the money and how much should be paid are determined by what contract(s) are valid for the event. A contract consists of a set of parameters that determines what data the contract has to contain. To make new types of payments, new types of contracts have to be implemented and that is what the prototype allows end users to do.

The prototype, also called ContractHandler, can be seen as an example of an explorative prototype (Dittrich and Lindeberg, 2002). We wanted to gain an understanding of the complexities related to this approach. Our aim is to give the user the opportunity to add components, or building blocks, to the program in a controlled way which does not require any programming. To do this we use a dual-interface: a traditional base-level program and a meta-level program that provides tailoring for the base-level program. (Dittrich and Lindeberg, 2002)

The prototype is divided into two levels; the meta-level and the base-level. A new contract is created in the base-level of the program by instantiating a contract type and the contract types are created in the meta-level of the program. Two catalogues, one storing contract types and the other parameter classes, implement the connection between the two levels. In the meta-level of the prototype, the new contract types are created and stored in the contract type catalogue. In the base-level the same classes are used as part of the program. The parameter class catalogue is used by the meta-level to identify which parameters exist and by the base-level as part of the program. By doing it this way, we isolate the meta-representation to the meta-level and when running the base-level it acts as a non tailorable system. This means that in the base-level we can provide for performance that is good enough, despite reflection overhead.

User Perspective

The contract types are created in the meta-part of the program. The prototype contains two tailoring interfaces: one that does not require any programming skill and another more advanced interface requiring basic knowledge of Java. Here, we will first describe the simple interface and then come back to the advanced interface.

Tailoring activity (a): When a user wants to create a new contract type, all existing contract types are displayed. This is done by collecting all the class files from the contract type catalogue in which they are stored. The end user

[1] To protect the business interests of our industrial partner we can only give an abstract description of the system.

chooses what contract type she or he wants to have as super class for the new contract type. To make it easier for the user to make a decision as to what contract type is the most suitable, the parameters and the methods of the contract type are also displayed. Java reflection API provides the necessary methods for this.

The next step is to collect all possible parameters for the new contract type. To find the set of all possible parameters, the program collects all classes in the catalogue dedicated to parameter classes.

Thereafter all possible parameters for this Event type are shown to the end user, who selects which ones to include in the contract type. A contract type is a composition of parameters. The parameters that are inherited are automatically selected and cannot be deselected.

In conclusion, from a user perspective tailoring activity(a) can be seen as either *customization*, *composition* (since the user links different parameters) or *expansion* (since a new contract type is created).

Tailoring activity (b): In the more advanced interface the end user or a software developer can also add and change methods within the new contract type. Since the system will already have constructed a working implementation, some modifications may be done even by end users having only rudimentary knowledge of programming. However, the user has full access to Java, which means that the end user can make unanticipated changes. With this interface it is possible for the end user to extend the capabilities of the system. In practice it means that the end-user can add methods or implement method bodies that differ from the generated ones, to achieve the intended functionality.

In conclusion, from a user perspective tailoring activity(b) can be seen as *extension* (since the user writes some code).

Use: When the end user wants to create a new contract, i.e. create an object from a contract type, all of the concrete classes are fetched from the contract type catalogue, their names are presented and the end user chooses which contract type to create a contract from. A contract is then created which has parameter objects without values. The object displays itself by forwarding all requests to the parameters. The same principle is used for storing and checking errors. When the user has put values in all slots the error check is forwarded to every parameter object. The parameter object checks that the value has the right format and is within the given limits. When a value is incorrect, the slot is marked and the user has to insert a new value. The new contract is not stored until all values are correct.

System Perspective

Tailoring activity (a): Inheritance, together with the meta representation and the inner structure of the contract types, is essential to the prototype. The Events are super classes of the contract types. An Event has a set of parameters and the

contract type is made up of a set of these parameters. Some parameters are compulsory for all contract types belonging to an Event; they are put in the Event so that they are present, thanks to the inheritance principle, in all the contracts. For example all contracts must have a contract id. Adding functionality by use of inheritance has its shortcomings since inconsistency occurs when a contract type in the hierarchy above is removed. In this case the history of the change has to be preserved for business reasons, which means that no contract type can be removed.

The meta-level of the program is constructed as a meta-model implemented with classes. The contract types correspond to objects of the class Metaobject. The source code for the new class is generated from the meta object. The java source code is then compiled and a class file is produced. The file is stored in the contract type catalogue.

In conclusion, from a system perspective tailoring activity(a) can be seen as either *configuration* (since a relationship is created between the parameters), *inheritance* (since new contract types inherit from Event) or *generation* (since code is generated to create a contract type).

Tailoring activity (b): The user can modify the generated contract type in the advanced interface. The user writes code and the code sequence replaces the equivalent generated code and new code is generated. The contract type is recompiled automatically.

In conclusion, from a system perspective tailoring activity(b) can be seen as either *generation* or *extension*.

Use: A contract is essentially a collection of parameters. In the existing system in use some of the parameters are very complex and some even collect values from other systems. This makes it natural to represent every parameter by an object. Most of the methods in the contracts are implemented using delegation to the parameters. For the three main methods in the contracts - checking, storing and displaying themselves - there are corresponding methods in the parameter classes. This is a vertical design where one class takes care of one type of parameter through the whole program instead of the more normal three-layer architecture (interface, logic and storing). This design makes it very easy to add new parameter classes to the system.

Conformance to Categorization

According to the four-to-five categorization, ContractHandler implements

- *expansion* by allowing the users to relate components to create a new composed component. The assembled parameter-components are incorporated into the software to be managed in the same way as the predefined components and by allowing previously added components to be a base for new contract types (Tailoring activity(a)), and

- *extension* by letting the user write code that is incorporated into the existing code. This is done by the software, through generating new code and compiling it. (Tailoring activity (b))

6.2.2 BasicDraw/KitchenDesign

The work presented in this section is work done by Anders Mørch. Mørch works with tailoring issues using components called application units, where tailoring is an option. (Mørch, 1997, Mørch, 2002, Mørch and Mehandjiev, 2000) It addresses the problem of software reuse by creating new software from existing systems. The example differs from our own approach in that it uses generic software as a base for tailoring, whilst the ContractHandler is designed for a special purpose from the beginning. The software system focused on is a generic application called BasicDraw, a graphical editor with the normal functionality found in, for example, McDraw or Paint. The varying levels of user experience, and different ways to accomplish tasks in an organization, make it likely that tasks change as the generic application is being used. Mørch sees the transition between use and tailoring as being identified by a breakdown. Breakdowns happen e.g. when an application is no longer sufficient for a task. The breakdown should leave the user, however, with a handle into the application. The handle can be used to access the parts of the application that have to be dealt with to repair the breakdown. The handle may be a button, a menu item or a window. The sequence of commands required to repair a breakdown is not totally smooth. (Mørch, 1997, Mørch, 2002, Mørch and Mehandjiev, 2000)

The graphical user interface of a generic application is composed of graphical presentation objects such as buttons, windows, toolbars etc. The user interface of BasicDraw is composed of application units, cognitive building blocks integrating multiple representations. Application units are reusable software components implemented as GUI widgets extended with event handlers that take over when the user wants to tailor. Application units consist of three parts: presentation objects or user interface, rationale, and implementation code. (Mørch, 1997, Mørch, 2002, Mørch and Mehandjiev, 2000)

The application units are to a large extent independent and can be tailored separately from other aspects; some application unit aspects are, however, also dependent on others. Changing one aspect (presentation, rational or implementation) may therefore require an update of other aspects or interfaces. (Mørch, 1997)

Presentation objects reflect the structure of tasks in the domain while rationale components describe the structure of implementation code. But rationale components are not interpreted or executed by the computer. The rationale captures the application's requirements for design and use. A rationale fills in the gap between the user interface and the implementation code, making a gradual transition from use to tailoring possible. Rationale components can

include representations from, for example, the kitchen design domain, modular arithmetic and programming code. The representations are presented in rationale viewers. (Mørch, 1997)

When moving from use to tailoring, all three parts of the application unit have to be reached from the user interface. The presentation object part serves as a handle. A handle accepts input from the user and forwards it to the application. Event handlers make this possible. An application unit has four event handlers. One event handler is for normal use, while the other three are for tailoring activities. An end user may select between the different event handlers by pressing different keys.(Mørch, 1997)

User Perspective

Through the tailoring capabilities BasicDraw can be transformed into a specialized drawing program for a specific domain. Anders Mørch gives the example of kitchen design. To be used for kitchen design, BasicDraw needs to be extended to make it possible to draw graphical symbols representing a sink, stove, refrigerator, standard sizes of appliances and cabinets etc.

Tailoring activity (a): At the first tailoring level it is possible for the end-user to edit attribute values in the application. Attributes that can be edited are, for example, height and width for shapes, or titles of menus or menu items (Mørch, 1997) i.e. we can make a special “kitchen menu” where we can place special objects (sink, stove, refrigerator etc.) and set the height and width to 60 cm, as well as colour the refrigerator square white.

In conclusion, from a user perspective tailoring activity(a) can be seen as *customization*.

Tailoring activity (b): From the user interface it is possible to access the existing graphical shapes and make new shapes by first copying and then modifying. A graphical shape is a class in the underlying system. Here the end user has access to all the methods defined in the class. The end user renames the class, calling it, for example, KitchenCabinet, and writes the extension code he or she needs. The user can, for example, specify that the shape cannot be bigger than 60 cm. (Mørch, 1997)

The extension editor makes it possible to tailor the application by changing the program code during runtime. The software components are encapsulated as a glass box. This exposes program code. The code cannot, however, be modified. (Mørch, 1997) The new code is built on top of the existing code for safety reasons: none of the old code in BasicDraw may be removed. The end user is not allowed to delete generic implementation code but can delete his or her own extensions.

In conclusion, from a user perspective tailoring activity(b) can be seen as *extension* (since the user adds code).

Use: In use the normal functionality connected to the presentation object is executed. The user can draw, move and arrange graphical elements.

System Perspective

Tailoring activity (a): An application unit has a presentation object. These presentation objects represent real world objects. The user edits the attribute values of the application unit to change the appearance. The new attribute values are saved and later used by the software to present the object in the desired way.

In conclusion, from a system perspective tailoring activity(a) can be seen as *parameterization*.

Tailoring activity (b): The language used in the application is the object-oriented programming language Beta. The Beta language provides for inheritance and virtual binding. This makes extensions with no overriding possible. Extensions can be made to other extensions if necessary. The extension code is saved in an extension file. The new code is compiled and must be linked to the existing code before the application can be re-executed. (Mørch, 1997)

In conclusion, from a system perspective tailoring activity(b) can be seen as either *extension* (since code is added) or *inheritance* (since extension can be made on other extensions).

Use: What happens in the application when using the tailored software is not revealed in the papers, but it is likely that the system acts in the same way as the generic version.

Conformance to Categorization

According to the four-to-five categorization, BasicDraw/KitchenDesigner implements

- *customization* as parameters are set to choose functionality and the software exposes the presentation object according to the parameters by interpretation in existing code. (Tailoring activity(a))
- *extension* is provided for as the user can add code to the application and the new code is saved in an extension file and compiled before the software can run again. (Tailoring activity (b))

6.2.3 Search Tool

In the following section a search tool is presented. This differs from the previous approaches in that it makes use of a predefined component model, namely JavaBeans. This system is also distributed, while the ContractHandler prototype and BasicDraw/KitchenDesigner are stand-alone applications.

A research team at Bonn University is working with tailoring CSCW-systems. They have constructed a search tool that makes it possible for different users to tailor the presentation of search results (documents), the handling of search results, and the search space. The search tool is intended as a part of the POLITeam-system, which provides electronic support for the work of the German government in Bonn and Berlin (Stiemerling et al., 1998). The POLITeam-system provides for asynchronous document-based cooperation and shared workspaces in a virtual desktop setting. The project had a participatory approach and a number of requirements were materialized that could not be addressed by the tailoring mechanisms in the commercial groupware platform that was intended to be used from the beginning. Some of these requirements concerned the search tool that was used in the groupware platform to search for documents. Moreover, the requirements appeared to alter over time and the change was quite short-lived and task-dependent. This resulted in the construction of a component-based tailorable search tool that could meet the end users' requirements.

To construct the tailorable search tool a set of components was designed. There are attribute components, invisible components (search engine, result switches), button components and output components (result lists). The search tool is implemented using the JavaBeans component model. JavaBeans interacts via events. The components depend on three kinds of events. Click events are used to transmit user commands from the graphical user interface, from button components to the search engine or to an output component. Attribute events transmit altered search attributes to the search engine. Result events are used to exchange search results, from search engine to result switch or result list, or from result switch to result list. (Stiemerling, 2000)

User Perspective

The action the user makes to configure a search tool involves choosing among the predefined components in the tailoring mode. The graphical representation of the chosen components is presented in the interface. In the graphical interface for the tailoring mode the user can connect different components. For example, the user can choose to use two attribute components, namely document type and document name. Naturally, the user wants a search button and an output window. But the user also wants to be able to make a copy of a specific selected document that has been found during the search. Accordingly, the user needs another button component, a copy button, which makes it possible to make a copy of a selected document (Stiemerling, 2000).

Tailoring activity (a): The components have a graphical representation that the user can combine in different ways. The components are equipped with circles in different colours and fillings representing output or input ports. By connecting different ports with lines the components are assembled into a new configuration. The graphical representations are used in a compositional

technique that allows the user to instantiate new components, link the different components together, disconnect ports or remove instances.

In conclusion, from a user perspective tailoring activity(a) can be seen as either *composition* (since the user relates different components to each other) or *expansion* (since a new component is created).

Tailoring activity (b): Component instances can be grouped into a composite component instance. Accordingly, it is possible to use old compositions as a starting point for tailoring. This means that it is possible to view and manipulate a composition on different levels of complexity and abstraction which in many cases results in a reduction of the number of components to combine. Alternative search tool compositions can be selected from a menu. This makes it possible for the end user to tailor the tool very rapidly.(Stiemerling, 2000)

In conclusion, from a user perspective tailoring activity(b) can also be seen as *composition* or *expansion* since a new component is created.

Use: The configurations from the tailoring activities can be used in use mode.

System Perspective

Tailoring activity (a): To be able to manipulate the internal representation and the connection to the actual application, a simple runtime tailoring environment based on BeanBox was used; this is the IDE (Integrated Development Environment) supplied by JavaSoft together with the JavaBean component model. The user interface and the mechanism for putting together the different components were, however, radically modified. The new BeanBox interface shows those connections between components and ports which can be manipulated by the user. The modified BeanBox preserves the composition using a composition language; it depends on direct connections between components, unlike the ordinary BeanBox that uses generation of code and compilation. (Stiemerling, 2000) The modified BeanBox also connects the representation of the composition to the actual search tool. This is done by reading a file describing the search tool at start-up. Proxy objects or “wrappers” are then created which manage a specific instance of a component.

In conclusion, from a system perspective tailoring activity(a) can be seen as *configuration* since a relationship between the components is defined by the composition language.

Tailoring activity (b): As component instances can be grouped into a composite component instance and the descriptions files are saved in a shared dictionary, the configuration files can be selected as menu items in both the use and tailoring mode (Stiemerling, 2000), the existing compositions are regarded as equal to the elementary components, as the components are used in the same way by the system. The configuration file of the composition is integrated into a new composition file containing the expanded composition.

In conclusion, from a system perspective tailoring activity(b) can be seen as either *configuration* or *inheritance* in the sense that the new search tool is used as a complex component and can thereby be used as a base for building new tools.

Use: Somewhat simplified, the search tool has the following functionality: The control button triggers the search engine and the search results are transported to a switch. This switch has been customized to channel all documents that correspond to certain criteria, e.g. those found on the user's own desktop, to one specific result list. Other documents that correspond to another criterion, found elsewhere, for example, are displayed in another result list. (Stiemerling et al., 1998).

Conformance to Categorization

According to the *four-to-five categorization* the Search Tool implements

- When the end user chooses, connects, withdraws and reconnects components and the component instances are wrapped into a proxy object, this would be regarded as *composition*. (Tailoring activity (a))
- It is also possible to use a search tool that someone else has tailored and make some changes to it so that it suits the end user's needs better. This is achieved in the search tool by a list of available components and previously configured search tools. The components are described and have a clear graphical representation that makes the configuration of a search tool similar to solving a puzzle where it is possible to combine the various pieces in a variety of ways. Due to the fact that old compositions are wrapped up in a proxy object and can be used as a base for new compositions, *expansion* is provided for. (Tailoring activity (b))

6.2.4 Summing Up

As shown in Table 6 : 6 and Table 6 : 7 the research cases can be classified in several different ways if only one of the two (user and system) perspectives are considered.

User perspective	*ContractHandler*	*BasicDraw*	*SearchTool*
customization	(a)	(a)	
composition	(a)		(a) (b)
expansion	(a)		(a) (b)
extension	(b)	(b)	

Table 6 : 6 The three research cases from a *user perspective* (a and b refer to the tailoring activities presented in Sections 6.2.1-3)

System perspective	*ContractHandler*	*BasicDraw*	*SearchTool*
parameterization		(a)	
configuration	(a)		(a) (b)
inheritance	(a)	(b)	(b)
generation	(a) (b)		
extension	(b)	(b)	

Table 6 : 7 The three research cases from a *system perspective* (a and b refer to the tailoring activities presented in Sections 6.2.1-3)

The Four-to-five classification of tailorable software was shown to be applicable to all three research cases and as summarized in Table 6 : 8 the categorization makes it possible to classify different tailoring activities unambiguously.

category	*ContractHandler*	*BasicDraw*	*SearchTool*
customization		(a)	
composition			(a)
expansion	(a)		(b)
extension	(b)	(b)	

Table 6 : 8 Summary of the classification of the research cases (a and b refer to the tailoring activities presented in Sections 6.2.1-3)

6.3 The Categorization Applied in Industry

To be able to determine if the four categories in Table 6 : 4 are recognized in industry by the participants in software projects, we interviewed developers and users at a telecom company in Sweden. The telecom business is characterized by fast changes. For example, new services continuously evolve and consequently the supporting business systems have to adapt to the altered requirements. The telecom company is dependent on flexible software where the user can alter the software when the need arises. Accordingly they have a lot of tailorable systems running, which means that this type of business is well suited for investigating whether the four-to-five categorization is recognized in the industry.

We interviewed six developers and four users. The developers represented various systems and positions, which means that they worked with different systems and had different tasks. The developers are programmers, system owners and technical project leaders. The users all work with several different systems, but their main tasks are with the same system. The users also represent different work roles. They are a system coordinator, work manager, users with

responsibilities for working with new requirements, and users helping out with further development of the system.

We performed ten interviews, each lasting approximately one hour to one and a half hours. The interviews were semi-structured (Robson, 2002) which means that all the respondents were asked the same questions in the same order, but follow-up questions were asked and explanations to the questions were given.

To be able to discuss the four categories on equal terms with both developers and users, the categories were translated into four written examples representing the categories. The examples were at a rather high level, free from unnecessary details, but concrete enough to make it possible for the respondents to discuss the examples. The examples were written from a general point of view and were not limited to the tasks in the telecom company. The examples can be found in Appendix A.

The respondents had to answer in which kinds of situations they thought the different examples would be suitable, and if they could recognize the different examples in software they worked with or had knowledge of. They were also asked to name the systems corresponding to the different examples and to describe what it was in the systems that resembled the example in question.

The interviews were conducted according to a specific order. The respondents first read all four examples and then they answered the questions.

All developers except one (the technical project leader) only recognized the examples of customization, composition and expansion. Some of them expressed certain scepticism about the fourth category, extension, where the users are allowed to write some code on their own. They considered this to be too risky. The technical project leader, however, knew about a system that implemented extension. It was a small system handled by the department of sales and no developer was involved in making the changes. The salesperson responsible for the system writes code to make changes requested by the other salespersons at the department. The reason for the technical project leader being aware of the system was that there was a discussion about whether the IT-department should handle the system instead, despite the fact that changes to the system would not take place as quickly as the salespersons were used to.

It was a similar situation when it came to the users. Only one of the users (the system coordinator) recognized the small system that implements extension. This is due to the fact that the system coordinator is the only one of the users who has a good overview of which systems are used in departments other than their own. Three users were familiar with an administration tool that could be categorized as extension. The users did not think that extension was too risky for a user to perform. They expressed the opinion that if a user was to make such changes, he or she must certainly know what he or she was doing. As mentioned, all the users had one system in common that they all worked with.

All of them recognized the other three categories in that system, but they also recognized the examples in other systems they worked with.

In conclusion it can be said that the categorization made it possible for the respondents to reflect over differences in system infrastructure when it comes to tailorability.

6.4 Related Work and Discussion

Some classifications of end-user tailoring already exist. Mørch, for example, has identified three different levels of tailoring (Mørch, 1995). The higher the level, the more radical the changes that can be carried out and the more expert knowledge one must have. The three levels are:

- Customization
- Integration
- Extension

Customization is defined as:
"Modifying the appearance of presentation objects, or editing their attribute values by selecting among a set of predefined configuration options." (Mørch, 1995, p. 44).
Integration means:

"Creating or recording a sequence of program executions that results in new functionality which is stored within the application as a named command or component." (Mørch, 1995, p. 45).

Extension is the most deep-going level of tailoring and it:
"…is an approach to tailoring where the functionality of an application is improved by adding new code." (Mørch, 1995, p. 47)

The categorization is done in respect to generic software. The purpose of the tailoring levels is to bridge the gap between the user and the implementation code.

Fischer and Girgensohn (Fischer and Girgensohn, 1990) discuss four characteristics of what they call end-user modifiable software. End-user modifiable software supports the following activities:

- setting parameters
- adding functionality to existing objects
- creating new objects by modifying existing objects
- defining new objects from scratch

Fischer's and Girgensohn's taxonomy of end-user modifiability is intended to systemize which kinds of changes can be supported from a user perspective.

There is a resemblance between Mørch's categorization and Fisher's and Girgensohn's. Customization correspond to 'setting parameters' and extension is comparable with 'creating new objects by modifying existing objects' (Mørch, 1995). You can say that 'adding functionality to existing objects' and 'defining new objects from scratch' is closer to the implementation code and thereby closer to the system perspective. The same issue remains; that the different characteristics do not implement both the user and the system perspective for each category.

The resemblance between the different categorizations is that they are all designed to facilitate the understanding and design of tailorable systems. The differences between them are that they are used in different stages in the development process. The four-to-five categorization is intended for the requirement phase while the other two are intended for the design phase.

Anders Mørch sees tailoring as a way of bridging the gap between presentation objects and implementation code and this is reflected in the categorization, since the three levels, customization, integration and extension, are different techniques to bridge the gap (Mørch, 1995). The categorization serves as a tool for understanding the importance of tailoring, and as guidelines for how to design tailorable software so that users get the desired functionality in the generic software. Mørch's categorization differs from the four-to-five categorization as his categorization aims more towards getting the users to understand the software whilst the four-to-five categorization is designed to help users and developers understand each other.

Fischer's and Girgensohn's categorization or taxonomy (Fischer and Girgensohn, 1990) for end-user modifiable systems focuses more on how we can achieve modifiability. They claim that their taxonomy has to be extended with illustrations of the consequences of different modifiability methods. They also say that modifications must be classified as to whether they are local or global, e.g. if the modification serves only one user or if the changes are intended for the whole user community. They also state that modifiability must be classified in terms of whether it leads to temporary or permanent changes. The intention is that the categorization should act as an instrument for developers when designing modifiable systems. The difference between Fischer's and Girgensohn's approach and the four-to-five categorization is that our categorization aims at being a tool for use earlier in the process, when users and developers negotiate requirements. The four-to-five categorization might be used in similar situations by taking into account the system view of the categories, but the developers are only given a hint of how to implement the new software. No concrete advice is given, apart from by relating the categories to a taxonomy of variability realization techniques (Irving and Eichmann, 1996, Jacobson et al., 1997, Svahnberg, 2005) which provide tangible design patterns and examples of different realization techniques that can be put into practice (Chapter Eight). However the taxonomy is not suitable for elucidating

requirements and communicating with users, since it is not designed for such use. The details and language are beyond the skills of the majority of users.

The four-to-five categorization fulfils the aim of providing a sufficient amount of clearly-defined categories to support communication in software projects dealing with tailorable software. The categorization can be used in initial discussions of what kind of flexibility the users need and how the changes have to be performed to be satisfactory from a user perspective. What is special with the four-to-five categorization is that it does not reveal any details. The definitions are at a rather conceptual level. The wording of the categories aims at raising the awareness that there are different types of tailoring and that this is reflected in the system in different ways. Since the categories are defined in general terms the discussions can be on a conceptual level that makes differences more obvious and makes it easier to focus on advantages and disadvantages without getting bogged down in unnecessary details at this time. The four-to-five categorization is designed to be a common base for communication between users and developers.

6.5 Summary

When cooperating with industry we have experienced the need to systemize tailorability in order to be able to understand and discuss the phenomenon more clearly. It is important that users and developers have a mutual understanding of what tailorability is to enable them to take informed decisions of what kind of flexibility to implement. In this paper we have made a suggestion of how to categorize tailoring in a way that may be a useful means of communication in industry. There is a big difference between the user and the system perspective, since the user perspective describes the flexible software from the point of view of which changes the user can make, whilst the system perspective focuses on what happens inside the software when a change is made. By exploring the two perspectives it was evident that confusion in discussions results from the lack of one-to-one relations between how users make a change and how the system performs the change. The investigation resulted in a new categorisation consisting of four categories of tailoring, i.e. customization, composition, expansion and extension, where we have taken into account both the user and the system perspective.

To determine the potential of the new categorization we have applied it to three research cases. It was found that the categorization was applicable to all cases and it was also found that the categorization could describe the cases without ambivalence.

We also interviewed developers and users at a telecommunication company in Sweden, in order to be able to establish that the categories were recognized in industry. We found that all respondents recognized the categories of customization, composition and expansion, while one developer and three users also recognized the fourth category, extension. The interviews also revealed that

the categorization made it possible for the respondent to pinpoint differences in the systems within the company's infrastructure. The categorization also facilitated communication of which tailorability was implemented in the different systems, which will be a useful asset in future software projects that implement tailorable software. The conclusion is that the four-to-five categorization is potentially useful and facilitates design discussions, and thereby decisions, when implementing tailorable software.

Chapter Seven

Paper VI

Chapter Seven

Characteristics of End-user Tailorable Software

The 2nd IFIP Central and East European Conference on
Software Engineering Techniques, CEE-SET 2007

Jeanette Eriksson

In a fast changing world more and more flexibility is needed in software to supply support for higher reusability and prevent the software from expiring too fast. One way to provide this kind of flexibility is end-user tailoring. A tailorable system is modified while it is being used as opposed to changed during the development process. To tailor a system is to "continuing designing in use" (Henderson and Kyng, 1991, p. 223). It is possible for the user to change a tailorable system by support of some kind of interface.

Tailorable software is needed when the environment is characterized by fast and continuous change. As Stevens and his colleagues put it "The situatedness of the use and the dynamics of the environment make it necessary to build tailorable systems. However, at the same time these facts make it so difficult to provide the right dimensions of tailorability." (Stevens *et al.*, 2006). The study presented in this paper aims for providing a tool that can support the work of finding the right dimension of tailoring when designing end-user tailorable software.

When discussing what we here call tailorability with people in industry they seldom think of or talk about this kind of software in terms of tailoring, instead they simply call it flexibility. When observing the work with tailorable software or interviewing or discussing tailorable software with people in industry it emerged that there were confusion in the discussions between users and developers when discussing flexibility. The reason is that they view flexibility from different perspectives. Flexible software is one thing when using it and a totally different thing when building the software. Accordingly, we have to look at tailoring from both system and user perspective (Stiemerling, 2000) as the user perspective reflects how users work with tailoring and the system perspective elucidates important issues from the developers' point of view.

Even between the developers themselves there were misunderstandings. It was revealed that the reason was that the perspective of the software seamlessly alters between a system and user perspective. Especially the developers make this shift without thinking of it. The reason is of cause that they have to consider both perspectives to make good software. The fact that the differences between the two perspectives are considerate and the shift in perspectives is unconscious

makes discussions about flexibility very complex. Under such circumstances it is hard to reach a consensus about what flexibility to implement and at the same time be convinced that the chosen type of flexibility is the best for the situation. To make software successful it is important that there is a consensus between users and developers of how the system must work. Users and developers must have a common understanding of the phenomenon to come to an valid agreement (Preece *et al.*, 2002). If both developers and users understand tailoring and its differences it is easier to discuss design issues and to make informed design decisions.

From an industrial perspective we end up with two issues to be dealt with:

- It is hard to know what dimensions of tailoring to implement.
- It is hard to discuss tailoring, as users and developers have different understanding of the phenomenon.

There is several aspects concerning user knowledge, technical issues and business organization that has to be fulfilled to make a tailorable system work in the long run and the tailorable software has to be supported by a collaboration between developers and users (Chapter Five). The development of tailorable software is an ongoing process where users are co-designers (Fischer, 2003) as it is users that evolve the software in use time. This kind of ongoing design can be called Meta Design (Fischer, 2003). Meta-Design is a development process where stakeholders are co-designers. Participatory Design (PD) (Schuler and Namioka, 1993) is another paradigm that includes stakeholders in the design process. PD has historically focused on involving users in the design process during design time, but the Participatory Design focus can be broaden to user design involvement during use time too (Fisher and Ostwald, 2002). Informed participation (Brown and Duguid, 2000) is related to PD as informed participation also involves others than developers in collaborate design efforts. Informed participation addresses open-ended design issues and tries to obtain an ownership of the problems among participants and to make the participants actively contribute to the design activities. The tool presented in this paper is intended as support for informed participation in a development project. Often users' participation in development projects is mainly concerned with the user interface. We agree with (Ilvari and Iivari, 2006) that the users' view of the system is not only the interface. Task related needs are what motivate end users to make changes to the system (Nardi, 1993).

As the users are co-designers human-centered design are required when designing tailorable software. The users bring profound knowledge of the business process and organizational issues into the development project, that should be made use of in the design of the technical solution (Gasson, 2003). Gasson (2003) also argue that there is a need for a dialectic process between organizational problems, implementation of changes in the business process and technical solutions to achieve a balance between human-centeredness and the

design of technical solutions. The study presented in this paper aims for providing an application of Gasson's statements in the context of tailorable software. The application, or tool, is targeted to deal with the issues of deciding what dimension of tailoring to implement, by supporting the common understanding of end-user tailoring among user and developers.

A classification is a useful tool to understand a phenomenon as tailoring. A classification of tailoring consisting of four categorises of tailoring is presented in Chapter Six. The categorization is designed to take both user and system perspective into account so that the categorization can act as a base for communication between developers and user when designing tailorable software. The categorization was found promising for use in industry. The categorization of end-user tailorable software is intended as a means of communications to involve the users more in the design process and therefore suitable as a base for a tool supporting cooperative design of end-user tailorable software.

The categorization is presented in Section 7.1. The formulation of the categories is at a rather abstract level and to make it more precise and easier to use in practice, the categories should be assigned tangible attributes or characteristics. The idea is that after pinpointing what type of business environment the software will be a part of, the skill and knowledge of the users and how much the developers are able to contribute to the tailoring process after the software has come in use, the attributes of the categories can guide you to the most appropriate type of tailoring for the specific situation.

In summary we have two research questions to answer to be able to deal with the industrial problems discussed above:

1. What attributes characterizes end-user tailorable software?
2. How can different dimensions of end-user tailoring be distinguished?

To answer the questions, a study was performed in cooperation with a major telecom company in Sweden. Both developers and users were interviewed to elucidate what attributes are relevant to describe tailoring and how they perceive different kinds of end-user tailoring.

The rest of the paper is structured as follows (Figure 7 : 1). The next section will present the categorization of tailoring that act as a base of the study. Section 7.2 describes the research method applied. In Section 7.3 the results from the study are presented. The section consists of two parts, each answering one of the research questions. The first research question result in ten attributes characterizing end-user tailorable software and the second research question result in a matrix summarizing the values of each attributes for the four different categories of tailoring. The matrix can be used as a tool to support the

cooperative design process when designing tailorable software. Furthermore the paper ends with a discussion and conclusions.

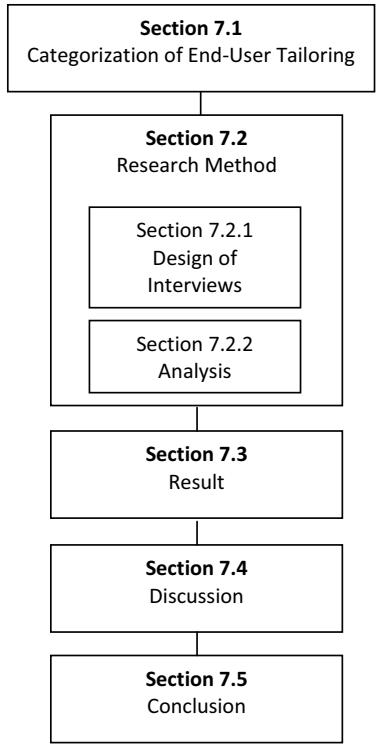


Figure 7 : 1 Overview of Chapter Seven

7.1 Categorization of End-User Tailoring

The categorization (Chapter Six) is intended as a means of communication between developers and users in situations when deciding what kind of tailorability to implement. The categorization takes into account both a user perspective and a system perspective. The user perspective represent what changes can be done or the intention with the activity, while the system perspective corresponds to how the change is achieved in the system (on a high level). The categorization is shown in Figure 7 : 1.

Customization is the simplest way of doing tailoring. It means that the user sets some values on one or more parameters and those parameters manage what functionality that is used. *Composition* means that the user has a set of components to choose from and he or she can connect them in specific ways to reach the desired functionality. *Expansion* also mean that the user chooses components out of a set, but the difference is that the users' combination of

components are build into the system to be an integrated part. The new component is treated as the predefined components and will be accessible in the set to choose from next time the software is tailored. *Expansion* is the category which provides for the highest flexibility. It means that the user writes code that are integrated into the system either by wrapping up the new code into system generated code or , if written in a predefined way, just adding it to the code mass of the software. The user can either write the code in some high level language or some visual programming language.

	User Perspective	System Perspective
Customization	Set parameter values	Interpretation of existing code
Composition	Link different existing components	Definition of relationships between components.
Expansion	Creation of a new component.	Definition of relationships between components.
		New and predefined components are treated uniformly
		Code generation (optional)
Extension	Insertion of code.	New code is added.
		Code generation (optional)

Table 7 : 1 Categorization of tailorable software

7.2 Research Method

Tailoring is especially well suited for applications used in a business environment that change very fast. The telecom business is characterized by fast changes. For example, new services continuously evolve and consequently the supporting business systems have to adapt to the altered requirements. The study was performed in cooperation with a telecom operator in Sweden. The telecom company is dependent on flexible software where the user can alter the software when needs occur. Accordingly they have a lot of tailorable systems running. The study aimed for elucidating (1) what attributes can be ascribed tailorable software and (2) how different types of tailoring can be distinguished from each other. To do so interviews were conducted and the categorization was used as a base for the interviews.

We interviewed six developers and four users at the company. The developers were programmers, system owners and technical projects leaders. The users all worked with several different systems, but their main tasks were with the same system. The users were system coordinator, work manager, users with responsibilities to work with new requirements and users helping out with further development of the system.

The interviews lasted for approximately one hour to one and a half hour. A pilot study made it clear that clarification of the questions could be needed, why we performed semi-structured interviews (Robson, 2002) which means that the same questions in the same order were asked to all the respondents, but follow-up questions were asked and explanations were given.

To be able to discuss the four categories on equal terms with both developers and users the categories was translated into four examples representing the categories (Appendix A). The examples was at a rather high level free from unnecessary details, but concrete enough to make it possible for the respondents to discuss the examples. The examples were not bounded to the tasks in the telecom company.

The interviews were audio taped and transcribed in full to provide for traceability. The transcriptions were then joined group wise. In that way it became easy to survey and compare the different opinions and reactions to the different attributes. The individual transcriptions and the analysis of the material were sent back to the respondents for verifications.

7.2.1 Design of Interviews

The researcher interviewed one respondent at a time. The developers were interviewed first and then the users were interviewed. The interviews were conducted according to a specific order. First the respondents read the examples of the different categories and thereafter they were asked if they spontaneously could assign attributes and qualities to the first example representing customization. Thereafter they had to answer some statements about the example and at the end they were asked if they could find any resemblances with the example and systems they work with or know about at the company. The procedure was the same for all four examples representing customization, composition, expansion and extension respectively.

After reading the examples and spontaneously expressed their view of the categories' characteristics the respondents had to take a standpoint to eleven attributes. The proposed attributes originate from the cooperation with the telecom company. The attributes have emerged through participant observations, discussions and interviews.

The interviews made it clear that changes can be required because of changes in the business environment, because of need of better usability or because of internal issues in the system itself. The attributes can be divided into corresponding groups. One group concerned with the category's suitability for different types of *business changes.* Another group with attributes related to *usability* and a third group involving *software attributes*. The attributes are listed below.

Business Changes

Attribute 1: Frequency of change – how often the business changes occur, often or seldom.

Attribute 2: Anticipation of change – in what extent it is possible to anticipate the business changes.

Attribute 3: Durability of change – for how long the business changes last.

Attribute 4: System support of change – how well the software support business changes

Attribute 5: Consequences if handled wrong – how extended consequences it would have for the company if the changes are handled wrongly.

Usability Issues

Attribute 6: Simplicity – how easy it is to realize the changes in the software

Attribute 7: User control – how much control the users have of what happens in the software

Attribute 8: Transparency – how easy it is for the users to know if the result is correct.

Attribute 9: Realization speed – how fast it is to realize the changes in the software.

Software Attributes

Attribute 10: Fault tolerance– to which degree the software prevents mistakes.

Attribute 11: Complexity– how complex the software is

7.2.2 Analysis

The analysis has been done in a systematical way, according to a specific, predefined schema. The materials from the interviews consist of attributes spontaneously stated, predefined attributes, comments and feedback from respondents. The four components have been considered in the analysis and constitute the result.

The analysis of the interviews consists of two parts corresponding to the two research questions respectively.

Analysis 1: Analysis to determine what attributes characterizes end-user tailorable software.

Analysis 2: The objective of Analysis 2 is to determine how the respondents perceive the different types of tailoring and put a value on each attribute to be able to distinguish different dimensions of tailoring.

Analysis 1. The first step in Analysis 1 is to compare each attribute to see if they are perceived the same for all four categories. If they are the same for all the categories they do not add any information that could be used to distinguish the categories from each other. Each attribute are compared and if they are not the same for all categories they are added to the pile of remaining attributes. If the attribute is the same for all four categories the respondents' comments are consulted to determine if the attributes really were perceived as the same. Perhaps the respondents had made a statement based on different interpretations of the proposed attributes. If the attributes are found to be the same they are removed otherwise they are added to the pile of remaining attributes. To facilitate to determine if the attributes were perceived as the same all statements were assigned a value. A positive statement of an attribute generated a score of 300 and a negative statement was assigned 100 points. Accordingly a statement in the middle generated 200 point. Initially to see if the attributes were the same for all categories, the value of the attribute were summarized. For example if all the users think that Example 1 has high fault tolerance the sum is 1200 points (4 users x 300 points) and if all the users think that Example 4 has low fault tolerance it generated totally 400 points (4 users x 100 points). The sums are compared and if they are the same they have to be examined further and each comment has to be checked.

The second step in Analysis 1 is an examination of how the respondent's answers relate to the other answers in the group. The coefficient of variance has also been used as a measure of the disagreements between respondents (Regnell *et al.*, 2000). If the respondents' view of the attributes of the examples varied a lot the attributes should be removed as it does not tell anything about the category. The remaining attributes from the first step were examined. If there is a deviation in opinions within the group the respondents' comments were checked. Based on the comments the relevance of the attributes was questioned. If the attributes was found relevant it was added to the pile of remaining statements otherwise it was removed.

In step three of Analysis 1, the respondents' spontaneously assigned attributes were listed and compared with the pre-defined attributes. If they were the same the attributes were added to the comments, otherwise they were considered as attributes of the intended category.

Analysis 2. The remaining attributes from the Analysis 1 were analysed to explore how the user group relates to the developers group per attributes. The median value for each attributes was used for guidance. If the users and developers agree upon the attributes the attributes were collected into one pile,

while if there is a deviation in opinions the respondents' comments are considered and the user specific and developer specific statements are accumulated into separate piles.

7.3 Result

When examining the totals in the first step of Analysis 1 there were some attributes that had the same total, but as the individual scores and the comments were inspected it was revealed that it was not the case. The result from the analysis is that neither of the attributes was perceived as the same for all four categories and therefore none of the attributes should be excluded at this stage.

The second step in Analysis 1 resulted in removal of three attributes (3, 5 and 6), e.g. attributes concerning durability of changes, consequences if handled wrongly and simplicity, as there were strong disagreement among the respondents. Durability of change and simplicity were regarded rather unimportant to the respondents and their answers were therefore kind of random. The consequences if the change is handled wrongly were too difficult to state as it is highly intervened by the situation.

The users thought it were difficult to spontaneously come up with attributes describing the four examples. They considered it difficult to move from the concrete example to a more abstract level. They sensed it to be easier to associate the example with a system they work with. The developer found it much easier to come up with attributes of the four examples and each developer came up with a couple of attributes each.

When comparing the developers' attributes with the pre-defined it was revealed that most of the attributes were the same. The attributes that differed from the pre-defined related to usability issues and were mentioned by several of the developers. The attributes were of two kind and concerned:

Frequency of use: how often the end users uses the software and thereby how used to the software the users are and

User competence: how skilled the users are that uses the software.

Analysis resulted thereby in ten relevant attributes that can be used to describe end-user tailorable software (see Table 7 : 2).

The result from Analysis 2 showed that the users and developers had the same perception of Example 1 (customization).

For Example 2 (composition) the users and developers had slightly different perception of user control, transparency, fault tolerance and complexity,. When it comes to user control and transparency the users judge the transparency and control to be medium high, while the developers think it is somewhat higher; somewhere between medium and high. In other words, the developers thought that Example 2 contains slightly more transparency and user control that the users. For fault tolerance and complexity there was also some small differences.

The users considered the fault tolerance and complexity for Example 2 to be medium high, but the developers though the fault tolerance should be somewhere between medium high and low and the complexity between medium high and low.. (see Table 7 : 2)

Also for Example 3 (expansion) there were some differences in views. One thing is that the developers had a united view of that Example 3 is well suited when there is a need for high support of changes, but the users are not that sure. They believe that such software provides for quite a lot of flexibility, but they are not certain that Example 3 really supports change so well that it should be stated "high support of change". There also exists a small variation in judgment of how much user control and transparency Example 3 provides for. The developers consider Example 3 to provide for medium high user control and transparency while the users believe it to be somewhere between medium high and high. But the differences in opinions in this case were very small. A more significant difference was found when it came to anticipation of change. Here the users and developers had diametrical opinions. The users thought that Example 3 was suitable for situations characterized by a high degree of anticipated changes. The developers thought to a higher degree that Example 3 was well suited for unanticipated changes too. (see Table 7 : 2)

Characteristics		Customization	Composition	Expansion	Extension
Business Changes	Frequency of change	M	M	H	H
	Anticipation of change	H	M	L-H[1]	L
	System support of change	L	M	M-H	H
Usability Issues	User control	H	M-H	M-H	?
	Transparency	H	M-H	M-H	?
	Realization speed	H	H	M	M-H
	Frequency of use	L	H	-[2]	-
	User competence	-[3]	-	M-H	H
Software Attributes	Fault tolerance	H	M-H	M	L
	Complexity	L	L- M	M	H

Table 7 : 2 Matrix of the attribute values of the four categories of end-user tailoring. (L=Low, M=Medium, H=High, ?= Uncertainty of how to use the attribute)

[1] Users thought the example was highly suitability for anticipated changes, developers thought the example was not that suitable for such situations.

[2] The spontaneously given attributes were not stated for Example 3 and 4.

[3] The spontaneously given attributes were not stated for Example 1 and 2.

The issue of user control and transparency for Example 4 (extension) resulted in some discussions of what knowledge is build into the system and what should be controlled by the user. Both users and developers agreed on that it is possible to view Example 4 as supporting either high control and transparency or low control and transparency. There is very little user control and transparency built into Example 4, but on the other hand the user handling the software should be skilled and know what he or she is doing. Thereby you could say that the software leaves the control to the users. The user control and transparency should therefore be regarded as high. The uncertainty is represented by question marks in Table 7 : .

Note that there are two pairs of attributes that show a dependency (Table 7 : 2). User control and transparency have corresponding values for all categories. When user control is perceived as high also transparency has a high value. Fault tolerance and complexity seams also related. If the fault tolerance is high the complexity is low and vice versa.

When it came to the spontaneously stated attributes, example 1 was considered suitable when there are many end users that use the software only occasionally and Example 2 was regarded as fitting when the end users are few and uses the software frequently. Example 3 and 4 was believed to be feasible when the end users are skilled and used to computer work, but Example 4 was judged to be appropriate only for a few users that are extremely skilled super users.

The matrix should be seen as a guiding tool not a tool providing the absolute truth. When designing a tailorable system the matrix could be used as a base for discussions of the needs and requirements of the specific situation. What can be expected from different types of tailorable software is listed in the matrix, but it is the participants in the project that have to make the tradeoffs between the attributes.

7.4 Discussion

The matrix is intended for design environment where the users are informed participants where users and developers claim a common ownership of the software product developed. The purpose of the matrix is to act as a base for design discussions where the users and developers discuss the requirements of the tailorable software to better understand the domain and design problems. The matrix can help the design team to pinpoint issues to discuss and to reach a consensus to be able to decide what dimensions of tailoring is needed in the given context. By consulting the matrix and comparing the values of the attributes with what is needed in a specific context, it is possible to get an indication of what kind of tailoring to implement and to be able to make informed design decisions.

There is a resemblance between assigning quality attributes to software and assigning attributes to tailoring categories. Both aim for describing a

phenomenon by assigning it characteristics. There are several software quality models, for example (Boehm et al., 1978, ISO/IEC 9126, McCall et al., 1977), and their common effort is to manage quality issues in software development. There is a resemblance between these quality models and the software attributes extracted from our study. Some of the attributes in the matrix can also be found in some quality models. But the intention with the matrix is not to give a general overview of different quality attributes. The matrix is aiming for distinguish different types of tailoring from each other and to be a tool to support design decisions when designing tailorable software. But there are some similarities, for example McCall's model is an effort to bridge the gap between the users' view and the developers' view (McCall et al., 1977). The matrix also aims for bridging the gap between users and developers by providing a means of communication, but we do not claim it to be complete as McCall's model, but the study gives us a good indication of what characteristics can be assigned the different types of tailoring.

Bosch (2000) advocates to assess the quality attributes during architectural design. The attributes are used for evaluating the architecture to determine if the architecture has to be transformed or not. The attributes in the matrix is not used for evaluation. The intended use of the matrix could be said to be a bottom up approach in comparison with Bosch's method. The four categories could be seen as a kind of "design pattern light" for tailorable software. Instead of imposing a design pattern after the architecture has failed to provide for the required quality attributes, the matrix starts out from the categories that have assigned attributes and trade offs are made. The architecture is then built based on the selected category. Another difference between Bosch's approach and ours is that Bosch presumes that it is possible to put an exact, measurable value of the quality attribute, but we only assume that the participants can grade the attributes from low to high.

7.5 Conclusion

The study made ten attributes visible of end-user tailoring. In the interviews with users and developers at a telecom company the respondents were asked to give their opinions of what characterizes four categories of end-user tailoring. Their perceptions of the categories were analysed and it was possible to process their views into a matrix representing four types of tailoring in form of attribute values. The attributes represent organizational, business and technical issues to consider and can be used in a dialectic process to balance the human-centeredness and the technical solution as Gasson (2003) requires.

The matrix can be used as guidance and base for design decisions when implementing end-user tailorable software. The attributes are at a level that can be understood by both users and developers and, as shown, the opinions of users and developers are quite similar even though differences exist. The matrix makes it possible to distinguish between different dimensions or types of

tailoring by providing values of the attributes that characterizes end-user tailorable software.

The categories and attributes of the categories together with the matrix and examples facilitate the understanding of different types of tailoring and it should make it easier for developers and users to discuss tailorability and the requirements associated to such systems.

Chapter Eight

Paper VII and VIII

Chapter Eight

Patterns in Design of End-User Tailorable Software

Usability and Design Patterns

The 7th Information Conference on Software Engineering Research and Practice in Sweden, SERPS'08 (Paper VII)

Submitted to the 10th biennial Participatory Design Conference (PDC 08). (Paper VIII)

Jeanette Eriksson

In a fast changing world more and more flexibility is needed in software to supply support for higher reusability and prevent the software from expiring too fast. "Real-world systems must change or they die" (Johnson et al., 2005). One way to provide this kind of flexibility is end-user tailoring. A tailorable system is modified while it is being used as opposed to being changed during the development process. To tailor a system is "continuing designing in use" (Henderson and Kyng, 1991, p. 223) It is possible for the user to change a tailorable system with the support of some kind of interface.

Tailorable software is needed when the environment is characterized by fast and continuous change. As Stevens and his colleagues put it "The situatedness of the use and the dynamics of the environment make it necessary to build tailorable systems. However, at the same time these facts make it so difficult to provide the right dimensions of tailorability." (Stevens, et al., 2006, p.273). This paper is aimed at providing support for the process of designing end-user tailorable software through introducing patterns as a mediating artefact between users and developers.

The development of tailorable software is an ongoing process where users are co-designers (Fischer, 2003), since it is users who evolve the software at use time. The absence of end-user participation can result in low acceptance of the software (Schümmer and Slagter, 2004), and in end-user tailoring, user acceptance is especially important since it is the users that carry out the intention with the software, to be evolved. We agree with (Ilvari and Iivari, 2006) that the users' view of the system is not only concerned with the interface. Task related needs are what motivate end users to make changes to the system (Nardi, 1993).

Since users are co-designers, human-centered design is required when designing tailorable software. The users bring profound knowledge of the business process and organizational issues into the development project, which should be made use of in the design of the technical solution (Gasson, 2003). But it is difficult to actively involve the end-users in the development process (Schümmer and Slagter, 2004). This is confirmed by our own interviews with users and

developers in a Swedish telecom company. Both users and developers express a desire and an interest in achieving an environment where users and developers take an active part and equal responsibility for the software developed, but they also agree that this is difficult to achieve. A precondition to make such a cooperative process work is that users and developers share the same language (Schümmer et al., 2005). Or in other words they share a base of mutual understanding of the phenomenon.

A classification can be a useful tool to understand a phenomenon such as tailoring. A classification of tailoring consisting of four categories of tailoring is presented in Chapter Six. The categorization is designed to take both the user and the system perspective into account so that the categorization can act as a base for communication between developers and the users when designing tailorable software. The categorization is intended as a means of communications to involve the users more in the design process and was found promising for use in industry. The categorization is briefly presented in Section 8.1.

Another obstacle to overcome is the transfer of knowledge of technical issues from developers to users. This is a difficult matter, but patterns have been found to be a useful instrument (Lukosch and Schümmer, 2006, Schümmer et al., 2005, Schümmer and Slagter, 2004) for knowledge transfer. Patterns facilitate understanding and communication, increase confidence in decisions, make it easier to consider different solutions and provide for control (Buschmann et al., 2007).

What is required to enable the use of a pattern approach in end-user tailoring design is a selection of suitable patterns. To be able to narrow down the number of patterns to consider for each type of tailoring, this selection of patterns should be connected to the categorization of tailoring. Since we believe that end-user participation in the design process is essential to gain quality in end-user tailorable software, it is important to neutralize possible obstacles. Especially for beginners it is hard if there are too many patterns to consider (Gamma et al., 1995).

There are two ways to introduce patterns in the cooperative design process, either by starting with architectural design patterns that transfer good practice when it comes to software design or patterns that expresses design issues of human interactions (usability patterns). The content of usability patterns is closely related to the task and to the users' domain, and usability patterns may provide a gentle slope towards patterns for software architectures. Usability patterns do not only deal with issues that are put on top of the basic software architecture. In fact separation of concern is not enough to achieve usability (John et al., 2004). Usability features that are recognized late in the design process are often expensive to attend to. Usability issues obviously have architectural impact beyond the detailed design of graphical interfaces and several usability scenarios are identified to influence software architecture (Bass and John, 2003). This chapter focuses on usability patterns with architectural

impact, which are of vital importance to end-user tailoring together with a subset of software design patterns suitable to start with when developing end-user tailoring.

To summarise, there are two objectives of this chapter:

1. To compile usability and design pattern collections to small subsets by relating the patterns to the categorization of end-user tailoring.
2. To determine what a pattern should consist of to be supportive in the cooperative design process involving both user and developers.

The result is a classification of patterns that can act as a mediating object between users and developers, as well as a concrete base for the technical solution when designing end-user tailorable software.

In the end-user tailoring community patterns are infrequently discussed. It is likely that the researchers and practitioners within the area of end-user tailoring use patterns, but there is no explicit discussion of the topic in the research community. We therefore argue that there is interesting to classify patterns suitable for end-user tailorable software, not only from an industrial perspective but also from an academic point of view. We do not claim that the collection of patterns presented in this article is exhaustive. Indeed, we hope that the collection will be extended with more dedicated patterns.

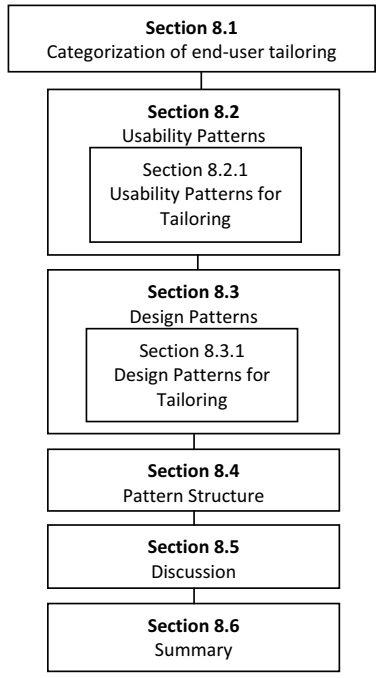


Figure 8 : 1 Overview of Chapter Eight

The rest of the paper is structured as follows (Figure 8 : 1). The next section will present the categorization of tailoring. Section 8.2 describes two approaches to identifying the usability patterns that have to be introduced early in the development. In Section 8.2.1 the usability patterns of vital importance to end-user tailoring are explored and related to the categories of end-user tailoring. In the next section, Section 8.3 design pattern introduced and in Section 8.3.1 design patterns are explored to find small subsets of design patterns to use in the cooperative design process of end-user tailoring. Section 8.4 contains a description of how the patterns for end-user tailoring can be presented, in form of a pattern template. Thereafter follows a discussion of how the results relate to other work and how the results can be used. The paper ends with a summary of the results.

8.1. Categorization of End-User Tailoring

The categorization proposed in Chapter Six is intended as a means of communication between developers and users in situations when deciding which kind of tailorability to implement. The categorization takes into account both a user perspective and a system perspective. The user perspective represents which changes can be made, or the intention of the activity, while the system perspective corresponds to how the change is achieved in the system (on a high level). The categorization is shown in Table 8 : 1.

	User Perspective	System Perspective
Customization	Set parameter values	Interpretation of existing code
Composition	Link different existing components	Definition of relationships between components.
Expansion	Creation of a new component.	Definition of relationships between components.
		New and predefined components are treated uniformly
		Code generation (optional)
Extension	Insertion of code.	New code is added.
		Code generation (optional)

Table 8 : 1 Categorization of tailorable software

Customization is the simplest way of doing tailoring. It means that the user sets some values on one or more parameters and those parameters manage what functionality is used. *Composition* means that the user has a set of components to choose from and he or she can connect them in specific ways to achieve the desired functionality. *Expansion* also means that the user chooses components out of a set, but the difference is that the users' combination of components is build into the system as an integrated part. The new component is treated in the same way as the predefined components and will be accessible in the set to

choose from next time the software is tailored. *Extension* is the category which provides for the highest flexibility. It means that the user writes code that is integrated into the system, either by wrapping up the new code in system generated code or, if written in a predefined way, just adding it to the code mass of the software. The user can either write the code in a high level language or a visual programming language.

This categorization can be used as a gateway leading to which patterns to consider. By defining both a user and a system perspective, the intention is to make it easier to discuss tailoring in a consistent way.

The next section will discuss what usability patterns to use in tailorable software and later on we will also discuss software design patterns.

8.2 Usability Patterns

Usability patterns or HCI (Human Computer Interaction) design patterns are useful tools when designing user interfaces (Wesson and Cowley, 2003). A number of different collections of patterns exists, for example a comprehensive pattern language for user interfaces by Tidwell[1] (Tidwell, 2006).Traditionally HCI (interface design) and software architectures have been kept separate by the notion of separation of concerns, but separation of concerns is not suitable if we want to design software with good usability, acceptable to users. Usability issues discovered late in the process can be expensive to recover (John et al., 2004) which indicates that usability issues have an impact at an architectural level of software design. There are two recent approaches (presented below) that deal with usability issues that should be considered early in the design process.

Based on experience, Bass and John (2003) have identified 27 usability scenarios that must be considered during the architectural design. For each scenario they created an architectural pattern as a solution to the scenario. The 27 scenarios are in short:

1. Aggregating data
2. Aggregating commands
3. Cancelling commands
4. Using applications concurrently
5. Checking for correctness
6. Maintaining device independence
7. Evaluating the system
8. Recovering from failure

[1] See also http://www.mit.edu/~jtidwell/common_ground.html and http://designinginterfaces.com/, accessed September 13, 2007

9. Retrieving forgotten passwords
10. Providing good help
11. Reusing information
12. Supporting international use
13. Leveraging human knowledge
14. Modifying interfaces
15. Supporting multiple activities
16. Navigating within a single view
17. Observing system state
18. Working at the users' pace
19. Predicting task duration
20. Supporting comprehensive searching
21. Supporting undo
22. Working in an unfamiliar context
23. Verifying recourses
24. Operating consistently across views
25. Making views accessible
26. Supporting visualization
27. Supporting personalization

A similar attempt to introduce usability aspects early in the development process was done within a European Union project (STATUS) (Ferre et al., 2003, Folmer and Bosch, 2003., Juristo et al., 2003). But compared to Bass and John they started from a different angle. The STATUS project started out with a set of usability attributes (satisfaction, learnability, efficiency and reliability) and then mapped the attributes to usability properties that in their turn were related to usability patterns. A usability property is specified in terms of the solution space and can be regarded as usability requirements expressed in a more concrete form. For example the quality attribute *efficiency* has a relation to the usability property *error prevention,* since error prevention has a positive effect on efficiency. Error prevention in turn has a relation to, for example, the usability patterns *form or field validation* and *workflow model* (Juristo et al., 2003) as the patterns fulfil the requirement.

The results from the two approaches overlap and consist of a set of usability pattern that have an impact on software architecture and thereby must be considered early in the development process. The relationship between the usability patterns from the STATUS project and the general usability scenarios provided by Bass and John is presented in (Juristo et al., 2003).

8.2.1 Usability Patterns for Tailoring

Our goal is to match the categories of end-user tailoring to a set of usability patterns that are especially important to provide for user satisfaction and confidence in the tailoring process. To achieve this we have made use of both approaches above.

We relate usability patterns to the categorization of end-user tailoring in three steps:

Step 1: We start the exploration from empirical results from our cooperation with a telecom operator in Sweden concerning *usability issues* essential to consider in the tailoring interface of the software.

Step 2: We match *usability issues* with *usability properties* (Ferre et al., 2003, Folmer and Bosch, 2003, Juristo et al., 2003) and *usability scenarios* (Bass and John, 2003).

Step 3: We match *usability scenarios* (Bass and John, 2003) with the categories of end user tailoring. The categories will automatically be related to usability patterns since Juristo et al. (2003) already have matched usability scenarios with usability patterns.

We start by discussing the usability issues in Step 1.

Step 1

During a project performed in corporation with our industrial partner, a major telecom operator, we explored how end-users could manage system infrastructure. We built a prototype that was evaluated by users and developers by "talking aloud" when using the prototype. In the same project we explored which technical issues are most important to consider in order to make end-user tailoring work. Four usability issues or overall requirements were revealed concerning the tailoring interface (Chapter Five):

1. Functionality for controlling and testing
2. Clear split between definition, execution and the tailoring process.
3. Unanticipated use revealed to the tailor.
4. Complexity

Functionality for controlling and testing is self-explanatory. It is essential that the user can control the tailoring process and test the changes. It was also important for the users to have a clear split between use and tailoring. One reason for this was that it was easier to focus on one abstraction level at a time. Another reason was that a clear split makes it possible to assign different people to the different tasks. In other words it is easier to separate the role 'tailor' from the role 'user' and thereby delegate the tailoring process to a few people. It was also evident that it was important that the different possibilities to change the software were revealed to the tailors even though it might not be what the designers had in mind when designing the tailoring feature. The software should be prepared for creative use. The last issue concerning complexity is somewhat connected to unanticipated use and it was shown that the users preferred a more complex tailoring interface with superfluous information in favour of just in

time information to minimize cognitive load, which is advocated as a pattern to support usability. The motivation was that a tailoring activity is not performed on a regular basis and is therefore allowed to take time. It is therefore preferable to have a complex interface that allows creative use. But to compensate, a complex tailoring interface requires a very simple user interface. As the complexity issue is the opposite of what is recommended in usability literature, we will not discuss complexity further. We do not need a pattern to decrease the complexity. However, there are patterns to handle complex data in user interfaces[2] (Tidwell, 2006).

Step 2

The second step towards a match between usability patterns and the tailoring categories is to match the usability issues presented above (unanticipated use revealed to the tailor, explicit user control, error correction and error prevention) with usability properties. The usability issues are requirements for end-user tailorable software and correspond well to usability properties, as the properties are also a form of requirements. Then the usability issues are mapped to the general usability scenarios. For example, if an end-user tailorable system provides for *unanticipated use revealed to the tailor* it also has to provide for the usability properties *explicit user control, error correction and error prevention* (Folmer and Bosch, 2003.). Then we examine the general usability scenarios. If you fulfil the requirement for error prevention it is easier to work in an unfamiliar context. Likewise to fulfil the requirement for guidance you have to provide for good help. The summary of the correspondences is shown in Table 8 : 2.

Usability issue	Usability property (Folmer and Bosch, 2003.)	Usability scenario (Bass and John, 2003)
Functionality for controlling and testing	Explicit user control Error management • Error correction • Error prevention	Checking for correctness Observing system state Supporting undo Working in an unfamiliar context Verifying resources
Clear split between definition, execution and the tailoring process.	Adaptability • Matching user preferences • Matching user expertise	*(no match to usability scenarios but the usability pattern "User profile" will satisfy the requirement)*
Unanticipated use revealed to the tailor.	Guidance Provide feedback	Providing good help

Table 8 : 2 Relations between usability issues and properties.

[2] http://www.mit.edu/~jtidwell/common_ground.html and http://designinginterfaces.com/, accessed September 13, 2007

Step 3

Table 8 : 2 results in a subset of scenarios that are of vital importance to end-user tailoring. Step 3 means matching the categories of end-user tailoring with usability patterns. The match is presented in Table 8 : 3 and explained below.

Category	Usability Scenario	Pattern (Juristo et al., 2003)
Customization	***Checking for correctness***	***Form/Field validation***
	Supporting undo	***Undo***
	Providing good help	***Wizard, Context-sensitive help, Standard Help, Tour***
		User profile
Composition	Checking for correctness	Form/Field validation
	Supporting undo	Undo
	Providing good help	Wizard, Context-sensitive help,
		Standard Help, Tour
		User profile
	Working in an unfamiliar context	***Workflow model***
Expansion	Checking for correctness	Form/Field validation
	Supporting undo	Undo
	Providing good help	Wizard, Context-sensitive help,
		Standard Help, Tour
		User profile
	Working in an unfamiliar context	Workflow model
	Observing system state	***Status indication***
Extension	Checking for correctness	Form/Field validation
	Supporting undo	Undo
	Providing good help	Wizard, Context-sensitive help,
		Standard Help, Tour
		User profile
	Working in an unfamiliar context	Workflow model
	Observing system state	Status indication
	Verifying resources	***Alert***

Table 8 : 3 Tailoring categories and corresponding scenarios and pattern.

Scenarios corresponds to activities and so do the categories of tailoring, therefore we match the subset of scenarios to the categories. It is therefore easy to imagine which scenarios should be relevant for the different categories. For example, independently of which kind of tailoring activity you perform you would like to be able to check for correctness, support of undo and good help. But if you do a composition, combining different component with each other, it involves doing things you are not doing on a regular basis. What you are doing is equivalent to the scenario of *working in an unfamiliar context*. The relationships between the categories reveal themselves automatically by matching the scenarios with usability patterns, according to (Juristo et al., 2003) (Table 8 : 3).

The result is a selection of usability patterns that have an architectural impact. By choosing a type of tailoring to implement we are given some examples of usability patterns we should consider using. We do not claim that the selection is complete. Actually there may be other usability patterns that match the scenarios and should be considered for use. Note that we have made a selection of usability scenarios that we state are of vital importance; we do not thereby say that the rest are unimportant for end-user tailoring. On the contrary, those scenarios with corresponding usability patterns are as important to tailorable software as to any other software concerned with user interaction. The rest of the scenarios can be used as a checklist to determine if important usability issues have been considered during architectural design. What we say is that the selected scenarios are not negotiable if the end-user tailorable software is to be a success. For example, providing for good help is not negotiable and one of the patterns “Wizard”, “Context-sensitive help”, “Standard Help” or “Tour” should therefore be considered.

In the next section we will discuss how to select a collection of software design patterns to consider when building tailoring capabilities into software

8.3 Design Patterns

Gamma et al. defines a design patterns as “...descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.” (Gamma et al. 1995, p. 3). Patterns catches previous successful experiences (Gamma et al., 1995) and can guide practitioners to build good software without standardizing the solution. Or as, Christopher Alexander et al. put it: “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice” (Alexander et al., 1977, preface p. x).

There are three main concepts in architectural design: architectural style, architectural pattern and design pattern. An architectural style is predominant, while an architectural pattern can be merged with an architectural style and it affects the whole architecture (Bosch, 2000). Design patterns on the other hand are local. In our exploration of how to select patterns that can support the design

of end-user tailorable software we have chosen not to make any distinction between the different concepts. We will use the unifying term *design pattern.* In the process of building the software it may be important, but at this stage we leave it to the design team to decide what should be used.

There is a lot of collection of software design patterns but the most referred books about design patterns are those by Gamma et al. (1995) and Buschmann et al. (Buschmann et al., 2007, Buschmann et al., 1996). In Table 8 : 4 all patterns from Gamma et al. are listed together with a short description of the patterns.

	Design Pattern	Description (Gamma et al., 1995)
Creational Patterns	Abstract Factory	Families of product can vary by not specifying their concrete classes
	Builder	The same construction process can create different representations of complex composite objects.
	Factory Method	Defers instances to subclasses although defining an interface for creating an object.
	Prototype	Creates new objects by copying a prototypical instance.
	Singleton	The number of objects only existing
Structural Patterns	Adapter	By changing the interface of a class into another interface to make components work together.
	Bridge	By separating the abstraction from its implementation they can vary independent of each other.
	Composite	Makes it possible for clients to treat individual objects and compositions of objects the same way.
	Decorator	Without subclassing dynamically attach additional responsibilities to an object.
	Façade	Defines a unified interface to other interfaces in a subsystem.
	Flyweight	Used for efficient object storage.
	Proxy	Used to supply a surrogate for other object.
Behavioural Patterns	Chain of responsibility	Delegates the request to a chain of object that may handle the request. Any of the object can choose to handle the request.
	Command	Possible to parameterize clients with different requests by encapsulate the requests as objects.
	Interpreter	Defines a representation for a language's grammar and an interpreter interpret sentences.
	Iterator	Used to traverse a collection of objects without exposing the underlying representation.
	Mediator	Defines how and which objects interact with each other.
	Memento	Defines what and when private information is stored outside an object.
	Observer	Define a one-to-many dependency between objects so that the objects stay up to date
	State	Makes it possible for an object to change behaviour when the internal state changes.
	Strategy	Makes it possible for algorithms to vary independently from clients
	Template Method	Makes it possible to redefine steps of an algorithm in subclasses without changing the structure of the algorithm.
	Visitor	Make it possible to define a new operation without changing the classes on which it operates.

Table 8 : 4 Design Patterns from Gamma et al. (1995)

We have chosen to use Gamma et al.'s pattern collection since the patterns do not form a pattern language which suits our purposes to provide for a gentle slope into learning about patterns. Gamma et al. classifies the patterns in creational, structural and behavioural patterns. Creational patterns have to do with creating objects, structural patterns concern the organization of classes and objects and behavioural patterns deals with how objects interact.

8.3.1 Design Patterns for Tailoring

If we remove the end-user tailoring part of end-user tailorable software we end up with an adaptable system or rephrased, a system embracing software variability. In the process of selecting a subset of design patterns to introduce to users, we can make use of classification in the area of software variability and adaptability.

We relate design patterns to the categorization of end-user tailoring in three steps:

Step 1: We start by relating the categories of end-user tailoring to variability realization mechanisms and hotspots. This is done by exploring the *meaning of change* in relation to the categories and the *type of change* provided for by the mechanisms.

Step 2: We continue to match the *type of change* to different *design patterns*. In this way we get a relationship between the mechanisms and the patterns.

Step 3: This step means that the categories are related to a set of design patterns via *type of change*.

We start by discussing variability realization mechanisms and hotspots in Step 1.

Step 1

Svahnberg et al. (2005) have an approach to classifying variability. They provide a taxonomy for variability realization techniques. The authors differentiate different types of variability among other things by how the variation point is populated (explicit or implicit) and how the binding of the variant should be done (internal or external). If a variation point is populated explicitly the set of variants is manage within the system, while implicitly populated variation points are managed by an application engineer outside the system. Internal binding means that the system contains the functionality to bind the variants while external binding require a person or a tool external to the system to perform the binding.

The notion of explicit and implicit population and external and internal binding is interesting to end-user tailoring to be able to determine what realization mechanism that are suitable for implementing end-user tailorable features. As the intention with tailorable software is to provide a set of possibilities to change the software the set of variants is managed within the system that is; the population is *explicit*. The binding is also preferably handled by the software itself which means that the binding is *internal*. Additionally the binding time is

in *runtime* for end-user tailorable software. When consulting the taxonomy containing 16 different realization mechanism (Svahnberg et al., 2005) the subset of realization techniques for end-user tailorable software is narrowed down to four, namely:

- Runtime variant component specialization (a number of alternative executions) – that is: *Choosing* specialisation within a component.
- Condition on variable (functionality to change variable) – that is: *Choosing* between different operations
- Variant component implementation (dynamically determine what component to use) – that is: *Choosing* components
- Infrastructure-centered architecture (the components are first class entities connected by connectors) – that is: *Providing for* an interface

Irwing and Eichmann (1996) have another approach and they define four different types of adaptability or hot spots for adaptable software:

- Composition (instances can be composed to greater whole) – that is: *Creating* a new component by connecting several components
- Semantics (the semantics of a class is changed by for example sub classing) – that is: *Creating* a new component by subclassing
- Type compatibility (the interfaces between the classes is changed), protocol (the protocol between the classes is changed) – that is: *Providing for* an interface

Type compatibility and protocol are tightly related and as the most common programming languages today are typed it is not relevant to distinguish between type compatibility and protocol change. Accordingly we have three types of hot spots.

Svahnberg's et al. (2005) approach (the four realization mechanism discussed above) deals with how to achieve variability in the software or rephrased how to facilitate flexibility by *choosing appropriate functionality*. In terms of hotspots Svahnberg's et al. approach is about providing for the hotspots while Irving's and Eichmann's approach is about how to *create new functionality* in specified places, the hotspots.

Changing an application by end-user tailoring means choosing among predefined entities or adding new entities. Of course it is possible to allow end-users to change an entity, but it is seldom the case, as functionality is lost when making a change to an existing component. If new functionality is required that are similar to existing it is better to make a new entity that embrace old functionality together with some new features.

If we look closer at the categorization of tailoring (Section 8.1) we can see that two of the categories, namely customization and composition, comprise choosing among predefined options. Customization means that the user *chooses* what shall happen in the application by setting parameters and the parameter

determine what happens in the application. Composition means that the user *chooses* components and relates them to each other.

On the other hand, the two other categories, expansion and extension lead to that new entities are created. When it comes to expansion the tailoring activity bring about a new component that can be incorporated into the application. Indeed the user relates different component, but the result is that a new component is *created.* Also extension means that a new component is added as the user, in some way (through a graphical user interface or by coding), add new code. A new component can be created from scratch or by adding some lines of code to a general shell. Additionally it is likely that there is a need for an interface between the created components and the application.

The variability realization mechanisms and hotspots presented above also focus on making changes by choosing between a set of options and creating new components respectively. Both approaches also consider interface change (Infrastructure-centered architecture and type compatibility/protocol) that is, two concepts representing two sides of the same coin.

The meaning of the change (choosing or creating a component or adding an interface) can be used to match the categories of tailoring to the different variability realization mechanisms and hot spots (Figure 8 : 2). The relationship is shown in Table 8 : 5.

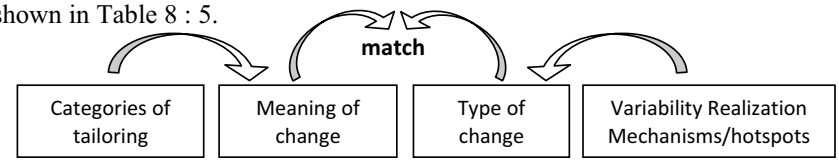


Figure 8 : 2 Matching categories and variability realization mechanisms/hotspots

Meaning of Change	Categorization of tailoring	Type of change	Variability realization techniques / hotspots
Changing by choosing	Customization	1.*Choosing* specialisation within a component	Runtime variant component specialization
		2.*Choosing* between different operations	Condition on variable
	Composition	3.*Choosing* components	Variant component implementation
Changing by creating new components	Expansion	4.*Creating* a new component by connecting several components	Composition
	Extension	5.*Creating* a new component by subclassing	Semantics
Changing by adding a new interface/ connector.	*Expansion/ Extension*	*6. Providing for an interface (optional)*	*Infrastructure-centered architecture*
			Type compatibility/protocol

Table 8 : 5 Change in relation to the categorization of end-user tailoring, variability realization techniques and hotspots

Step 2

What we want to achieve is a match between the end-user categories and design patterns. The variability realization techniques/hotspots have to be matched to design patterns to further on match the design patterns with categories of end-user tailoring.

We limit the exploration to design patterns presented by Gamma et al. We believe this limited scope is a good start as Gamma et al.'s patterns are well known and widely used. The different patterns are investigated in terms of type of change the patterns supply (Figure 8 : 3) (Table 8 : 6).

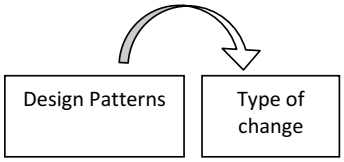


Figure 8 :3. Matching design patterns

Design Pattern	Type of change
Strategy	1.*Choosing* specialisation within a component
Template Method	
Command	2.*Choosing* between different operations
State	
Chain of Responsibilities	3.*Choosing* components
Mediator	
Decorator	
Composite	4.*Creating* a new component by connecting several components
Builder	
Abstract Factory	5.*Creating* a new component by subclassing
Prototype	
Adapter	*6. Providing for an interface*
Bridge	
Facade	
Proxy	

Table 8 : 6 Matching patterns with type of change

Step 3

As well as the meaning of change can guide us to what kind of tailoring the type of change is representing it can also lead us to what design patterns that can support the different categories of tailoring.

By mapping type of change from design pattern to categories of tailoring we get a subset of design patterns that are related to the different categories of tailoring (Figure 8 :4)

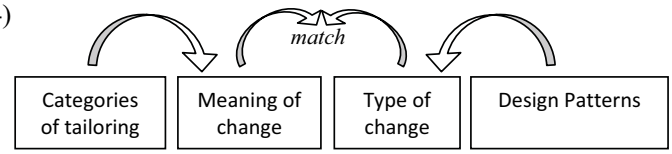


Figure 8 :4. Matching categories to patterns

Eight patterns were found having very little appliance to features for end-user tailoring. The patterns were Factory Method, Singleton, Flyweight, Interpreter, Iterator, Memento, Observer and Visitor. Of cause these patterns can be combined with other patterns and used for flexible software as shown by Hummes and Merialdo (2000), but the patterns do not provide for tailorability by themselves. The rest of the patterns are matched with the tailoring categories in Table 8 : 7.

Pattern type	Design Pattern	Type of change	Tailoring category
Behavioural patterns	Strategy	1	customization
	Template Method	1,2	
	Command	2	
	State	2	
	Chain of Responsibilities	3	composition
	Mediator	3	
Structural patterns	Decorator	3	
	Composite	4	expansion
Creational patterns	Builder	4	
	Abstract Factory	5	extension
	Prototype	5	
Structural patterns	Adapter	6	expansion and extension
	Bridge	6	
	Facade	6	
	Proxy	6	

Table 8 : 7 Design patterns matched tailoring categories (the numbers refer to the numbers in Table 8 . 6)

The selection is by no means complete. We have chosen to categorize some of the most well known If a development team come up with a pattern they experience being relevant for end-user tailoring the pattern can be categorized as we have done here. The categorization narrow down the number of patterns that is important to consider in the design process. And as we consider users

being important participants in the design process it is vital that the numbers of patterns is kept on a reasonable level to avoid overwhelming the participants. At the same time, time is saved as the categories can guide the team to consider specific patterns without go through a lot of irrelevant patterns. The pattern collection is intended as a start to learn about and use patterns and the pattern collection should be customized to suit the specific situation.

We have related the categorization of end-user tailoring to a subset of usability patterns as well as a subset of design patterns. In the next section we will discuss what a pattern (both usability and design pattern) should consist of.

8.4 Pattern Structure

It is important that the different patterns are not too comprehensive. One objective of the patterns is that both users and developers should get an overview of the different design possibilities. To make the patterns easy for the end-users to understand, it is essential that they are written in a more prosaic style than if the patterns are solely intended for use by developers (Schümmer and Slagter, 2004). The patterns should provide the participants with an understanding of the pattern almost *at a glance*, whilst at the same time it is essential that the patterns provide the participants, both users and developers, with enough information to be able to transform the pattern into the software architecture without having to re-invent the wheel. In other words the patterns should not only be a base for discussion but should at the same time be an effective instrument for the developers.

There are many different pattern forms (Buschmann et al., 2007). We have chosen to compare four different approaches, to evaluate the suitability of using one of the approaches for the patterns intended for end-user tailoring and to determine if we should compile our own pattern template. The four approaches are chosen because they fulfil at least one of the requirements for a pattern template for end-user tailoring. Borchers's pattern structure (2001) is uniform and supports application domain patterns, HCI patterns and software patterns. Schümmer et al. (Lukosch and Schümmer, 2006, Schümmer and Slagter, 2004) supports both users and developers and is constructed as a means of communication, which is exactly what we also want to do. John et al. (2004) explicitly manifests the importance of considering different types of forces influencing the design, which we consider important, and the last approach is Gamma et al. (1995) which is the most widely known pattern collection. This collection is written for developers and since an end-user tailoring pattern should also be useful and effective for developers when implementing the software, it is relevant to compare the other approaches to this.

Borchers (2001) extends the notion of pattern languages to Human-Computer Interaction, since patterns is a suitable instrument to capture experiences of user interface design. Borchers also extends the pattern language approach to the area of the application domain, and has worked a lot with interactive exhibitions in, for example, music. Borchers has constructed an interdisciplinary pattern

language framework to be able to collect design experiences from both HCI, software engineering and the application domain. The pattern structure is uniform and is intended to be suitable for all three areas. Table 8 : 8, left column, lists the different subsections in the pattern structure.

Schümmer and colleagues (Lukosch and Schümmer, 2006, Schümmer and Slagter, 2004) outline a pattern structure of design patterns that are constructed to meet both users' and developers' requirements for detailed description and visualization. This structure was tried out in two projects and found useful in the context of educational groupware. The patterns acted as metaphors and made it possible for the participants to talk about the software system and also helped the participants to focus on one feature at a time (Schümmer and Slagter, 2004). The pattern structure is used for a pattern language and is constructed to facilitate communication and learning. The pattern template consists of three main sections. The first section is to help decide if the patterns seem to fit the situations, the second section contains solutions and the final part presents the solution in more detail. Table 8 : 8, second column, lists the different subsections in the pattern structure.

Most patterns, both design and usability patterns, are constructed so that the pattern should be independent of external forces (John et al., 2004) (e.g. not influenced by, for example prior design decisions), but John et al. (2004) have constructed a structure for usability-supporting patterns that have a section dedicated to a 'Specific Solution'. John et al. have identified different types of forces that influence the implementation of the patterns and have incorporated them in their usability-supporting patterns. This makes the pattern dependent on the actual situation it would be used in. The forces identified are:

- Forces exerted by the environment and the task
- Forces exerted by human desires and capabilities
- Forces exerted by the state of the software
- Forces that come from prior design decisions

These identified forces correspond well to our own experiences from prolonged observations of a project developing an end-user tailorable subsystem to one of the telecom operator's business systems. Also Buschmann et al. (2007) claim that forces are the heart of every pattern. Table 8 : 8, third column, lists the different subsections in the pattern structure.

The fourth column in Table 8 : 8 lists the structure of Gamma et al.'s patterns (1995) This approach is well known amongst developers and it is also developers that are the target group for the patterns. The patterns "help designers reuse successful designs by basing new designs on prior experience." (Gamma et al., 1995, p. 1). The patterns structure consists of not only graphical diagrams but also relationships between classes and objects, alternative solutions and trade-offs. Examples are also important as it shows how the pattern can be applied.

Pattern for an interdisciplinary framework	Pattern for user participation	Usability-supporting pattern	Pattern by Gamma et al.
Pattern Name			
• Name	• Pattern name	• Name	• Pattern name
Problem			
• Ranking			• Also known as
• Illustration	• Intent		• Intent
• Context (relates to other patterns) and references	• Context	Usability Context • Situation: (usefulness from the end-users' perspective) • Conditions on the Situation (conditions of usefulness) • Potential Usability Benefits	
• Problems and forces	• Problem	Problem • Forces exerted by the environment and the task • Forces exerted by human desires and capabilities • Forces exerted by the state of the software	
• Illustration	• Scenario		• Motivation
• Illustration	• Symptoms (identify the need)		• Applicability
Solution			
• Solution	• Solution	General solution: • Responsibilities of the general solution that resolve the forces	
		Specific solution • Forces that come from prior design decisions • Allocation of responsibilities to specific components • Rationale gives reason for how the responsibilities have been assigned to the components.	
• Diagram		• Component diagram of specific solution • Sequence diagram of specific solution • Deployment diagram of specific solution	• Structure
	• Collaboration		• Participants • Collaborations
Consequences			
	• Rationale		• Consequences
	• Danger spots (the rise of new unbalanced forces)		• Implementation
Extras			
			• Sample code
• Examples	• Known uses		• Known uses
• Context (above)	• Related patterns		• Related patterns

Table 8 : 8 Comparison of four different pattern structures

The question is which of the approaches is most suitable for a pattern for end-user tailoring. We must list the requirements for a pattern for end-user tailoring:

- The pattern structure should also be practicable for both usability and software design patterns.
- The patterns should start generally and gradually be more detailed to facilitate learning.
- The patterns should be easy to overview, grasp and understand.

The pattern structure should be an effective instrument for both users and developers, together and individually.

If we compare how well the different approaches comply with the requirements (Table 8 : 9) we can see that Borchers's and Schümmer's et al. approaches are equally favourable. Borchers's pattern structure is better than Schümmer's et al. when it comes to how practical it is for software design patterns, but this is compensated for by the fact that Borchers's patterns are less detailed. It is easy to take care of the lack of details by adopting the parts from John's and Bass' approach, where the different forces are described in detail. John and Bass also recommend diagrams on a detailed level.

Requirement	Pattern for an interdisciplinary framework	Pattern for user participation	Usability-supporting pattern	Pattern by Gamma et al.
Practical for design patterns	+	-	++	++
Gradually more detailed	-	+	++	+
Easy to overview and understand	++	+	--	--
Instrument for both developers and users	+	+	-- user ++developer	-- user ++developer

Table 8 : 9 Compliance of requirements
(Legend: ++ = very good, + = good, - = not that good, -- = bad)

It seems to be a good idea to begin with Borchers's pattern structure and fill in with good features from the other approaches. Borchers's patterns start out in a general way and there are few headings, which makes it easier to grasp and overview. The headings are general and easy to understand. The details should not appear until later on, in the solution part. The solution should first be introduced generally and then become more detailed. This is attended to by adding the sections *general solution* and *specific solution* from John and Bass's pattern structure. But compared to patterns for user participation and Gamma's et al. pattern there are more details that should be added to better support the developers. These are: consequences, danger spots, sample code and related

patterns. In Borchers's approach, related patterns are incorporated in the context section. We however find it better to explicitly point out the related patterns, in favour of ease of use.

The resulting pattern structure (Table 8 : 10) is intended solely for end-user tailorable software and the tailoring categories act as a gateway to the patterns, therefore it is of course important to relate each pattern to the type of tailorability it is suitable for.

Design Pattern for End-user tailorable software		
Introductory description		
• Name		
• Ranking		*The author's confidence in the pattern*
• Tailoring Categories		*Which categories of tailoring the pattern is suitable for*
• Illustration		
Overall description of problem and solution		
• Problem		
• Forces	• Environment and task	*Forces from environment and task that influence the choice of solution.*
	• Human desires and capabilities	*Forces from human desires and capabilities that have an impact on the choice of solution.*
	• State of the software	*Forces generated by the system state, for example software is sometimes unresponsive (John et al., 2004)*
• General Solution		
Detailed description of solution		
• Specific Solution		*Example of prior design decisions that influence the choice of solution. The forces are specific for the situation.*
	• Prior design decisions	
• Diagrams		
• Consequences		
• Danger spots		
• Sample code		A short example of how to implement the pattern. Written in the language used at the company or in C++ since this is well known.
• Examples		Examples of features in applications where the pattern is used
• Related patterns		

Table 8 : 10 Template of design pattern for use in the cooperative design process of end-user tailoring.

The template is constructed so that it begins in a general way and becomes more detailed and specialized further on. It is essential to remember that the descriptions in the pattern template have to be written in a way that complies with the needs of different types of stakeholders.

8.5 Discussion

That design patterns are useful when designing software has been proven over and over again during the past decades. In 1997 when the design pattern concept in software engineering was intensely discussed, Pree and Sikora (1997) expressed their concern about design patterns being a hype, but now ten years later we are beyond the hype (Buschmann et al., 2007) and we can see that design patterns are here to stay. We have made an attempt to adjust a part of the concept of patterns to end-user tailoring. Apart from the previously discussed benefits from using patterns, the use of patterns can also decrease development time (Bass et al., 1998). Since there are constant discussions regarding the trade-off between the benefits of tailoring and the possibly increased development time for a tailorable system, decreased development time is advantageous.

We believe that the selection of usability patterns presented in Section 8.2 can act as a gateway to a wider use of patterns in cooperative design projects developing end-user tailorable software. It is our hope that users as well as developers may find the patterns beneficial and be encouraged to gradually incorporate more patterns. As the patterns are kept separate and not related in a comprehensive pattern language, the patterns can be used in any type of development process, independently of other tools used in the process. It is also possible to simply be inspired by the patterns to be used for a specific type of tailoring and then use whatever pattern structure you prefer. But the intended use is that a team consisting of different types of stakeholders can discuss tailoring, using the categorization as a base. As the categorization explicitly defines both a user perspective and a system perspective it is easier to reach a consensus of the tailoring that is needed. When the participants have agreed upon which type of tailoring is needed they can continue the design process and then go further and look for which patterns should be considered for the chosen category of tailoring. The other usability scenarios that also have an architectural impact, but are not vital to tailoring can be used as a checklist to find out if all essential usability issues are taken into account. If the participants find patterns to be useful, they can use the corresponding usability patterns for the usability scenarios that were found to be important for the software.

How does our approach differ from the other approaches discussed? Borchers's approach (2001) involves a pattern language that guides the team members to the next pattern. He, as we also do, advocates patterns as a *lingua franca,* but there is a difference. When Borchers assumes collaboration between the users and the usability experts and other cooperation between usability experts and developers, we advocate a direct cooperation between all the different

stakeholders. We have previously not discussed usability experts at all, but we believe that usability experts are closer to the software than to the task and we have therefore incorporated usability experts in the term developer. The intention of having the same pattern structure for all types of patterns dealt with within the project is advocated by us as well as Borchers.

Schümmer and colleagues (Lukosch and Schümmer, 2006, Schümmer and Slagter, 2004) have, in the same fashion as Borscher, constructed a whole process that is based on a pattern language. We have started in the small by introducing a small selection of vital unrelated patterns. Schümmer et al. support an iterative process and so do we. One of the advantages of patterns is that you can and may focus on one feature at a time and in an iterative way fill up with new features and patterns. Also Schümmer et al. use patterns as means of communication and learning and their pattern structure becomes more detailed further on, in the same way as ours does.

It is John and Bass (Bass and John, 2003, John et al., 2004) who have taken the most unusual approach, by explicitly naming the different forces influencing the design decisions. We find their work with forces very insightful and as their findings are mirrored in our experience from industry, we felt it was essential to incorporate the forces in the pattern structure for end-user tailoring. Unlike us, John and Bass have built in a sort of process in the pattern structure. For example the responsibilities of the general solution are transferred to the section of specific solutions to get a better overview of what the specific solution should look like.

The last approach, but the most well known, is the approach of the Gang of Four, Gamma et al. (1995). Gamma et al. also have patterns that are not related in a pattern language. The main difference between Gamma's et al. approach and ours is that the patterns are mainly intended for developers and are described thereafter. But the patterns are intended as a base for communication even though it is within the developers' group.

8.6 Summary

The study has resulted in a subset of usability patterns with architectural impact and suitable software design patterns for end-user tailorable software. The subset is matched with a corresponding tailoring category to make it possible to focus on a few patterns. The selection of vital, not negotiable, usability patterns is intended as a sample of how useful patterns can be in a cooperative design process. By allowing for designing with this kind of building blocks the cognitive load of the participants decreases (Bass et al., 1998) and the patterns can be a mediating artefact in the design discussions and decisions. The study also resulted in a pattern structure for patterns of end-user tailoring design. The pattern structure is a merge between several different approaches to be able to satisfy the needs of both users and developers. The patterns have to be easy to grasp and understand as well as detailed enough to be useful when

implementing the software. This is achieved by starting with a prosaic description of problems and a general solution and then a more detailed description of the solution is presented along with detailed diagrams and so on. This latter part aims more at the developer, but it is also our belief that interested users become more and more familiar with the pattern structure and gradually learn the meaning of, not only the beginning of the patterns, but also the more detailed and developer adjusted part.

Chapter Nine

Toolkit

Chapter Nine

Tools to Support the Cooperative Design Process

As discussed in Part I, a cooperative design process that includes users and developers is needed in order to make durable end-user tailorable software. The benefit of collaboration is that the decisions made in cooperation potentially find greater acceptance. But the basic prerequisite for achieving this benefit is that the people affected by the decisions (or their representatives) participate in collaboration, and that there is productive communication between the participants.

This chapter presents four tools that can be used as a base of communication in the cooperative design process of end-user tailoring. The tools are at this stage paper based and can be found in Appendices B to E. Each tool consists of four components:

- An artefact that is the core of the tool.
- Documents supporting the use activities.
- A BoundLet[1]. The instructions for how to use the tool are contained in a document called BoundLet. The BoundLet also contains information about, for example, in which situations use of the tool is appropriate, and which rules should guide the use of the tool.
- Additionally, the BoundLet is accompanied by a document giving an overview of the workflow, showing in which step of the instruction the different documents are used.

The creation of the tools was guided by the findings from Project 4, where a development project was observed and an interview study with both users and developers was performed. During the studies some *collaboration issues* were revealed:

- There were misunderstandings concerning flexibility.
- There was no common ground regarding what tailorable software means.
- In interviews the users expressed a desire to gain a better understanding of the technology and learn more about the decisions behind the software.
- The respondents also expressed a desire to achieve a shared responsibility for the developed software product, ensuring that both users and developers feel they own the software.

[1] The origin of the name of the document is discussed in Section 9.2

- It was also revealed during observation of the development project that there were concerns about creating a good architectural design for the situation.

The empirical findings in Project 4 lead to following conclusions:

- Misunderstandings concerning tailorable software indicate a need for a common base for discussion and *communication*. A concrete action to take is to implement some kind of *classification* (Figure 9 : 1 (a)).
- Users want to understand the technology of tailorable software better, which shows the need for a *learning* environment that makes it possible for the user to understand technical decisions and their consequences for use. In practice, it means that the tools to support the cooperative design process should enable the users to gradually learn more. The tools should be on different *levels* (Figure 9 : 1 (b)).
- The request to *share the responsibility* for the software product points to the need for both users and developers to *take part in design decisions* to come to an agreement about trade-offs. The parties must share an understanding of the decisions and how they influence the software construction. It means that users and developers must explicitly discuss the context and the environment, bring individual thoughts to the surface, and not take anything for granted or let anything be unspoken. The tools should promote shared decisions (Figure 9 : 1 (c)).
- Uncertainty concerning whether the software constructed is the most suitable solution for the situation indicates that there is a need for support in the architectural design of the software. Design *patterns* have been found to help in achieving the *right design* faster, and patterns are grounded in successful experiences (Gamma et al., 1995). By discussing patterns for the software the parties can also share a mental model of the software (Figure 9 : 1 (d)).

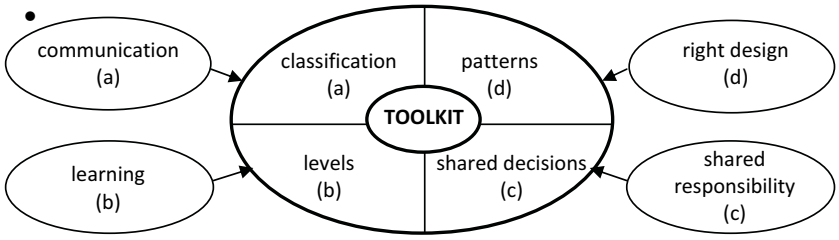


Figure 9 : 1 Relationship between cooperation issues and the developed tools

The issues or requirements concerning collaboration led to the creation of four artefacts: Categorization, Matrix, Usability Patterns and Design Patterns. The artefacts are at the core of four different tools:

- Categorization tool to get a common understanding of tailoring.
- Matrix tool to discuss which tailorability to implement.

- Usability Pattern tool to implement the needed usability patterns and to discuss the impact on the architecture and the trade-offs this entails.
- Design Pattern tool to reach a consensus about design decisions and initial trade-offs.

There were two factors which guided the creation of the artefacts from the start, since these two factors are the basis of the collaboration: that they should act as a basis for discussion and be used in Participatory Design activities. The correspondence is shown in Table 9 : 1. The artefacts are presented in detail in Chapters Six to Eight and the corresponding tools are presented in Section 9.1.

COLLABORATION ISSUES	IMPLEMENTATIONS	Basis for discussion	PD techniques	TOOLS
Misunderstandings in communication	Classification			Categorization tool Matrix tool
Users want to learn about techniques	Levels of tools			Usability Patterns tool Design Patterns tool
Shared responsibility for product	Shared design decision			
Good software and architecture	Pattern			

Table 9 : 1 Relationship between the collaboration issues (requirements) and the created artefacts

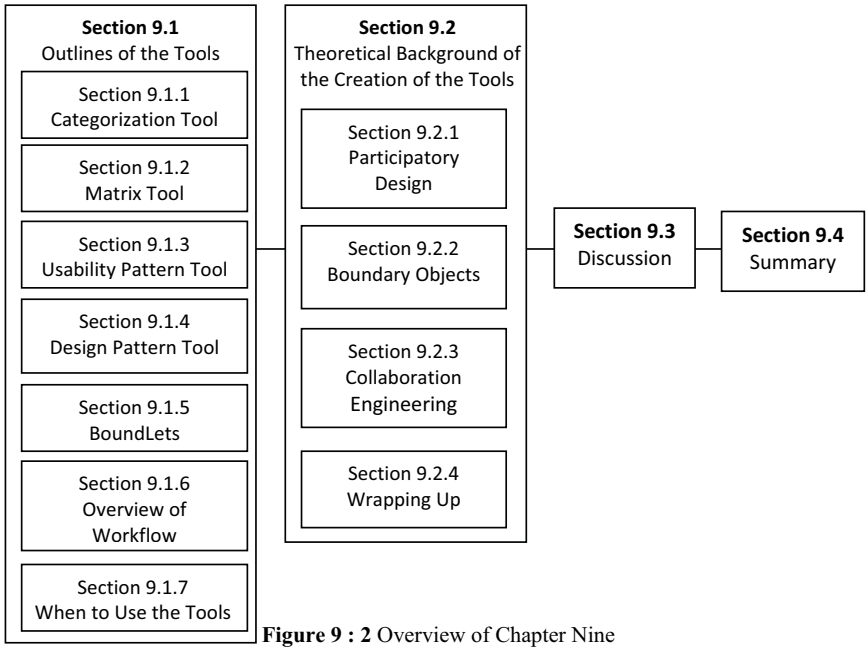


Figure 9 : 2 Overview of Chapter Nine

The rest of the chapter is structured as shown in Figure 9 : 2. The outlines of the different tools are presented first (Sections 9.1.1-9.1.6) and in Section 9.1.7 the use of the tools is described. Thereafter, in Section 9.2 the theoretical background or underpinning of the creation of the tools is discussed, together with the related work. The toolkit relates mainly to three areas: Participatory Design, Boundary Objects and Collaboration Engineering. The chapter ends with a discussion and a short summary.

9.1 Outlines of the Tools

In this section the four tools (Categorization tool, Matrix tool, Usability Patterns tool and Design Patterns tool) are presented. Each tool consists of four parts:

- Artefact (Categorization, Matrix, Usability Patterns, Design Patterns)
- Support documents
- BoundLet with instructions
- Workflow overview

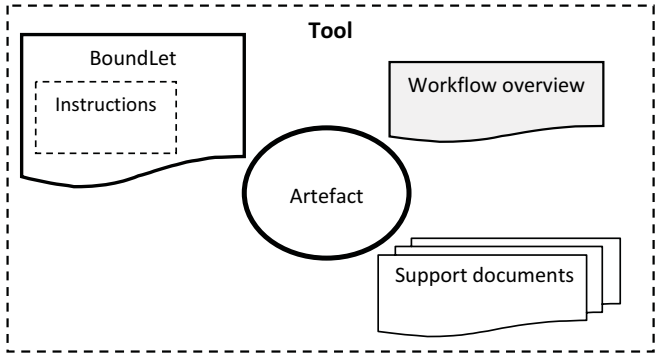


Figure 9 : 3 Outline of a tool

The artefact is the core of the tool and the other parts support (Figure 9 : 3) the use of the artefact in different ways. The instructions are a guide to how to work with the artefact, and the supporting documents are documents that either clarify the artefact or facilitate its use by providing lists or additional instructions to fulfil the task. The BoundLet (Section 9.1.5) packages the instructions together with, for example, advice on when to use the tool or which rules should guide the use of the tool. Finally, the workflow overview visualizes the connection between the different parts. The workflow (Section 9.1.6) relates the instructions to the other documents.

The next section begins by presenting the tools with their artefacts, instructions and support documents. The overall structure of the BoundLets and the workflow are similar for all tools, and these two parts are therefore treated separately in Sections 9.1.5 and 9.1.6. The section ends with a short discussion of when to use the toolkit. It is advisable to look at the appendices while reading the following subsections.

9.1.1 Categorization Tool

The tool can be found in Appendix B.

The empirical studies revealed that misunderstandings arose in the communication when discussing end-user tailoring. Both users' and developers' experiences can be made use of by using the categorisation as a starting point to discuss different kinds of tailoring and thus find a common base for further work. The purpose of the Categorization tool is to get a common understanding of tailoring

Artefact

The artefact of the Categorization tool is the categorization of end-user tailoring (Table 9 : 2). The categorization defines four types of end-user tailoring from both a user and a system perspective, e.g. how users and developers perceive the different categories. The categorization is brief, but additional explanation is provided for in the tool. In addition the categories can be customized to describe a local situation more specifically. It is also possible to interpret the categories to suit the specific needs of the group whilst retaining the naming and overall meaning.

Category	User Perspective	System Perspective
Customization	Set parameter values	Interpretation of existing code
Composition	Link different existing components	Definition of relationships between components Code Generation (optional)
Expansion	Creation of new component	Definition of relationships between components New and predefined components are treated uniformly Code Generation (optional)
Extension	Insertion of code	New code is added Code Generation (optional)

Table 9 : 2 Categorization of end-user tailoring.(For more details see Chapter Six)

Instructions are provided to allow the use of the Categorization in the collaboration. The instructions are presented in the next section. The support documents are marked in bold italic text in the instructions and an explanation of the documents is found below the instructions.

Instructions

The instructions for how to use the tool are as follows:

1. Define what a flexible software system means to you. Write down your definition.

2. When everyone is finished you must display your definition for everybody to see.
3. Sort (together) all the definitions into suitable categories by using the artefact.
4. Use ***Example 1 to 4*** if you need a common example for the participants to relate to.
5. When sorting takes place, motivate and explain your definition.
    a. Why do you think of flexibility in this way?
6. Use ***BoundLetExtra: CreateAgreement*** to agree upon a common definition and to clarify differences in opinions.
7. Specify the descriptions of the categories in the artefact if this is needed for the categories to work in the specific context or situation.
8. Write down the common definition on a large paper and put it up on the wall.

Support Documents

Examples 1 to 4 exemplify the categories to make it easier to grasp the differences between the categories if the group does not have a real world case to relate to.

BoundLetExtra is a name for some BoundLets that are add-ons to BoundLets presented here. The BoundLetExtras are general and can be used in different situations and they do not contain any artefact. In this case a BoundLetExtra called *CreateAgreement* is proposed to manage the process of reaching a valid agreement between the participants.

9.1.2 Matrix Tool

The tool can be found in Appendix C.

Both users and developers are experts in their own field. In the Matrix tool the participants have the opportunity to discuss different aspects of the specific context relevant to the implementation of the software, and to reach a common understanding of what is needed. The core of the Matrix tool is the Matrix, which is created on the basis of interviews with both users and developers and should thereby fulfil the information requirements of both users and developers.

The intent with the Matrix tool is to discuss which tailorability to implement

Artefact

The matrix consists of a set of attributes of end-user tailoring divided into three areas: Business changes, usability issues and software attributes. The attributes have been assigned values (L (=Low), M (=Medium) or H (=High)). The Matrix is shown in Table 9 : 3.

As an example of how to read the Matrix; the type of end-user tailoring called customization is perceived to provide a high degree of user control (bold text in Table 9 : 3. An explanation of the attributes is presented in Table 9 : 4.

Characteristics		Customization	Composition	Expansion	Extension
Business Changes	Frequency of change	M	M	H	H
	Anticipation of change	H	M	L-H	L
	System support of change	L	M	M-H	H
Usability Issues	User control	**H**	M-H	M-H	?[2]
	Transparency	H	M-H	M-H	?[2]
	Realization speed	H	H	M	M-H
	Frequency of use	L	H	-	-
	User competence	-	-	M-H	H
Software Attributes	Fault tolerance	H	M-H	M	L
	Complexity	L	L- M	M	H

Table 9 : 3 The Matrix (For more details see Chapter Seven)

Area	Attribute	Explanation
Business Changes	Frequency of change	how often the business changes occur, frequently or infrequently
	Anticipation of change	to what extent it is possible to anticipate the business changes
	System support of change	how well the software must support business changes
Usability Issues	User control	how much control do the users have to have of what happens in the software
	Transparency	how easy it should be for the users to know if the result is correct
	Realization speed	how fast it should be to realize the changes in the software.
	Frequency of use	how often the functionality will be used
	User competence	how skilled the users are
Software Attributes	Fault tolerance	to which degree the software has to prevent mistakes.
	Complexity	how complex the software can be

Table 9 : 4 Meanings of attributes

[2] ?= Uncertainty. Is the control obtained by the user or the software? (see Chapter Seven)

The attributes and the attribute values can all be customized to more specifically suit a particular context.

Instructions

The matrix may be hard to understand at a glance. The instructions should make it easier and allow the participants to gradually become acquainted with the content of the matrix. In the next sections, the support documents marked in bold, italic text are explained.

The instructions for how to use the tool are as follows:

1. On the basis of the ***questions***, define what the context (business market, organisation, personnel, software infrastructure) is like for the flexible functionality that shall be implemented.
    a. Mark your answer individually. Deal with one question at a time and judge it as low (L), medium (M) or high (H). Use the ***personal form.***
2. Discuss in the group how the answers vary and the reason for any differences in the answers.
3. Mark the joint answer in the ***group form.***
4. When all the questions are dealt with, the group's judgements are compared with the Matrix. The number of corresponding answers is summarised.
5. The result is discussed with the help of the ***speech bubble questions***.
6. ***BoundLetExtra: CreateAgreement*** is used to reach an agreement of what type of flexibility to try to achieve for the functionality.

Support Documents

The *questions* are based on the explanations of the attributes (see Table 9 : 4) and guide the participants concerning what to take into account. The process of defining the context is not limited to these questions or attributes. Additional questions should be formed to discuss the specific context.

There are two *forms (personal and group)*. In these forms, the participants can mark their answers and document their opinions. In this way the decisions and considerations are traceable and it is possible to go back and reconsider the choices.

The *speech bubble questions* are questions intended to initiate a deeper, nuanced discussion of what kind of tailorability is needed. This list of questions should also be extended to be more specific. The questions are called speech bubble questions as they are put in a speech bubble to emphasize that a discussion is encouraged.

The BoundLetExtra *CreateAgreement* is proposed to manage the process of reaching a valid agreement between the participants.

9.1.3 Usability Pattern Tool

The tool can be found in Appendix D.

The pattern approach is a way to preserve successful experiences, to impose a good architectural solution on the software, and to inspire confidence in the quality of the software.

Patterns are used in both the Usability Pattern tool and in the Design Pattern Tool. Others have also used usability patterns elsewhere in Participatory Design. For example, Dearden et al. (2002) report their experiences of developing and evaluating the use of pattern languages in participatory design of web-based systems. They state that the users found the patterns helpful once they were familiar with the patterns.

Since they are close to the user domain, usability patterns are a good introduction, as they provide a gentle learning slope for how to use patterns. Since the purpose is to involve the users in the technical design process, usability patterns with architectural impact are used. In this way the step from usability patterns to design patterns becomes shorter. These types of usability patterns should be considered early in the design process just like other design patterns.

The intention with the Usability Pattern tool is to implement the needed usability patterns and to discuss the impact on the architecture and the trade-offs this entails.

Artefact

The artefact of the Usability Pattern tool is a number of usability patterns with architectural impact to be considered early in the design process. The structure of how the patterns are presented is shown in Table 9 : 5. The table is not populated with data. It is only the structure that is defined, as the content should be adapted to the specific context.

The pattern structure is created to comply with the needs of both users and developers. The pattern structure starts in a quite general way that should be easy for any participant to grasp. As they deal with levels that are deeper in the structure, the patterns become more and more specific, and closer to architectural solutions. In this way the structure should fulfil the information requirements of both communities. A pattern has different meanings for users and developers and what the tool does is coordinate and facilitate the discussions to reveal differences and to provide a common ground to work from. The intention of the patterns is that they shall be applicable in different settings and it is up to the team to make the patterns more specific and concrete.

Usability Pattern for end-user tailorable software	
Introductory description	
Name	
Ranking	
Tailoring Categories	
Illustration	
Overall description of problem and solution	
Problem	
• **Forces**	**Environment and task**
	Human desires and capabilities
	State of the software
General Solution	
Detailed description of solution	
• **Specific Solution**	
• Prior design decisions	
Diagrams	
Consequences	
Danger spots	
Sample code	
Examples	
Related patterns	

Table 9 : 5 Structure of usability pattern. (For more details see Chapter Eight)

Instructions

The instructions for how to use the Usability pattern tool are rather complex and consist of two loops: one loop to work through the vital usability patterns (mandatory patterns) and one loop to work through other relevant patterns. In the next section, the support documents marked in bold, italic text are explained.

The instructions for tool use are as follows:

1 - VITAL PATTERNS

a) Based on the flexibility type chosen, the ***vital usability patterns*** are selected.
b) By using the patterns description, work through the Speech Bubble Questions together.
c) Work through the parts of the pattern description that have not been considered. Base the work on the maturity of the group. A more mature

group can proceed deeper into the pattern description. Does this add anything to the assessment?

d) Move to the next pattern by returning to b).

2 – OTHER PATTERNS

a) Work through the other ***usability scenarios*** and choose those that are relevant for the situation.
e) Choose the usability patterns corresponding to the usability scenarios
f) Based on the name, make a preliminary prioritization. Mark the result in the ***priority list***. Which pattern is most important? Start with that pattern.
g) By using the pattern's description, work through the ***Speech Bubble Questions*** together.
h) Work through the parts of the pattern description that have not been considered. Base the work on the maturity of the group. A more mature group can proceed deeper into the pattern description. Does this add anything to the assessment?
i) Chose the next pattern. Repeat g) to i) until all the patterns are worked through.
j) When all the selected patterns have been worked through, prioritize the patterns (together in the group).
k) Write down the prioritizations and comments. Use the ***priority list***.

Support Documents

It is convenient to present a small set of patterns together (Dearden et al., 2002). Therefore a small selection of *vital usability patterns* that are a subset of usability patterns is presented first to the participants. The selection is based on empirical studies, and the reason for dividing the usability patterns with architectural impact into sub groups is that it is easier to grasp a small set of patterns, particularly if the participants are beginners in the area of patterns. To help the participants probe into the area of patterns the vital usability patterns are mandatory and no selection must be made.

The *usability scenarios* are an entrance to the usability patterns. One scenario can be supported by several patterns. Therefore it is easier to start with the usability scenarios and reach the patterns from there.

The *priority list* is a document where the participants can mark their prioritizations, thereby documenting the prioritizations.

9.1.4 Design Pattern Tool

The tool can be found in Appendix E.

In the empirical studies it was found that there are users who want to learn more about the techniques and architecture of the software. By introducing the Usability Pattern and Design Pattern tools it is possible to gradually get used to working with patterns and gradually learn more about the techniques. Different levels of learning are introduced, both through the two patterns themselves and

through the pattern structure, as it gradually gets more detailed. As the users become involved in the technical decisions, the pattern tools provide a potential foundation of shared responsibility for the software, since all participants are invited to a democratic process of deciding how to implement the software.

The overall intention with the Design Pattern tool to reach a consensus about design decisions and initial trade-offs.

Artefact

The artefact of the Design Pattern tool is the design patterns themselves, which are considered as suitable for end-user tailorable software. The pattern structure is similar to the structure of usability patterns. The only difference is that the design patterns also contain a metaphor visualizing the pattern, making it easier to choose a pattern.

Instructions

The instructions for this tool are rather straightforward since it involves iterating through a set of selected patterns. In the next section, the support documents marked in bold, italic text are explained.

The instructions for how to use the tool are as follows:

1. A collection of design patterns are chosen, dependent on the type of flexibility that is to be implemented. Use ***Base for selection of Design Patterns***
2. Based on the pattern metaphors, the patterns that best match up to the idea of the software system are chosen.
3. The participants work through the pattern by using the Design Pattern's pattern description**.** The ***Speech Bubble Questions*** may help the work.
4. Continue with the next pattern, in order to gain an overview and understanding of the different patterns (go to 2)
5. Compare the patterns. Use ***BoundLetExtra: Evaluation.***

Support documents

The *Base for selection of Design Patterns* is a document describing which design pattern can be suitable for a specific category of end-user tailoring.

The tool also contain another BoundLetExtra to use; *BoundLetExtra: Evaluation*. That is, a general BoundLet providing assistance when evaluating and comparing different alternatives.

In the next section BoundLets are introduced as a general concept. The specific BoundLets for the tools can be found in Appendices B to E.

9.1.5 BoundLets

The BoundLets package information on how to use the artefact.

A BoundLet defines the following elements:

- Input and output
  - Specifies what is required to use the tool.
  - Specifies which result can be expected.
- Choose this tool if…
  - Specifies in which situations the tool is suitable
- Overview
  - Gives a brief overview of how to use the tool.
- Artefact
  - Specifies which artefact is contained in the tool.
- Instructions
  - Specifies how to use the tool.
- Rules
  - Specifies the overall rules that guide the use of the tool.
- Experiences
  - A section where the participants in the group can collect positive and negative experiences of the use of the tool to guide future use.

9.1.6 Overview of Workflow

Each tool also contains an overview of the workflow when using the tool. Figure 9 : 4 shows an example of an overview of a workflow, in this case the workflow for the Categorization tool. The figure visualizes how the BoundLet embraces instructions, an artefact and support documents. As we can see the tool contains one BoundLet, one artefact and several support documents (in this case two documents). The artefact and the support documents are connected to the steps in the instructions where they are used.

The intention of the overview of the workflow is to increase the tools' affordance. Also, when the group is used to the tool and knows the different steps and their meaning, the workflow can act as a reminder of what to do and in which order.

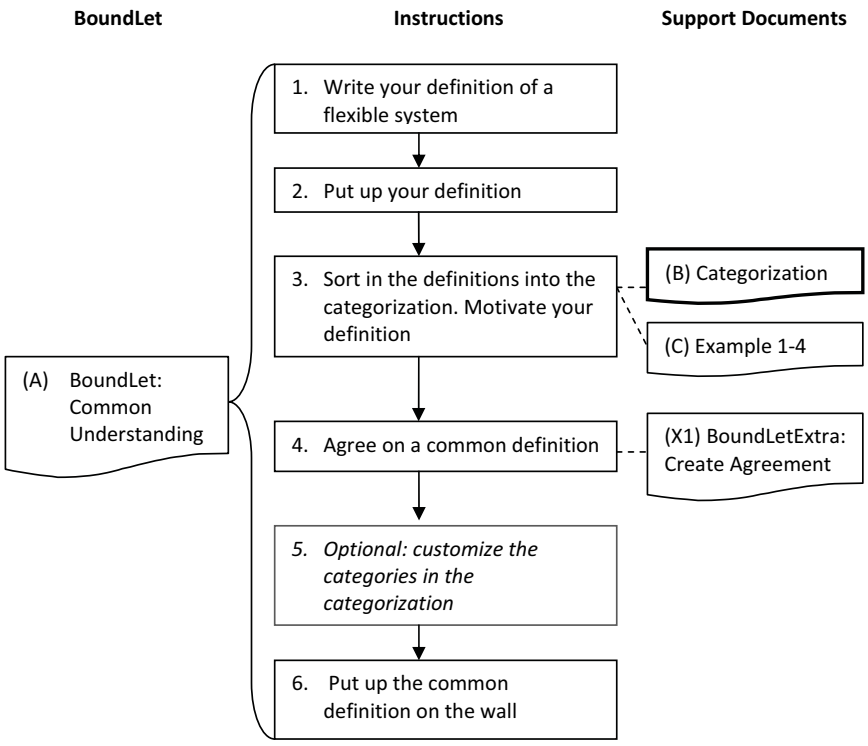


Figure 9 : 4 Example of overview of workflow

9.1.7 When to Use the Tools

The tools can be used when the tailoring capabilities for an end-user tailorable system have to be extended. In such situations, the developers, users and tailors have to work together to achieve a suitable tailoring capability. The prerequisites for the use of the tools are that there is a need for a *basis for discussion* between the different participants and that there is a *Participatory Design setting* where developers and users work together in a democratic design process.

The tools are not intended for use in an intact sequence. They should be used just like other PD technique, when they are required. This means that it is important that the tools can adapt to local constraints and requirements, or in other words be used in different situations, and that they can be used in any process. This means that it is essential that the tools are freestanding from each other and from the overall process. The tools are therefore not connected to a specific type of development process and can be used in different phases and different processes.

There is, however, a relationship between the tools in the toolset (Figure 9 : 5). The four categories of end-user tailoring from the categorization in the Categorization tool are represented in the three other tools. The values in the matrix are based on the different *categories* and the *categories* are included in the patterns to imply where the patterns are suitable.

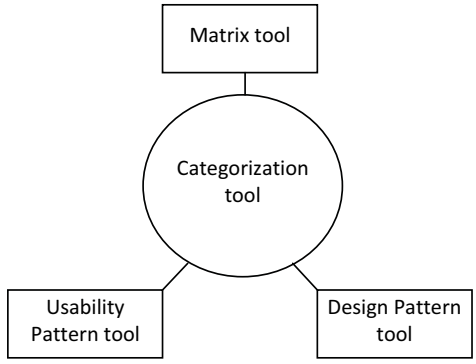


Figure 9 : 5 Relationships between the tools in the toolset

However, if *all* of the tools are used at different occasions in the cooperative design process the order of use should be

- Categorization tool to get a common understanding of tailoring.
- Matrix tool to discuss which tailorability to implement.
- Usability Pattern tool to implement the needed usability patterns and to discuss the impact on the architecture and the trade-offs this entails.
- Design Pattern tool to reach a consensus about design decisions and initial trade-offs.

9.2 Theoretical Background of the Creation of the Tools

In this section the theoretical background to the creation of the tools is discussed. The tools were created to support the cooperative design process of end-user tailoring, since the empirical studies revealed a need for such tools.

The tools are intended to support cooperation between users and developers by bringing them together in discussions of end-user tailoring that gradually deepen the participants' understanding of tailoring, making it possible for the users to participate in technical design discussions and decisions. It is essential that the users are able to take part in these discussions since in end-user tailoring they are co-designers, and it is important that they understand the underlying decisions about the design, to recognize the possibilities and boundaries inherent in the software.

Therefore, the two foundations for the creation of the tools were that they should be able to

- work as Participatory Design techniques, promoting a democratic design process where all participants take part, and
- act as basis for communication between the different perspectives among the participants.

These issues lead to three related areas:

- *Participatory Design* since the tools are intended to support a democratic process,
- *Boundary Objects* as the toolkit aims for mediating between diverse competences in diverse projects consisting of different types of stakeholders, such as developers and users.
- The area of *collaborative engineering* (CE) is also relevant since CE deals with the issue of repeating successful collaboration sessions, which has been experienced to be a problem in PD (Kensing and Blomberg, 1998).

A short overview of the three related areas is given below.

9.2.1 Participatory Design

The core principles of PD are (Sanoff, 2007)

- that every participant is an expert in their own field,
- that every participant's voice must be heard,
- that good design solutions come from the collaboration of diversely composed groups,
- participatory democracy in decision making and
- engaging people in changing their own environment.

In summary, those individuals that have to adapt to the introduced change should be part of the decision making (Kensing and Blomberg, 1998).

The participation can range from users being limited to supplying designers with access to the users' skills and experience, to the users being considered valuable since their interest in the design solution is recognized. In this last type of setting the users take part in the analysis of the requirements, the evaluation and selection of technological components, the design and prototyping as well as the organizational deployment (Kensing and Blomberg, 1998).

Tools and the development of tools is an essential part of PD projects. The *techniques* utilize informal ways of exposing the relationship between the work and the technology. There are many *tools and techniques* to be used in a PD project ranging from techniques for analyzing the work to tools to use in system design (Kensing and Blomberg, 1998). The tools and techniques can be used in different phases of the development cycle or iteration.

A more elaborate description of Participatory Design can be found in Chapter One.

9.2.2 Boundary Objects

Communities of practice (Lave and Wenger, 1991) are a central term in the context of Boundary Objects. A community of practice cuts across formal organizations and can be seen as relations between people working together. Bowker and Star (1999) state that Boundary Objects are a way to handle different perspectives in communities of practice.

Boundary Objects have the following characteristics (Bowker and Star, 1999)

- They are applicable in several communities.
- They fulfil the requirements of information from each community.
- They have a constant identity across communities.
- They can be tailored to meet the needs of a community.
- They are both ambiguous and constant. (They have common identity across settings. They are weakly structured in common use and more strongly structured in specific use.)
- They can be abstract or concrete.

Boundary Objects emerge when there is a stable relationship between different communities of practice and shared objects are built (Bowker and Star, 1999). Boundary Objects were first observed in scientific settings where different participants with different perspectives work together to balance different categories and meanings (Star and Griesemer, 1989). In other words, the Boundary Objects arise from practice.

Boundary Objects can also be seen as evolving artefacts that become meaningful and understandable when they are used (Fischer and Ostwald, 2001). Boundary Objects can act as basis for discussion, initiating relevant knowledge and shared understanding. Or as Fischer et al. (2005, p.10) put it: “It is the interaction around a Boundary Object, not the object itself, that creates and communicates knowledge.”

9.2.3 Collaboration Engineering

The research field of Collaboration Engineering (CE) arose from the trend that organizations more and more frequently use collaborative teams to produce increased value for their stakeholders (Kolfschoten et al., 2006). CE aims at designing and deploying processes that can be used and executed by practitioners themselves without the involvement of professional facilitators.

Group Support Systems (GSS) can increase the productivity of a team, however the success of GSS sessions is somewhat unpredictable (de Vreede et al., 2003). For the potentials of GSS to be realized their use must be guided by experience (Kolfschoten et al., 2006). Therefore many organizations use facilitators to benefit from GSS. CE aims to find a way for teams to gain advantages from GSS and to manage the collaboration process themselves, and still reach a predictable result (Kolfschoten et al., 2006).

ThinkLets is a key concept of CE. ThinkLets is a technique that produces a pattern of interaction between people working together to reach a goal. ThinkLets are building blocks that can be put together to design team processes. "A ThinkLet is a named, tightly scripted, process for creating a single repeatable predictable pattern of collaboration among people working together towards a goal." (Kolschoten et al., 2004, p. 1). ThinkLets have been around for a while but it is only recently they have been formalized (Kolfschoten et al., 2004). ThinkLets are built on known techniques, for example brainstorming.

There are five types of ThinkLets (Briggs et al., 2003):

- Diverge
    - you start with a few concepts and end up with more
- Converge
    - you have several concepts and end up with a few concepts worth more attention.
- Organize
    - you gain more understanding of the relationship between concepts
- Evaluate
    - you gain more understanding of the consequences of choices
- Build consensus
    - you gain more agreement between participants in a group and you gain more congruence between individual goals and group goals.

A ThinkLet is defined in terms of (Kolfschoten et al., 2006):

- name,
- pattern of collaboration,
- successor and predecessor,
- capabilities (what is required to take the actions),
- actions (what to do) and.
- rules with constraints (how to do the actions).

9.2.4 Wrapping Up

Participatory Design research has often been criticized for functioning well as long as the PD facilitator is present, but when the development team must stand on its own feet it is difficult for them to repeat the successful design effort (Kensing and Blomberg, 1998). Collaboration Engineering (CE) deals with the same problem, but in terms of group support systems (GSS). Collaboration Engineering means "the development of repeatable collaborative processes that are conducted by practitioners themselves" (Briggs et al., 2003, p. 32). To be able to repeat successful patterns of collaboration ThinkLets are created. ThinkLets define a facilitator's actions and choices of how to use the GSS to facilitate collaboration. By using ThinkLets facilitators can develop and transfer successful processes to practitioners themselves (Briggs et al., 2003).

A Boundary Object must have a certain degree of 'perceived affordance' (e.g. "the appearance of the device could provide the critical clues required for its proper operation." (Norman, 1999, p. 39) The proposed artefacts do not possess this. Perceived affordance is built partly on previous experience and the first time the participants use the artefacts they do not have any previous experience. The lack of perceived affordance therefore makes it necessary to provide instructions together with the artefacts.

ThinkLets worked as inspiration to encapsulate the instructions for how to use the artefacts into BoundLets. In this way the BoundLets add affordance to the tools as they define how to use the artefact. BoundLets contain elements similar to those contained in ThinkLets, such as rules and indications of when to use them, but there are also differences (Section 9.3). The reason for creating a BoundLet is to make it possible for the participants in development projects to use the tools without involvement of a facilitator. It is not thereby said that there should not be any facilitator. There will certainly be a need for one in the initial stages, but as time passes, and after the participants have used the tools for some time, it is possible for one of the practitioners to act facilitator since he or she has support of the tools.

9.3 Discussion

The tools proposed in this chapter can be seen as techniques or tools for use in Participatory Design. However the tools should not be seen as excluding other PD techniques. The tools simply aim at building a common understanding of end-user tailoring in a specific context and involving the users in the technical design process. Participatory Design should be considered in all stages of the cooperative design process of end-user tailoring, meaning that techniques such as mock-ups and future workshops are likely to be used in other phases of the collaboration.

The facilitator is central to Participatory Design. In the context of using patterns in PD the facilitator is important to support users by helping them interpret the patterns and also to interpret users' statements (Dearden et al., 2002). The tools presented in this chapter aim at reducing the need for a facilitator. Role hybridization can be a solution to this, meaning that users act in a hybrid role as both user and developer. Fleischmann (2006) has observed this phenomenon. During the cooperation with the telecom company, we have observed the same thing. We have seen both a developer who changed tasks to become a user and a user gaining employment in the IT-department of the company due to interest and involvement in developing one of the software systems.

The tools are intended to act as a Boundary Object between users and developers in the design process. Bowker and Star (1999) argue against introducing artificial Boundary Objects (like, for example, standards) as they "strip away the ambiguity of the objects of learning" (p. 305) which often

means “empowering the self-proclaimed objective voice of purity” (p. 307) and thereby focusing too narrowly and ruling out relevant information.

The tools should be able to act as proper Boundary Objects in the same way as Boundary Objects that have arisen from practice, because, even though they are artificial from the beginning, they preserve the ambiguity and complexity of design discussions and avoid pointing out what is right or wrong. The tools should comply with both users’ and developers’ needs and thereby conform to the definition of a Boundary Object, since the tools:

- are applicable in both the user and developer communities.
- fulfil the requirements of information from both communities.
- have a constant identity across communities.
- can be tailored to meet the needs of a specific context.
- are weakly structured in common use and can be more strongly structured for specific use.

Others have also considered artificial Boundary Objects. For example one of the intentions of a workshop at the Conference on Human-Computer Interaction 2004 (CHI 2004) (John et al., 2004) was to propose new Boundary Objects for the gaps between UI developers and software engineers. Also Fischer et al. (2005) are exploring how to “create active Boundary Objects that can activate information relevant to the task at hand” (p. 491).

One of the criteria of Boundary Objects is to be plastic enough to meet different situations (Bowker and Star, 1999). The content of the proposed artefacts can be altered to be more specific and therefore meet different situations and needs. The instructions can also be altered and the affordance of the artefacts will also increase when the participants become more familiar with the tools. Thereby the artefact will be more and more rooted in the community and more and more similar to a Boundary Object arisen from practice. It also means that it becomes easier to modify the object.

Collaboration Engineering and Boundary Object might seem to be diametrically opposite. CE deals with extremely formalized concepts like ThinkLets, while Boundary Objects develop spontaneously from work in communities. However, on a higher abstraction level there are similarities. Both ThinkLets and Boundary Objects facilitate collaboration; the ThinkLet by defining the forms of the collaboration and the Boundary Object by mediating the communication between different perspectives. The tools proposed here combine the two concepts by introducing BoundLets.

BoundLets are inspired by ThinkLets and accordingly there are similarities, but also differences between the concepts. A comparison of the two concepts is shown in Table 9 : 6. A ThinkLet often has a catchy name, capturing the pattern of collaboration, such as the LeafHopper, symbolizing how the participants jump from concept to concept. The BoundLets have names that describe the theme of the discussion. The BoundLets only define the successor and this is

done inside the instructions. Likewise, the capabilities are also described in the instructions. For example, if some specific documents are needed in the discussion, this is defined in the instructions. The actions and the rules in ThinkLets correspond to the instructions and the rules in the BoundLets.

ThinkLets	BoundLets
Name,	Name
Pattern of collaboration	n.a.
Successor and predecessor	Successors are defined in the instructions
Capabilities	Defined in the instructions
Actions	Instructions
Rules with constraints	Rules

Table 9 : 6 Similarities and differences between ThinkLets and BoundLets

In conclusion, the main similarity between the concepts is that they aim to facilitate collaboration and reduce the need of facilitators. The overall difference is that the ThinkLet is related to Group Support Systems while the BoundLets are related to design artefacts.

Finally, the relations between humans and the constructed artefacts, power relations, social norms and policies make the outcome of collaborative techniques difficult to predict (DePaula, 2004). The tools are evaluated in Chapter Ten by an expert panel; however the tools have to be used in a real world development project for us to be able to make statements concerning their usefulness.

9.4 Summary

This chapter has described a toolkit that can be used as a PD technique for building a common understanding of end-user tailoring in a specific context and to involve the users in the technical design process. The tools in the toolkit combine Boundary Objects with the concept of ThinkLets from Collaboration Engineering.

The four tools in the toolkit are all intended to support cooperation between users and developers by joining them in discussions of end-user tailoring that gradually deepen the participants' understanding of tailoring, to enable the users to take part in technical design discussions and decisions. The four tools are the *Categorization tool* and the *Matrix tool,* aimed at reducing the misunderstandings that arise during communication when discussing end-user tailoring, and the *Usability Pattern tool* and the *Design Pattern tool,* which make it possible for users to learn more about the underlying techniques and are aimed at supporting shared responsibility for the product.

Chapter Ten

Evaluation

Chapter Ten

Evaluation of Toolkit

The first part of this thesis deals with the overall awareness of the problem of how to make end-user tailorable software sustainable. The result is the proposition that a continuous cooperative design process is needed. The second part of the thesis deals with how to make it possible for a cooperative design process to work, by suggesting a set of artefacts that are intended to facilitate the communication between end-users and developers. Chapters Six to Nine describe the suggested and developed tools, and this chapter treats the evaluation of the tools.

The research in this thesis follows the Design Research paradigm. Design Research basically consists of two activities: create and evaluate (March and Smith, 1995). The plan is that the design of the tools should go through three loops of the design process. The first loop, which is presented in this thesis, ends with an evaluation by researchers (expert evaluation). The second loop will be evaluated by practitioners at the company. The third loop will include implementing the tools in a real world setting, where the tools are tried out in a real project and evaluated by analyzing how well the tools worked as boundary objects (Bowker and Star, 1999) between the collaborating participants.

Foster-Fisherman et al. (2001) have created a framework of critical elements of collaborative capacity, which is useful in understanding which factors influence collaboration (Section 10.1). The framework of building collaborative capacity can act as a baseline when considering what to evaluate, and what must be considered as prerequisites to the evaluation. What we evaluate is the tool's *potential to influence positively the collaboration between end-users and developers*. Other factors, such as participants' attitudes towards the collaboration itself, are beyond the scope of the evaluation (e.g. a positive attitude towards collaboration is considered a prerequisite). The expert evaluation resulted in a set of concrete suggestions for improvements and a conclusion that the tools have the potential to influence the collaboration between end-users and developers as they e.g. provide for the formation of common concepts.

The rest of the chapter is structured as follows (Figure 10 : 1). First there will be an overview of the framework of collaborate capacity (FCC). Thereafter the scope of the evaluation is defined and presented. The framework of collaborative capacity acts as a foundation for the evaluation questions, and how the framework is used to evaluate the toolkit will be discussed in Section 10.3 where the list of evaluation question will be presented. In Section 10.4 the

result will be mapped out and the chapter will end with a summary and future work.

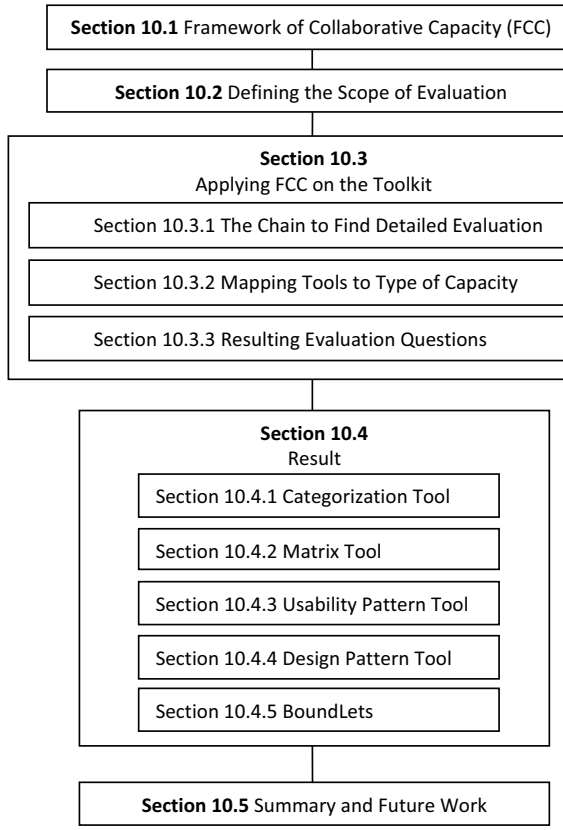


Figure 10 : 1 Overview of Chapter Ten

10.1 Framework of Collaborative Capacity

When the number of publications in the area of collaboration increased, Foster-Fisherman et al. (2001) attempted to develop a framework that captures core competences and processes that are needed for successful collaboration. Eighty publications were reviewed, resulting in a framework for building collaborative capacity. Collaborative capacity means the conditions needed for groups to establish effective collaboration (Goodman et al., 1998). The framework of collaborative capacity (FCC) makes it possible for researchers and practitioners to identify questions to ask and identify critical factors to target (Foster-Fisherman et al., 2001).

The framework identifies collaborative capacity at four critical levels that are essential for successful collaboration:

- Member capacity,
- relational capacity,
- programmatic capacity and
- organizational capacity.

Figure 10 : 2 gives an overview of the framework, showing the four levels and the main subcategories.

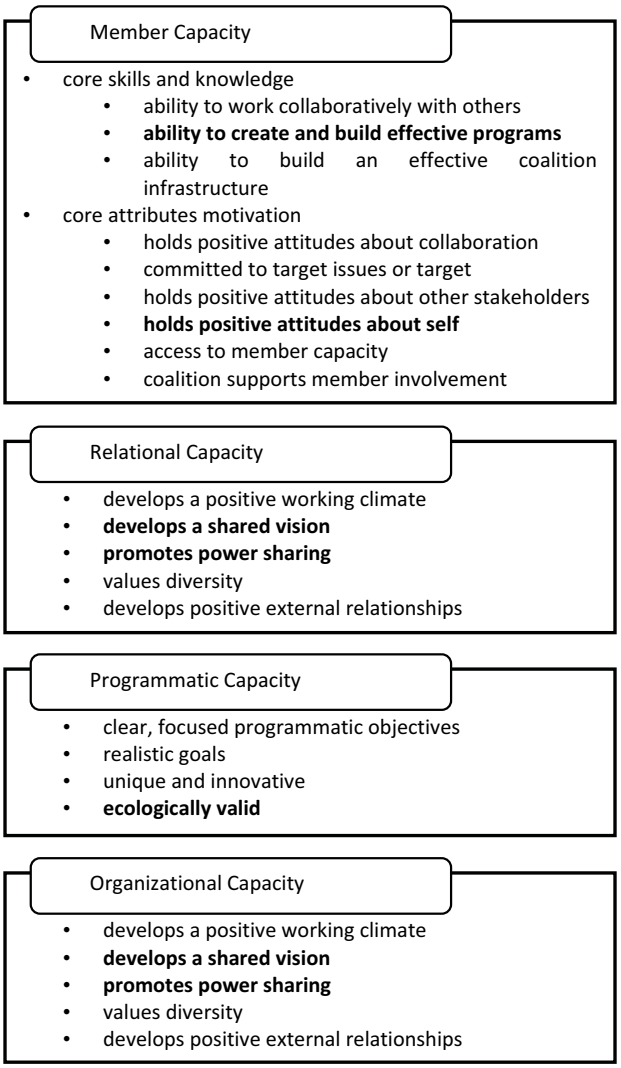


Figure 10 : 2 Critical elements of collaborative capacity (adapted from (Foster-Fisherman et al., 2001)) (The elements in bold text are used in the evaluation)

Collaborative work often requires certain types of specialized skills, attitudes and behaviour from the participants. The collaborative capacity is very much influenced by the members' attitudes and capacities, such as existing skills and knowledge. Members' skill and knowledge embrace critical elements for collaboration such as the ability to work collaboratively with others and the ability to create and build effective programs (Foster-Fisherman et al., 2001).

Relational capacity concerns collaboration, which is about creating social relationships between members to make it possible to achieve the goal. Relational capacity means for example the ability to create a positive environment for the collaborative work (Foster-Fisherman et al., 2001).

Programmatic capacity means that the group must have the capacity to design and implement meaningful programs that have an impact on the community (Foster-Fisherman et al., 2001). The framework is developed in a social context where the term program means, for example, family programs that help families evolve positively. However, the correspondence to software programs is evident and the concept can be transferred to software development projects. The overall meaning is that there is a need for collaborative capacity that makes it possible to create a good product.

Organisational capacity is essential for the collaboration to survive. Organizational capacity is about how to organize members in a productive fashion, and how to engage members in the work tasks. Examples of organizational capacity are that the group must have sufficient recourses to complete the task and that there are formalized procedures guiding the work.

The framework also identifies strategies to build the different capacities. For example, to build member capacity, members should be supported by technical assistance and training and to access member capacity, member diversity should actively be supported (Foster-Fisherman et al., 2001). Relational capacity should be supported by helping the members identify and gather around a shared vision. Relational capacity must also be supported by creating a inclusive environment where the decision making is shared and the members' diverse needs are attended to (Foster-Fisherman et al., 2001). When it comes to organizational capacity the organization must ensure that good leaders are fostered, as good leadership is essential to success in collaboration. Another thing that is required to build organizational capacity is, for example, to have an efficient communication system (Foster-Fisherman et al., 2001).

The development of the toolkit presented in this thesis is an effort to build collaborative capacity in some aspects of the framework.

Not all critical elements in the framework are relevant for the evaluation since it is impossible for an expert group to evaluate for example the members' attitudes and motivation. The relevant elements in the framework are marked with bold text in Figure 10 : 2.

10.2 Defining the Scope of Evaluation

In this first loop of the design process, evaluation will be performed by an expert group of researchers. The researchers will discuss and judge the tools. We have four questions that would be interesting to evaluate:

1. Do the tools facilitate collaboration between end-users and developers?

2a. Are the concepts (constructs) of the tools (Categorization, Matrix, Usability Patterns, Design Patterns) appropriate as boundary objects?

2b. Do the tools have the right content?

3. How do the tools work in an industrial development project?

Expert evaluation puts some constraints on the kind of questions that can be discussed. Question (1) 'Do the tools facilitate collaboration between end-users and developers?' origins in the evaluation against requirements and what we can evaluate in an expert evaluation is the tools' *potential to influence positively the collaboration between end-users and developers*.

Question (2a) deals with the concepts of the tools as such. This means that questions are posed regarding the concepts of, for example, categorization and patterns. The concept of categorization as means to develop an understanding of a phenomena is known to be effective (Gershkoff-Stowe and Rakison, 2005) and therefore there is no need to evaluate the concept as such again. The same thing applies to patterns. Many authors have given evidence of the usefulness of patterns as a means of communication (Buschmann et al., 2007, Gamma et al., 1995) and for the transfer of knowledge (Lukosch and Schümmer, 2006, Schümmer et al., 2005, Schümmer and Slagter, 2004). The construct of the Matrix has its origin in cooperation with industry and should thus be regarded as valid in this first loop of the design process.

The next question (2b) questions the content of the tools, but since a boundary object by definition should be able to adapt to different settings it is the content and to some degree the instructions that will adapt to the situation. The conclusion is that the content is dependent on the situation. To answer this question and evaluate if the content of the tools is suitable for a specific situation, requires either an expert group, such as a group of practitioners, who can judge the tools from a common context, or a real setting or a setting close to reality. Since this chapter reports only on the first evaluation loop in the design process it is not possible to evaluate question (2b).

The third question (3) is posed when the artefact is evaluated against the environmental effects. It means, in terms of the tools presented in this thesis, that the tools are implemented in a real setting or a setting close to reality. The outcome is then analyzed and conclusions are drawn. In this case the tools are not implemented in a context, and this type of evaluation is therefore beyond the scope of this loop in the design process.

Accordingly the question left to answer in the evaluation session is:

Do the tools facilitate collaboration between end-users and developers?

To be able to answer this question, sub questions are derived from the framework of collaborative capacity. The next section will start with an overview of the evaluation setting and then continue with how the evaluation questions are obtained.

10.3 Applying the FCC on the Toolkit

Expert evaluation is widely used for usability assessment (Doubleday et al., 1997, Nielsen and Mack, 1994, Rosenbaum, 1989) especially for evaluating user interfaces. Expert evaluation is also used in the area of software architectures (Bosch, 2000). In an expert evaluation, usability specialists carry out an evaluation that combines analysis of a product's possibilities to be applied to a specific task, with in depth knowledge of general rules and norms in the area (Rosenbaum, 1989). The difference between expert evaluation and user testing is that while user testing of a product often makes visible the symptoms of what is wrong, expert evaluation points out the causes (Doubleday et al., 1997). Because of this, a combination of the two types of evaluation is preferable and in a future design loop user testing will be performed. Also, since different people notice different things, it is preferable if several experts evaluate a product (Nielsen and Mack, 1994).

The evaluation panel consist of four experts in user participation. Each expert has additional expertise in one or more areas such as, software development, pedagogy, psychology, usability testing and e-democracy.

The evaluation session lasted for 3.5 hours and was divided into two meetings.

The session started with a presentation of the background to the development of the tools, and the tools and their intended use were explained. The reason for the evaluation was also presented to ensure that all evaluators had the same base for evaluation. The evaluator was given an evaluation kit, with the tools to evaluate and the questions to discuss. Then the evaluators were given time to acquaint themselves with the material.

Thereafter the actual evaluation began. The author introduced the first question and all the evaluators had to present their opinions and were invited to motivate their standpoint and argue pros and cons. When the evaluators felt that they had reached the end of the discussion, the author invited the evaluators to make a quantitative judgment of how well the tools facilitate collaboration, in terms of the capability it is supposed to support. The evaluator had to pick a number on a scale from 1 to 5 where 5 is 'supports the capability very well.

The evaluation session was audio taped and the author also took notes during the session. After the evaluation the author checked and supplemented the notes by listening to the recording. The results from the evaluation were summarized

in a report together with the quantitative evaluation. The report was sent to the evaluators for members' check (Robson, 2002).

10.3.1 The Chain to Find Detailed Evaluation Questions

The framework of collaborate capacity can assist in elaborating the implicit requirements to focus on in the evaluation. The explicit requirements of the tools are neither precise nor measurable (Table 10 : 1). The requirements are purely qualitative and to make it possible to evaluate them in depth, the explicit requirements have to be mapped to the implicit requirements. The framework of critical elements of collaborative capacity can help guide that work.

	requirements
1	Common base for communication
2	Learning environment that makes it possible for the user to understand technical decisions and their consequences for use
3	Both parties take part in design decision
4	Consensus of trade-offs
5	Learn from experience
6	Shared Mental Models

Table 10 : 1 Requirements

For each explicit requirement we have determined the type of capacity it is related to and the sub-elements have then been examined, and the corresponding capacity element has been formulated as a question that can be discussed in depth in the evaluation. The chain to find the evaluation questions are visualized in Figure 10 : 3.

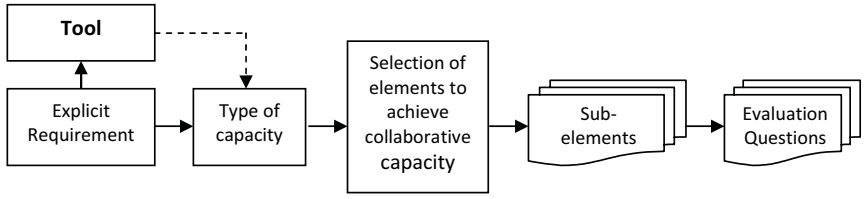


Figure 10 : 3 Chain to arrive at detailed evaluation questions

The different requirements resulted in different tools. These tools will be evaluated on the basis of different capacities and questions. In other words, explicit requirements resulted in a specific tool, and the requirements also indicate what type of capacity the tool should support, and in the long run this leads us to the evaluation questions to ask. The relationship between the explicit requirements and the tool and the type of capacity respectively are elaborated in Section 10.3.2.

The first requirement (common base for communication) (Table 10 : 1) is, for example, concerned with relational capacity. To achieve good relational capacity it is important to:

- develop a positive working climate,
- develop a shared vision,
- promote power sharing,
- value diversity, and
- develop positive external relationships.

Of these five elements, "develop a shared vision" corresponds to the requirement "common base for communication", as the purpose of a common base of communication is to develop a shared understanding of a phenomenon. The explicit requirements resulted in the Categorization and Matrix tools which make it possible to match the tools to evaluation questions (Figure 10 : 4).

The framework of critical elements of collaboration capacity also divides the different capacity elements into sub-elements. In this case, "develop a shared vision" is divided into:

- develop superordinate goals,
- develop shared solutions, and
- develop a common understanding of problems

These sub-elements correspond well to what we wanted to achieve with the tools. Evaluation questions were created from these sub-elements, (Figure 10 : 4).

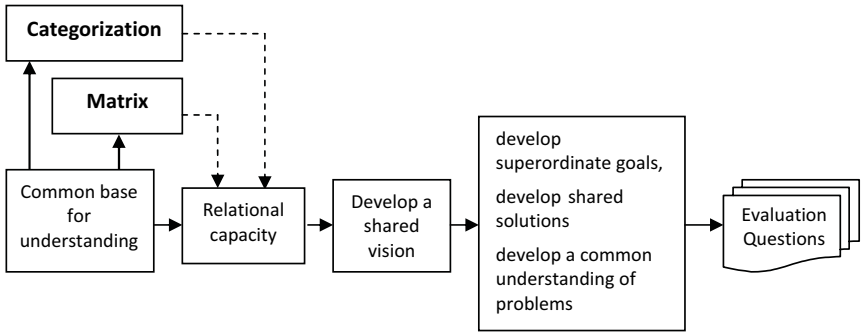


Figure 10 : 4 Example of mapping between the explicit requirements and the evaluation questions.

The critical elements that were found to be out of scope for the evaluation will be regarded as prerequisites and act as a base for the evaluation. For example we assume there is a positive working climate in the group and that diversity is valued.

10.3.2 Mapping Tools to Type of Capacity

In this section the relationship between the explicit requirements, tools and type of capacity is mapped out through a set of pictures.

Table 10 : 2 shows the relationship between the requirements and the tools.

	requirements	Tool
1	Common base for communication	Categorization Tool Matrix Tool
2	Learning environment that makes it possible for the user to understand technical decisions and their consequences for use	Usability Patterns Tool Design Pattern Tool
3	Both parties take part in design decision	
4	Consensus of trade-offs	
5	Learn from experience	
6	Shared Mental Models	

Table 10 : 2 Requirements in relation to the tools

Requirement (1) resulted in the Categorization and the Matrix tool. Some relational capacities can be derived from the explicit requirement and therefore the Categorization and Matrix tools can be mapped to some critical elements that they should support. (Figure 10 : 5).

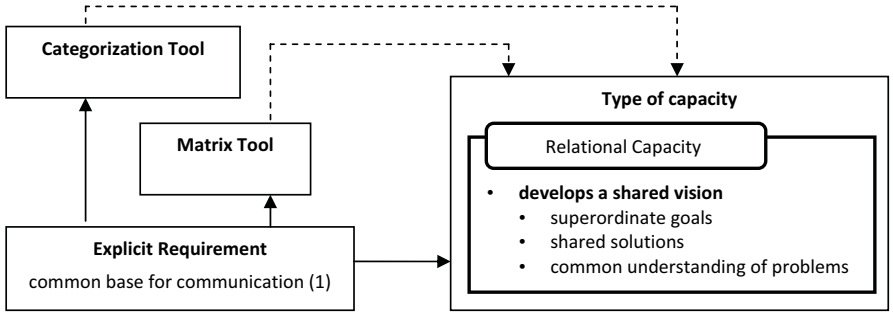


Figure 10 : 5 Overview of the relationship between the Categorization and Matrix tools and the capacities they should support

Requirements (2)-(6) gave rise to the Usability Pattern and Design Pattern tool and the tools can thereby be mapped to a set of critical elements of collaborative capacity (Figure 10 : 6).

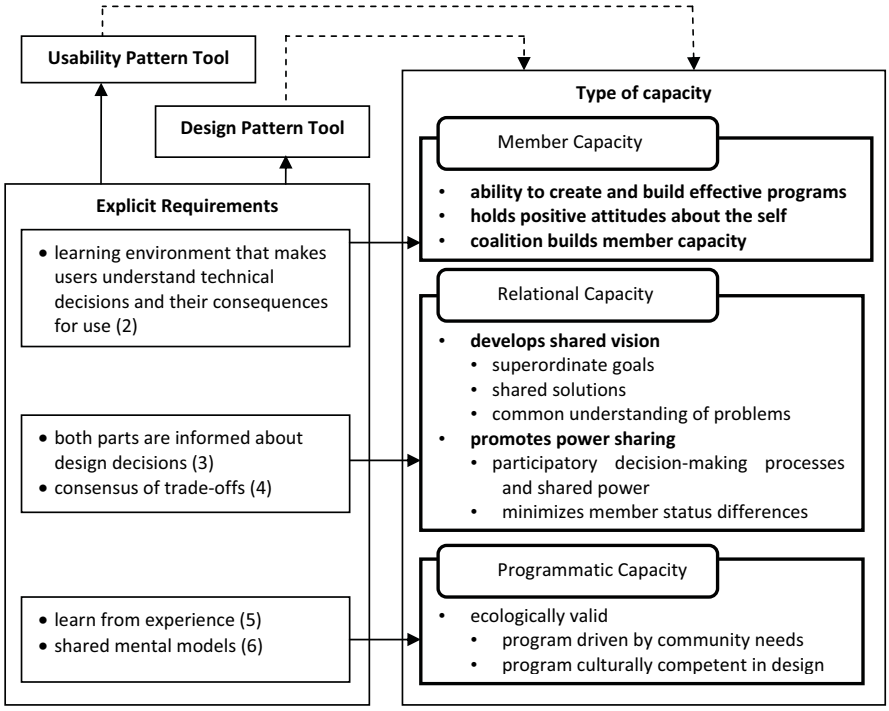


Figure 10 : 6 Overview of the relationship between the Usability Pattern and Design Pattern tools and the capacities they should support

In addition to what has been discussed above, there is one type of capacity, organizational capacity (Figure 10 : 7), that has not yet been included. As the tools are intended to be freestanding from the overall development process, organizational capacity is not a major concern in the context.

The process of use is a part of the tool itself and accordingly will be evaluated along with the tools, but the BoundLets should be evaluated separately too, to determine if a BoundLet provides the tool with the necessary affordance.

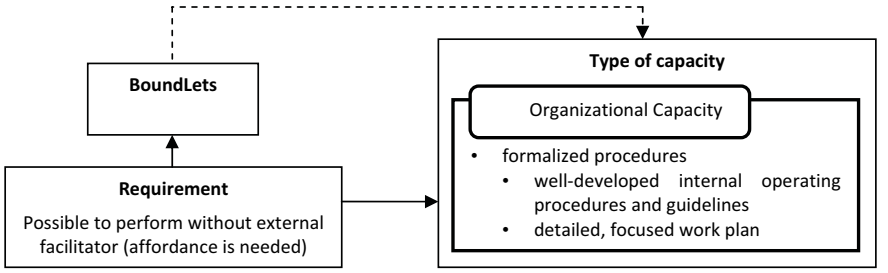


Figure 10 : 7 Overview of the relationship between the BoundLets and the capacities they should support

10.3.3 Resulting Evaluation Questions

The subset of critical elements from Figure 10 : 5, Figure 10 : 6 and Figure 10 : 7 is a base for the evaluation questions. There are a couple of complementary questions in the list below. They are marked in italics. Each of the questions is accompanied by the follow up question: What can be improved?

Categorization and Matrix Tool

Relational capacity

To what degree does the tool contribute to create a shared vision in terms of…

…superordinate goals

…shared solutions

…a common understanding of problems

Usability pattern and Design Pattern Tool

Member capacity

To what degree does the tool contribute to…

…the ability to create and build good programs?

…an increase in positive attitudes about the self?

…make it possible for a coalition to build member capacity?

Does the pattern structure contain what is needed to support both end-users and developers? Is there a good balance?

Relational capacity

To what degree does the tool contribute to create a shared vision in terms of…

…superordinate goals

…shared solutions

…a common understanding of problems

To what degree does the tool contribute to promoting power sharing in terms of…

…participatory decision-making processes and shared power

…minimizing differences in member status

Programmatic capacity

To what degree does the tool contribute to building good software in terms of ...

…a program driven by community needs

…a program culturally competent in design

BoundLets

Organizational capacity

To what degree do the BoundLets correspond to formalized procedures in terms of…

…well-developed internal operating procedures and guidelines

…a detailed, focused work plan

Do the BoundLets provide for affordances?

10.4 Result

In this section the outcome of the evaluation is presented. Each sub section starts with a repetition of the questions discussed for each tool along with the quantitative judgement. Thereafter the qualitative assessment is presented and each section ends with a table of suggested improvements. The five level scale used for the quantitative judgement is translated into plusses (1='+', 2='++', 3='+++', 4='++++', 5='+++++') for easier visualization.

The overall results are summarized in Section 10.5.

10.4.1 Categorization Tool

In this section the *Categorization tool* is evaluated.

Relational Capacity

To what degree does the tool contribute to creating a shared vision in terms of…	Evaluation
· superordinate goals?	n.a
· shared solutions?	n.a
· a common understanding of problems?	++++

Figure 10 : 8 Relational capacity questions and evaluation of Categorization tool

The evaluation group did not find the tool to be applicable in terms of creating a subordinate goal or shared solutions, but found that the tool is very good as a basis for creating a common understanding of a phenomenon such as end-user tailoring.

The quantitative assessment is shown in Figure 10 : 8.

Summary of expert group's comments: The evaluators found that the tool is suitable as a basis for developing common concepts, since the resulting definition is made visible for all. They also thought it was valuable for the participants to be forced to motivate their opinions since this advances the process. The evaluators also expressed the opinion that it is essential to start with some kind of tool as this allows both immature and more mature groups to reach a joint explanation through discussion. The evaluators also approved of the fact that it was clear that the use of the tool must end with an agreement on a

specific definition to be used by group as a basis for discussions of tailorable software. However, the discussions also revealed the need to be observant of the fact that it is impossible to reach a total consensus, since consensus is something of a utopia. It is important that the problems inherent in the resulting definition, and disagreements among the participants, are collected and written down, thereby making them visible. This will increase the usefulness of the definition, by making all participants aware that the definition might not suit everyone, but that in order to have a common base, there is an agreement on which definition to use, and the participants act in accordance with this decision. *Summary of expert group's comment:* Even though it is possible to discuss end-user tailoring in general terms it was stated that it would be easier to use the tool if there was a case that could be used as a starting point for the discussions.

Improvements

The suggested improvements are listed in Table 10 : 3. The proposed improvements are implemented in the tool presented in Chapter Nine. The part of the BoundLet where the improvement is made is shown in the right hand column of the table. Some improvements affect documents outside the BoundLet and they are marked in italics. All documents are updated and can be found in Appendix B.

Suggested Improvement	attended to in
Clarify that the tool can be used detached from a specific case.	"Choose this BoundLet…"
Clarify when the Categorization is used in the process.	"Instructions"
Clarify that if the descriptions of the categories are imprecise for the situation they can be rephrased to suit the specific circumstances better.	"Instructions"
Point out that it is important to write down differences in opinions regarding the definition the group has agreed upon.	*BoundLetExtra: CreateAgreement*

Table 10 : 3 Suggested improvements to the Categorization tool

10.4.2 Matrix Tool

In this section the *Matrix tool* is evaluated.

Relational Capacity

To what degree does the tool contribute to creating a shared vision in terms of……	Evaluation
· superordinate goals?	+++
· shared solutions?	n.a
· a common understanding of problems?	+++

Figure 10 : 9 Relational capacity questions and evaluation of the Matrix tool

Summary of expert group's comments: The evaluators agreed that the tool functions as a guide to get on the track of the types of tailoring to consider for an application, but the intent is not to state which type should be chosen for a specific situation. The intention is to make the participants understand which compromises must be made. The value of the tool as a key to which tailorability to implement is hard to estimate. The tool can help the participants gain an understanding of the goal, which can support finding a common view of what to achieve. The tool cannot be of assistance in finding a solution to a problem, but the discussion it gives birth to was found to be practical, since the participants gain an understanding of the case. In other words, the tool promotes an understanding of the problem, and conflicts in the definition of tailoring are revealed and tensions are made visible.

The quantitative assessment is shown in Figure 10 : 9.

Improvements

The proposed improvements are listed in Table 10 : 4. These improvements are implemented in the tool presented in Chapter Nine. The part of the BoundLet where the improvement is made is shown in the right hand column of the table. Some improvements affect documents outside the BoundLet and they are marked in italics. All documents are updated and can be found in Appendix C.

Suggested Improvement	attended to in
Point out that the tool must be used in the context of a case.	"Choose this BoundLet…"
Use questionnaires to note individual, as well as the group's, opinions before comparing with the matrix. An empty matrix with only one column should be used.	*New documents*
Sum up the number of correspondences between the group's matrix and each category of tailoring. For example, the group's one column matrix corresponded in three places with the column for customization in the matrix, and so on. This makes it easier to consider which type of tailoring to use in the application.	*In the new documents*
The title should mirror the activity and purpose of the tool – Flexibility Dilemmas	Title of BoundLet

Table 10 : 4 Suggested improvements to the Matrix tool

10.4.3 Usability Pattern Tool

In this section the *Usability Pattern tool* is evaluated.

Summary of expert group's comments: The evaluators agreed that the goal of this tool is to make the application more usable and that flexibility is of secondary interest in this tool. The tool invites the participant to a learning situation concerning patterns and it is important that the tool is understandable, since it targets untrained users and developers. The tool was perceived as rather

complex with many different documents to handle and understand. But it was also stated that it is probably worth the effort, compared to starting to build the wrong thing. The tool supports discussions with the intention of systematically discussing relationships between and consequences of different actions. The evaluators pointed out that this kind of tool is unnecessary in mature groups, but that groups dissolve, and new members join the group, and unwritten rules and agendas sink into oblivion. Then it is good to be able to fall back on the kinds of tools discussed here. One of the rules in the tool is that everyone must listen to and consider other members' opinions. The evaluation group pointed out that it is essential that this process embraces deliberation, where different opinions are brought together and actively used to advance the process.

Member Capacity

To what degree does the tool contribute to …	Evaluation
• the ability to create and build good programs?	+++
• increasing positive attitudes about the self?	++++
• making it possible for the coalition to build member capacity?	++++
• *Does the pattern structure contain what is needed to support both end-users and developers? Is there a good balance?*	n.a

Figure 10 : 10 Member capacity questions and evaluation of Usability Pattern tool

Summary of expert group's comments: The tool contributes to the ability to create and build good programs as it supports a systematic walkthrough of the patterns. The assembled knowledge and information improves the design, but the choice is of course still open and there is no guarantee that the program will be better simply because the tool has been used. The tool provides a structured way of working with a problem. The tool also contributes to the participants daring talk about these kinds of issues. It is important for the users, especially untrained users, to grasp the terminology. It is also an advantage that everything concerning usability patterns is gathered together, and the package can be used without having to search for scattered information. Regarding whether there is a good balance in the pattern structure between user and developer support, the evaluators stated that it does not hinder it, but that it is hard to be specific, since it depends on how the patterns are formulated and specified.

The quantitative assessment is shown in Figure 10 : 10.

Relational Capacity

To what degree does the tool contribute to …	
…building a shared vision in terms of…	**Evaluation**
· building superordinate goals?	n.a
· building shared solutions?	++++
· building a common understanding of problems?	++++
…promoting power sharing in terms of…	
· participatory decision-making processes and shared power?	+++
· minimizing differences in member status?	+++

Figure 10 : 11 Relational capacity questions and evaluation of the Usability Pattern tool

Summary of expert group's comments: Patterns are concrete and goals are so much larger and more abstract. The tool lets the members of the group learn about the product from the perspective of relationships and consequences, but the goal is a good product, and this cannot be verified before the product is deployed. This is a long chain and there is no explicit course of action. The tool lets the participants understand what they are doing, but it does not ensure a good product. However, this should increase the chance of achieving a satisfactory product. It is difficult to state how well the tool promotes power sharing, since so many factors influence how power is shared among the participants. The organization of the company as well as the structure within the group influence power sharing, but the tool can act as a foundation of power sharing as it educates users in areas previously reserved for developers, and the users are invited to discuss and participate in decision making.

The quantitative assessment is shown in Figure 10 : 11.

Programmatic Capacity

To what degree does the tool contribute to building good software in terms of…	
	Evaluation
· software driven by community needs?	++++
· culturally suitable software?	++++

Figure 10 : 12 Programmatic capacity questions and evaluation of Usability Pattern tool

Summary of expert group's comments: It was agreed that the community in this case was the users. The evaluators expressed the opinion that the users' needs are well provided for. And as one of the evaluators put it "If it doesn't work for the users, then it's unprofitable".

The quantitative assessment is shown in Figure 10 : 12.

Improvements

The proposed improvements are listed in Table 10 : 5. These improvements are already implemented in the tool presented in Chapter Nine. The part of the BoundLet where the improvement is made is shown in the right hand column of the table. Some improvements affect documents outside the BoundLet and they are marked in italics. All documents are updated and can be found in Appendix D.

Suggested Improvement	attended to
Clarify that it is a prerequisite that the type of tailoring is chosen beforehand.	”Choose this BoundLet…”
“Everyone should make their voice heard” should be exchanged by “take a turn around the table and collect the participants’ point of view.	”Rules”
The title should mirror the activity and purpose of the tool –Selecting Usability Pattern	Title of the BoundLet
Clarify if you should start with usability patterns or usability scenarios.	”Instructions”
The instruction has to be clear that you must handle the vital patterns first and then choose the scenarios you think are important. There should be a description of how this is done.	”Instructions”
The patterns should be prioritized. (except for the vital patterns)	”Instructions”
The instructions must be simpler and clearer.	”Instructions”
The table of the vital patterns must be simplified and the categories must be separated from each other.	*Vital Usability patterns*
The vital usability scenarios should be put first in the list of usability scenarios	*Usability Scenarios*
A workflow is required showing how and in which order the different documents should be used.	“Overview”
Clarify how to access the different parts in the pattern structure, but there should not be questions in every section in the pattern structure as this makes the process too controlled.	“Instructions”
Explanations are needed of usability scenarios and corresponding usability patterns	In the next evaluation loop.

Table 10 : 5 Suggested improvements to Usability Pattern tool

10.4.4 Design Pattern Tool

In this section the *Design Pattern tool* is evaluated.

Summary of expert group’s comments: If a group is not sufficiently mature, it may choose not to use this tool even though it uses the other tools in the toolkit. This tool is intended for groups that are used to patterns and where the users are interested in learning about the techniques of the application and participating in the technical decision making. The BoundLet is named *Technical trade-offs* alluding to the intention of the tool, which is to elucidate and discuss trade-offs

and make informed design decisions. Extending the tailoring capabilities of a tailorable application leads to many trade offs, as previous design decisions and other factors in the environment influence what can be done and which choices can be made. The tool is intended for such situations.

Member Capacity

To what degree does the tool contribute to …	Evaluation
• the ability to create and build good programs?	+++
• increasing positive attitudes about the self?	++++

Figure 10 : 13 Member capacity questions and evaluation of Design Patterns tool

The Usability and Design Pattern tools are quite similar and have similar intent, but the Design pattern tool is at another maturity level. It requires more specific interest and dedication to learn about the 'invisible' technology in the application. The evaluation result for member capacity was therefore similar.

The quantitative assessment is shown in Figure 10 : 13.

Relational Capacity

To what degree does the tool contribute to …	
…building a shared vision in terms of…	**Evaluation**
· building superordinate goals?	n.a
· building shared solutions?	++++
· Building common understanding of problems?	++++
…promoting power sharing in terms of…	
· participatory decision-making processes and shared power?	++++
· minimizing member status differences?	++++

Figure 10 : 14 Relational capacity questions and evaluation of Design Pattern tool

Just as for member capacity, the evaluation of relational capacity provided by the Design Pattern tool is similar to the evaluation of the Usability Pattern tool. There was however a difference when it came to shared power. The Design Pattern tool was to a higher degree perceived as promoting shared power since it is so obvious that we deal with technical issues here, and the knowledge is shared by users and developers, and thereby the power is also shared to a greater extent.

The quantitative assessment is shown in Figure 10 : 14.

Programmatic Capacity

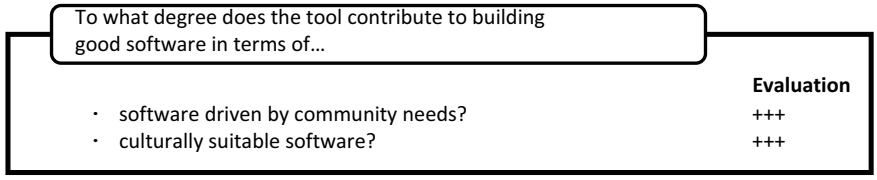


Figure 10 : 15 Programmatic capacity questions and evaluation of Design Patterns tool

Summary of expert group's comments: in comparison to the Usability Pattern tool, community needs and culture are less in focus in this tool as this tool is aimed more at architectural solutions. But the solution is still based on the users. The expert group thought that fewer users could be expected to participate at this level.

The quantitative assessment is shown in Figure 10 : 15.

Improvements

The proposed improvements are listed in Table 10 : 6. These improvements are already implemented in the tool presented in Chapter Nine. Where the improvement is made in the BoundLet is shown in the right hand column of the table. Some improvements affect documents outside the BoundLet and they are marked in italics in the table. All documents are updated and can be found in Appendix E.

Suggested Improvement	attended to in
The patterns should be accompanied by a metaphor so that the participants can choose patterns via the metaphor.	Instructions and in the *pattern structure*
Instead of evaluating one pattern at a time, the participants should choose some promising patterns, study them and then agree upon which pattern or patterns to use.	Instructions

Table 10 : 6 Suggested improvements to Design Pattern tool

Instead of coming up with a metaphor as initially suggested in the BoundLet, the evaluators suggested that the metaphor should be a part of the pattern structure. It takes time to come up with a metaphor and it is much easier to choose from a collection. Metaphors are used in XP (eXtreme Programming) (Beck, 1999) and it has been shown to be difficult to come up with useful metaphors (Wake, 2002).

The participant must choose some potential patterns based on the metaphor. Then they work through the patterns to get a feeling for them. Then the selected patterns are evaluated and a decision is taken about which pattern or patterns to use.

10.4.5 BoundLets

In this section the concept of BoundLets is evaluated in terms of organizational capacity. The improvements to the BoundLets are presented in Sections 10.4.1-10.4.4 above.

Organizational Capacity

To what degree does the BoundLet correspond to formalized procedures in terms of…	Evaluation
· well developed internal operating procedures and guidelines?	++++
· detailed, focused work plan?	++
• *Do the BoundLets provide for affordance?*	+++++

Figure 10 : 16 Organizational capacity questions and evaluation of BoundLets

Summary of expert group's comments: The evaluators agreed that the BoundLets were formalized procedures. It would be overwhelmingly complex to discuss the issues concerned without the formalization. You would not know where to start in such a difficult area. The conclusion was that the BoundLets are well-developed internal operating procedures and guidelines. The BoundLets have loose coupling to an overall process as they can be used in any process, which is why they score so low in terms of detailed, focused work plans. The BoundLets were perceived as providing for good affordance as it would have been impossible to use the different boundary objects without the BoundLets, as the tools are complex in the sense that they consist of many different parts.

The quantitative assessment is shown in Figure 10 : 16.

10.5 Summary and Future Work

The summary (Table 10 : 7) shows that the Usability and Design Pattern tools are equally good at providing for member capacity, while the Design Pattern tool is perceived as better at promoting power sharing. The Usability Pattern tool is superior when it comes to providing adequate software for the community, which might be regarded as a somewhat surprising result.

The table shows that the toolkit supported all of the critical elements of collaborative capacity that were the subject for evaluation. There were some elements that were found not to be applicable in the context of the Categorization and Matrix tool, but in those cases the Usability pattern and Design Pattern tools have the capability, and vice versa. Additionally the Categorization tool and the Matrix tool together supported a shared vision, although they do not do this separately.

To what degree does the tool contribute to..	Categorization tool	Matrix tool	Usability Pattern tool	Design Pattern tool
..creating a shared vision in terms of superordinate goals?	n.a	++++	n.a	n.a
..creating a shared vision in terms of shared solutions?	n.a	n.a	++++	++++
..creating a shared vision in terms of a common understanding of a problem?	++++	+++	++++	++++
…the ability to create and build good programs?			+++	+++
…increasing positive attributes about the self?			++++	++++
…making it possible for a coalition to build member capacity?			++++	++++
…promoting power sharing in terms of participatory decision making processes and shared power?			+++	++++
…promoting power sharing in terms of minimizing member status differences?			+++	++++
…building good software in terms of software driven by community needs?			++++	+++
…building good software in terms of culturally suitable software?			++++	+++

Table 10 : 7 Summary of the quantitative assessment

The expert evaluation resulted in a set of concrete proposals for improvements that should be made to the tools. The evaluation also concluded that the tools have the potential to influence positively the collaboration between end-users and developers since they for example provide a common base for discussions and shared terminology.

The toolkit must go through additional evaluations and improvements. For example the toolkit should be evaluated by practitioners in an experiment close to a real world setting and then be tested in two or three real projects with different maturity levels in terms of user participations, so that the results can be compared and improvements can be made with different target groups in mind.

There is also an open question of how to implement the tools in the organization. Perhaps the toolkit should remain low tech, but it is also possible that the tools could be encapsulated in a cooperative IT-system. Another interesting alternative to look into in future research would be to represent the toolkit both physically and digitally so that the participants work with physical objects but reflections and decisions are collected and stored digitally with minimal cognitive overhead.

References

Alexander, C., Ishikawa, S., Silverstein, M., et al. (1977): *A Pattern Language*, 1st edition, Oxford University Press, New York, USA.

Association of Information Systems, IS World, V. Vaishnavi, V., and B. Kuechler (Eds.): 'Design Research in Information Systems', <http://www.isworld.org /Researchdesign /drisISworld.htm)>, (17 November, 2007).

Barret, R. and Maglio, P. P. (1998): Informatic Things - how to attach information to the real world, in *Proceedings of the Symposium on User Interface Software and Technology (UIST'98)*, San Francisco, CA, USA, pp. 81-88.

Bass, L. and John, B. E. (2001): *Achieving Usability through Software Architecture*, Technical Report. CMU/SEI-2001-TR-005, Carnegie Mellon, Software Engineering Institute, Pittsburgh, PA, USA, March 2001.

Bass, L. and John, B. E. (2003): Linking usability to software architecture patterns through scenarios, *Journal of Systems and Software,* 66(2003), pp. 187-197.

Bass, L., Clements, P. and Kazman, R. (1998): *Software Architecture in Practice*, 1st edition, Addison Wesley, Chichester, UK.

Bauersfeld, P., Bennet, J. and Lynch, G. (1992): Striking a balance, in *Proceedings of the Conference on Human Computer Interaction (CHI'92)*, Monterey, CA, USA.

Beck, K. (2005): *Extreme Programming Explained*, 2nd edition, Addison-Wesley, Massachusetts, MA, USA.

Bennett, K. and Rajlich, V. (2001): Software evolution, in *Proceedings of IEEE International Conference on Software Maintenance (ICSM'01)*, position paper for panel discussion, Florence, Italy, pp. 4.

Bennett, K. H. and Rajlich, V. T. (2000): Software maintenance and evolution - a roadmap, in *Proceedings of the Conference on the Future of Software Engineering*, Limerick, Ireland, pp. 73-87.

Beyer, H. and Holtzblatt, K. (1997): *Contextual Design - Defining Customer-Centered Systems*, 1st edition, Morgan Kaufmann Publishers, Inc., San Fransisco, USA.

Bleek, W. G. (2004): *Software-Infrastruktur: Von Analytischer Perspektive zu Konstruktiver Orientierung*, 1st edition, Hamburg University Press, Hamburg, Germany.

Blomberg, J. and Giacomi, J. (1993): Ethnographic field methods and their relation to design, in *Participatory Design; Principles and Practices*, D. Schuler and A. Namioka (Ed.), 1st edition, Lawrence Erlbaum Associates, Hillsdale, New Jersey; USA, pp. 123-156.

Blomberg, J., Suchman, L. and Trigg, R. H. (1996): Reflections on a work-oriented design project, *Human-Computer Interaction,* 11(3), pp. 237-265.

Boehm, B. W., Brown, J. R. and Lipov, M. (1978): *Characteristics of Software Quality*, 1st edition, Elsevier North-Holland Publishing Company Inc., Amsterdam, Netherlands.

Borchers, J. (2001): *A Pattern Approach to Interaction Design*, 1st edition, John Wiley & Sons Ltd, Chichester, UK.

Bosch, J. (2000): *Design and Use of Software Architectures - Adopting and Evolving a Product Line Approach*, 1st edition, Pearson Education (Addison-Wesley and ACM Press), Reading, MA, USA.

Bowker, G. C. and Star, S. L. (1999): *Sorting Things Out - Classification and its Consequences*, 1st edition, MIT Press, Cambridge, MA, USA.

Briggs, R. O., de Vreede, G.-J. and Nunamaker, J. F. (2003): Collaborating Engineering with ThinkLets to pursue sustained success with Group Support Systems, *Journal of Management Information Systems,* 19(4), pp. 31-64.

Brown, J. S. and Duguid, P. (2000): *The Social Life of Information*, 1st edition, Harward Business School Press, Boston, MA, USA.

Burnett, M., Rothermel, G. and Cook, C. (2003): Software Engineering for end-user programmers, in *Proceedings of the Conference on Human Factors in Computing Systems (CHI'03)*, Fort Lauderdale, Florida, USA, pp. 12-15.

Buschmann, F., Henney, K. and Schmidt, D. C. (2007): *Pattern-Oriented Software Architecture - On Patterns and Pattern Language*, 1st edition, John Wiley & Sons Ltd, Chichester, UK.

Buschmann, F., Meunier, R., Rohnert, H., et al. (1996): *Pattern-Oriented Software Architecture - A System of Patterns.*, 1st edition, John Wiley & Sons Ltd, Chichester, UK.

Bødker, S. (1999): *Computer Applications as Mediators of Design and Use - a Developmental Perspective,* Doctoral Dissertation, DAIMI PB-542, Computer Science Department, Aarhus University, Aarhus, Denmark.

Bødker, S., Grønbæk, K. and Kyng, M. (1993): Cooperative design: techniques and experiences from the Scandinavian scene, in *Participatory Design; Principles and Practices*, D. Schuler and A. Namioka (Ed.), 1st edition, Lawrence Erlbaum Associates, Hillsdale, New Jersey; USA, pp. 157-176.

Canfield Smith, D., Cypher, A. and Tesler, L. (2000): Novice programming comes of age, *Communication of ACM,* 43(3), pp. 75-81.

Capron, H. L. (2004): *Computers - Tools for an Information Age*, 8th edition, Addison-Wesley, Reading, MA, USA.

Carroll J.M., Rosson, M. B., Chin G., et al. (1998): Requirements development in scenario-based design, *IEEE Transactions on Software Engineering,* 24(12), pp. 1156-1170.

Carroll, J. M. (1995): *Scenario-Based Design - Envision Work and Technology in System Development*, 1st edition, John Wiley & Sons Ltd., New York, USA.

Carter, K. and Henderson, A. (1999): Tailoring culture, in *Proceedings of the 13th Information Systems Research Seminars (IRIS'13)*, Åbo Akademi University, Finland, pp. 103-116.

Chan, D. K. C. (1998): A document-driven approach to database report generation, in *Proceedings of the 9th International Workshop on Database and Expert Systems Applications*, Le Chesnay, France, pp. 925-930.

Clement, A. and Besselar, P. V. d. (1993): A retrospective look at PD projects, *Communication of the ACM,* special issue on Graphical User Interfaces - the Next Generation, 36(4), pp. 29-37.

Cook, S., Harrison, R., Lehman, M. M., et al. (2006): Evolution in software systems: foundations of the SPE classification scheme, *Journal of Software Maintenance and Evolution: Research and Practice,* 2006(18), pp. 1-35.

Costabile, M. F., Fogli, D., Mussion, P., et al. (2006): End-user development - the software shaping workshop approach, in *End User Development*, H. Lieberman, F. Paternò and V. Wulf (Ed.), 1st edition, Springer, Netherlands, pp. 183-205.

Dearden, A., Finlay, J., Allgar, E., et al. (2002): Using pattern languages in Participatory Design, in *Proceedings of the Participatory Design Conference*, Malmö, Sweden, pp. 104-112.

DePaula, R. (2004): Lost in Translation - A critical analysis of actors, artifacts, agendas, and arenas in Participatory Design, in *Proceedings of the Participatory Design Conference 2004*, Toronto, Canada, pp. 162-172.

Diestelkamp, W. (2002): *On Design Methodology for Flexible Systems,* Licentiate thesis, Department of Software Engineering and Computer Science, Blekinge Institute of Technology, Ronneby.

Dittrich, Y. and Lindeberg, O. (2002): Designing for changing work and business practices, in *Evolutionary and Adaptive Information Systems*, N. Patel (Ed.), *1st* edition, IDEA Group Publishing, USA, pp. 152-171.

Dittrich, Y., Lundberg, L. and Lindeberg, O. (2006): End-user development as adaptive maintenance, in *End User Development*, H. Lieberman, F. Paternò and V. Wulf (Eds.), 1st edition, Springer Verlag, Netherlands, pp. 295-313.

Dittrich, Y., Rönkkö, K., Eriksson, J., et al. (2007): Co-operative Method Development – combining qualitative empirical research with process improvement, accepted for the *Empirical Software Engineering Journal*, published online December 2007, <http://www.springerlink.com/content/712m872162v41l86/?p=5f045b7d307f4f379423ac3e07a4af64&pi=0>

Doubleday, A., Ryan, M., Springett, M., et al. (1997): A comparison of usability techniques for evaluating design, in *Proceedings of the 2nd Conference on Designing Interactive Systems - Processes, Practices, Methods, and Techniques (DIS '97)*, Amsterdam, Netherlands, pp. 101-110.

Dourish, P. (1996): *Open Implementation and Flexibility in CSCW Toolkits,* Doctoral Dissertation, London University College, London, UK.

Ehn, P. and Kyng, M. (1991): Cardboard computers: mocking-it-up or hands on the future, in *Design at Work - Cooperative Design of Computer System*, J. Greenbaum and M. Kyng (Eds.), 1st edition, Lawrence Erlbaum Associates, Hillsdale, New Jersey, USA, pp. 139-154.

Ely, M., Anzul, M., Friedman, T., et al. (1993): *Kvalitativ Forskningsmetodik i Praktiken - Cirklar inom Cirklar*, 1st edition, Studentlitteratur, Lund, Sweden (in Swedish).

Ericsson, K. A. and Simon, H. A. (1993): *Protocol Analysis - Verbal Reports as Data*, 1st edition, MIT Press, Cambridge, MA, USA.

Farooq, U., Merkel, C. B., Nash, H., et al. (2005): Participatory Design as apprenticeship: sustainable watershed management as community computing application, *Proceedings of the 38th Hawaii International Conference on System Science*, Hawaii, USA, pp. 178c-187c.

Ferre, X., Jusisto, N., Moreno, A. M., et al. (2003): A software architectural view of usability patterns, *Proceedings of INTERACT 2003*, Zürich, Switzerland.

Fischer, G. (2001): Communities of Interest - learning through the interaction of multiple knowledge systems, in *Proceedings of the 24th annual Information Systems Research Seminar in Scandinavia (IRIS 24)*, Ulvik, Norway, pp. 1-14.

Fischer, G. (2003): Meta-Design - beyond user-centered and participatory design, in *Proceedings of the 10th International Conference on Human-Computer Interaction (HCI 2003)*, Crete, Greece, pp. 88-92.

Fischer, G. and Girgensohn, A. (1990): End-user modifiability in design environments, in *Proceedings of the conference on Human Factors in Computing Systems, CHI'90*, Washington, USA, pp. 183-192.

Fischer, G. and Ostwald, J. (2001): Problems, promises, realities, and challenges, *IEEE Intelligent Systems*, January/February, pp. 60-72.

Fischer, G. and Ostwald, J. (2002): Seeding, evolutionary growth, and reseeding - enriching Participatory Design with Informed Participation, in *Proceedings of Participatory Design Conference (PDC'02)*, Malmö University, Sweden, pp. 135-143.

Fischer, G., Giaccardi, E., Eden, H., et al. (2005): Beyond binary choices - integrating individual and social creativity, *Human-Computer Studies,* 63(2005), pp. 482-512.

Fischer, G., McCall, R., Ostwald, J., et al. (1994): Seeding, evolutionary growth and reseeding - incremental development of collaborative design environments, in *Proceedings of Human Factors in Computing Systems, CHI'94*, pp. 292-298.

Fleischmann, K. R. (2006): Do-it-yourself information technology - role hybridization and the design-use interface, *Journal of the American Society for Information Science and Technology,* 57(1), pp. 87-95.

Floyd, C. (1984): A systematic look at prototyping, in *Approaches to Prototyping*, R. Budde, K. Kuhlenkamp, L. Mathiassen and H. Zuellighoven (Eds.), 1st edition, Springer-Verlag, Berlin, Germany, pp. 1-18.

Folmer, E. and Bosch, J. (2003.): Usability patterns in software architecture, in *Proceedings of the 10th International Conference on Human-Computer Interaction (HCI 2003)*, Crete, Greece, pp. 93-97.

Foster-Fisherman, P. G., Berkowitz, S. L., Lounsbury, D. W., et al. (2001): Building collaborative capacity in community coalitions - a review and integrative framework. American Journal of Community Psychology, 29(2), pp. 241-261.

Gamma, E., Helm, R., Johnsson, R., et al. (1995): *Design Patterns - Elements of Reusable Object-Oriented Software*, 26th edition, Addison-Wesley, Indianapolis, USA.

Gantt, M. and Nardi, B. A. (1992): Gardeners and gurus - patterns of cooperation among CAD users, in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, Monterey, California, USA, pp. 107-117.

Gasson, S. (2003): Human-centered vs. user-centered approaches to information system design, *JITTA: Journal of Information Technology Theory and Application,* 5(2), pp. 29-46.

Gershkoff-Stowe, L. and Rakison, D. H. (Eds.) (2005): *Building Object Categories in Developmental Time*, 1st edition, Lawrence Erlbaum, Philadelphia, PA, USA.

Gerson, K. and Horowitz, R. (2002): Observation and interviewing - options and choices in qualitative research, in *Qualitative Research in Action*, T. May (Ed.), 1st edition, SAGE Publishers, Trowbridge, Wiltshire, UK, pp. 199-224.

Golm, M. (1997): *Design and Implementation of Meta Architecture for Java,* Diplomarbeit, Friedrich-Alexander-Universität, Erlangen-Nürnberg, Germany.

Goodman, R. M., Speers, M. A., Mcleroy, K., et al. (1998): Identifying and defining the dimensions of community capacity to provide a basis for measurements, *Health Education & Behavior*, 25(6), pp. 258-278.

Greenberg, S. and Fitchett, C. (2001): The phidget architecture - rapid development of physical user interfaces, in *Proceedings of the UbiTools'01 Workshop on Application Models and Programming Tools for Ubiquitous Computing at UBICOMP'2001*, Atlanta, USA.

Grønbæk, K., Kyng, M. and Morgensen, P. (1997): Toward a cooperative experimental systems development approach, in *Computers and Design in Context*, M. Kyng and L. Mathiasen (Ed.), 1st edition, MIT Press, Cambridge, MA, USA, pp. 201-238.

Henderson, A. and Kyng, M. (1991): There's no place like home - continuing design in use, in *Design at Work*, J. Greenbaum and M. Kyng (Eds.), 1st edition, Lawrence Erlbaum, Hillsdale, NJ, USA, pp. 219-240.

Hevner, A. R., March, S. T., Park, J., et al. (2004): Design science in information systems research, *MIS Quarterly,* 28(1), pp. 75-105.

Huang, A. C., Ling, B. C., Barton, J., et al. (2001): Making computers disappear - appliance data service, in *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking*, Rome, Italy, pp. 108-121.

Hummes, J. and Merialdo, B. (2000): Design of extensible component-based groupware, *Computer Supported Cooperative Work (CSCW),* 9(1), pp. 53-74.

Ilvari, J. and Iivari, N. (2006): Varieties of user-centeredness, in *Proceedings of the 39th Annual Hawaii International Conference on System Sciences, HICSS '06*, Hawaii, USA, pp. 176a-186a.

Irving, C. W. and Eichmann, D. (1996): Patterns and design adaptability, *Pattern Languages of Programs,* 2(1996), pp. 1-10.

ISO *ISO/IEC 9126 Information Technology - Software Quality*, International Standard Organization.

Jacobsen, K. and Johansen, D. (1999): Ubiquitous devices united - enabling distributed computing through mobile code, in *Proceedings of the 1999 ACM Symposium on Applied Computing*, San Antonio, Texas USA, pp. 399-404.

Jacobson, I., Griss, M. and Jonsson, P. (1997): *Software Reuse; Architecture, Process and Organization for Business Success*, 2nd edition, Addison Wesley Longman Limited, Palantino, USA.

John, B. E., Bass, L., Kazman, R., et al. (2004): Identifying gaps between HCI, Software Engineering, and design, and boundary object to bridge them, in *Proceedings of the conference on Human Computer Interaction (CHI 2004)*, workshop, Vienna, Austria, pp. 1723-1724.

John, B. E., Bass, L., Sanchez-Segura, M.-I., et al. (2004): Bringing usability concerns to the design of software architecture, in *Proceedings of 9th IFIP Working Conference on Engineering for Human-Computer Interaction*, Hamburg, Germany, pp. 1-19.

Johnson, B., Woolfolk, W. W., Miller, R., et al. (2005): *Flexible Software Design - Systems Development for Changing Requirements*, 1st edition, Auerbach Publications, Taylor & Francis Group, Boca Raton, FL, USA.

Juristo, N., Lopez, M., Moreno, A. M., et al. (2003): Improving software usability through architectural patterns, in *Proceedings of ICSE 2003 Workshop "Bridging the Gaps between Software Engineering and Human-Computer Interaction"*, Portland, Oregon, USA, pp. 12-19.

Kahler, H. (2001): *Supporting Collaborative Tailoring,* Doctoral Dissertation, Datalogiske Skrifter No. 91, 2001, Department of Computer Science, Roskilde University, Roskilde, Denmark.

Kahler, H., Mørch, A., Stiemerling, O., et al. (2000): Introduction, *Computer Supported Cooperative Work (CSCW),* 9(1), pp. 1-4.

Kensing, F. (2003): *Methods and Practices in Participatory Design,* Doctoral Thesis, ITU University, ITU Press, Copenhagen.

Kensing, F. and Blomberg, J. (1998): Participatory Design - issues and concerns, *Computer Supported Cooperative Work (CSCW),* 7(1998), pp. 167-185.

Kensing, F., Simonsen, J. and Bødker, K. (1998): MUST - a method for Participatory Design, *Human-Computer Interaction,* 13(1998), pp. 167-198.

Kiczales, G. (1991): *The Art of the MetaObject Protocol*, 1st edition, MIT Press, UK.

Kiczales, G. (1992): Towards a new model of abstraction in the engineering of software, in *Proceedings of the International Workshop on New Models for Software Architecture (IMSA): Reflection and Meta-Level Architecture*, Tokyo, Japan.

Kiczales, G., Ashley, J. M., Rodriguez, L., et al. (1993): Metaobject Protocols - why we want them and what else they can do, in *Objectoriented Programming - The CLOS Perspective*, A. Paepcke (Ed.), 1st edition, MIT Press, pp. 101-118.

Kindberg, T., Barton, J., Morgan, J., et al. (2000): People, places, things - web presence for the real world, in *Proceedings of 3rd IEEE Workshop on Mobile Computing Systems and Applications (WMCSA 2000)*, Monterey Marriot, CA, USA, pp. 19-28.

Klann, M. (2003): EUD-Net's roadmap to end-user development, in *Proceedings of the Workshop on End-User Development in Conjunction with CHI 2003 Conference*, Fort Lauderdale, USA, pp. 23-26.

Kolfschoten, G. L., Appelman, J. H., Briggs, R. O., et al. (2004): Recurring patterns of facilitation interventions in GSS sessions, in *Proceedings of the 37th Hawaii International Conference on System Sciences*, Hawaii, USA.

Kolfschoten, G. L., Briggs, R. O., de Vreede, G.-J., et al. (2006): A conceptual foundation of the ThinkLet concept for Collaboration Engineering, *International Journal of Human-Computer Studies,* 64(2006), pp. 611-621.

Kuniacsky, M. (2003): *Observing the User Experience - A Practitioner's Guide to User Research*, 1st edition, Morgan Kaufmann Publishers, San Fransisco, USA.

Langheinrich, M., Mattern, F., Roemer, K., et al. (2000): First step towards an event-based infrastructure for Smart Things., in *Proceedings of Ubiquitous Computing Workshop (PACT 2000)*, Philadelphia, PA, USA.

Lave, J. and Wenger, E. (1991): *Situated Learning - Legitimate Peripheral Participation*, 1st edition, Cambridge University Press, Cambridge, USA.

Lee, J., Siau, K. and Hong, S. (2003): Enterprise integration with ERP and EAI, *Communications of the ACM,* 46(2), pp. 54-60.

Lehman, M. M. (1980): Programs, life cycles, and laws of software evolution, in *Proceedings of the IEEE (Special issue on Software Engineering),* 68(9), 1060-1076.

Lehman, M. M. (1994): Software evolution, in *Encyclopedia of Software Engineering*, J. L. Marciniak (Ed.), 1st edition, John Wiley & Sons Ltd, New York, USA, pp. 1202-1208.

Letondal, C. (2006): Participatory programming - developing programmable bioformatics tools for end-users, in *End User Development*, H. Lieberman, F. Paternò and V. Wulf (Ed.), 1st edition, Springer, Netherlands, pp. 207-242.

Letondal, C. and Mackay, W. E. (2004): Participatory programming and the scope of mutual responsibility - balancing scientific, design and software commitment, in *Proceedings of Participatory Design Conference (PDC'04)*, Toronto, Canada, pp. 31-41.

Lindeberg, O. and Diestelkamp, W. (2001): How much adaptability do you need? Evaluating meta-modeling techniques for adaptable special-purpose systems, in *Proceedings of the 5th conference on Software Engineering and Applications*, Anaheim, USA.

Lukosch, S. and Schümmer, T. (2006): Groupware development support with technology patterns, *International Journal of Human-Computer Studies,* 64(7), pp. 599-610.

Löwgren, J. and Stolterman, E. (2004): *Design av Informationsteknik*, 2nd edition, Studentlitteratur, Lund, Sweden, (in Swedish).

Mackay, W. E. (1990): Patterns of sharing customizable software, in *Proceedings of the Conference on Computer Supported Cooperative Work (CSCW'90)*, Los Angeles, California, USA, pp. 209-221.

Mackay, W. E. (1991): Triggers and barriers to customizing software, in *Proceedings of the Conference on Human Factors in Computing Systems (CHI'94)*, Boston, Massachusetts, USA, pp. 153-160.

MacLean, A., Carter, K., Lövstrand, L., et al. (1990): User-tailorable systems - pressing the issues with buttons, in *Proceedings of the Conference on Human Factors in Computer Systems, CHI 90*, New York, USA, pp. 175-182

Maes, P. (1987): *Computational Reflection*, Technical Report 87_2, Vrieje Universiteit, Brussels, Belgium.

Malone, T. W., Lai, K.-Y. and Fry, C. (1995): Experiments with oval - a radically tailorable tool for cooperative work, *ACM Transactions on Information Systems,* 13(2), pp. 177-205.

March, S. T. and Smith, G. F. (1995): Design and natural science research on information technology, *Decision Support Systems*, 15(1), pp. 251-266.

Mason, J. (1996): *Qualitative Researching*, 2nd edition, Sage Publications, London, UK.

McCall, J. A., Richards, P. K. and Walters, G. F. (1977): Factors in software quality, *Nat'l Tech Information Service,* 1, 2 and 3.

Mens, T., Wermelinger, M., Ducasse, S., et al. (2005): Challenges in software evolution, in *Proceedings of the 8th International Workshop on Principles of Software Evolution*, Lisbon, Portugal, pp. 13-22.

Minar, N., Gray, M., Roup, O., et al. (1999): Hive - distributed agents for networking things, in *Proceedings of the 1st International Symposium on Agents Systems Applications and the 3rd Symposium on Mobile Agents (ASA/MA '99)*, Palm Springs, CA, USA, pp. 118-129.

Muller, M. J., Wildman, D. M. and White, E. A. (1993): Participatory Design, *Communication of ACM,* 36(4), pp. 23-28.

Muller, M. J., Wildman, D. M. and White, E. A. (1994): Participatory Design trough games and other group exercises, in *Proceedings of the conference on Human Factors in Computing Systems*, Boston, MA, USA, pp. 411-412.

Muller, M., Matheson, L., Page, C., et al. (1998): Methods & tools - participatory heuristic evaluation, *Interactions,* 5(5), pp. 13-18.

Mørch, A. (1995): Three levels of end-user tailoring - customization, integration, and extension, in *Proceedings of the 3rd Decennial Aarhus Conference*, Aarhus, Denmark, pp. 157-166.

Mørch, A. (1997): Evolving a generic application into domain-oriented design environment, *Scandinavian Journal of Information System,* 8 (2), pp. 63-89.

Mørch, A. (2002): Aspect-oriented software components, in *Evolutionary and Adaptive Information Systems*, N. Patel (Ed.), 1st edition, IDEA Group Publishing, USA, pp. 105-124.

Mørch, A. (2002): Evolutionary growth and control in user tailorable systems, in *Evolutionary and Adaptive Information Systems*, N. Patel (Ed.), 1st edition, IDEA Group Publishing, USA, pp. 30-58.

Mørch, A. and Mehandjiev, N. (2000): Tailoring as collaboration - the mediating role of multiple representations and Application Units, *Computer Supported Cooperative Work (CSCW),* 9(1), pp. 75-100.

Mørch, A. I., Stevens, G., Won, M., et al. (2004): Component-based technologies for end-user development, *Communications of the ACM,* 47(9), pp. 59-62.

Nardi, B. A. (1993): *A Small Matter of Programming - Perspectives on End User Computing*, 1st edition, MIT Press, Cambridge, USA.

Nardi, B. A. and Miller, J. R. (1991): Twinkling lights and nested loops - distributed problem solving and spreadsheet development, *International Journal of Man-Machine Studies,* 34(1), pp. 161-184.

Nielsen, J. and Mack, R. L. (1994): *Usability Inspection Methods*, 1st edition, John Wiley & Sons, Inc, New York, NY, USA.

Norman, D. A. (1999): Affordance, conventions, and design, *Interactions,* 6(3), pp. 38 - 43

Nunamaker, J., Chen, M. and Purdin, T. (1991): System development in information systems research, *Journal of Management Information Systems,* 7(3), pp. 89-106.

Olsson, E. (2004): What active users and designers contribute in the design process, *Interacting with Computers,* 16(2004), pp. 377-401.

Paterno, F., Klann, M. and Wulf, V. (2002): End-user development - empowering people to flexibly employ advanced information and communication technology, *Research Agenda and Roadmap for EUD*, Deliverable for EUD-Net Network of Excellence, Action Line: IST-2002-8.1.2.

Patton, M. Q. (1987): *How to Use Qualitative Methods in Evaluation*, 2nd edition, SAGE Publications, USA.

Pipek, V. and Kahler, H. (2006): Supporting collaborative tailoring, in *End User Development*, H. Lieberman, F. Paternò and V. Wulf (Ed.), 1st edition, Springer, Netherlands, pp. 315-345.

Pree, W. and Sikora, H. (1997): Design patterns for object-oriented software development, in *Proceedings of the International Conference on Software Engineering (ICSE '97), tutorial*, Boston, Massachusetts, USA, pp. 663-664.

Preece, J., Sharp, H. and Rogers, Y. (2002): *Interaction Design - Beyond Human-Computer Interaction*, 1st edition, John Wiley & Sons, Inc., New York, USA.

Regnell, B., Höst, M., Natt och Dag, J., et al. (2000): Visualization of agreement and satisfaction in distributed prioritization of market requirements, in *Proceedings of 6th International Workshop on Requirements Engineering: Foundation for Software Quality*, Stockholm, Sweden.

Rivard, F. (1996): Smalltalk - a Reflective Language, in *Proceedings of the 1st International Conference on Computational Reflection (Refelction'96)*, San Francisco.

Robinson, M. (1999): Computer Supported Cooperative Work - cases and concepts, in *Proceedings of Groupware'99, pp. 59-75.* Reprinted in R.M Baecker (Ed.) Readings in Groupware and Computer-Supported Cooperative Work: Assisting Human-Human Collaboration, pp. 29-49.

Robson, C. (2002): *Real World Research*, 2nd edition, Blackwell Publishers Ltd, Oxford, UK.

Rode, J., Rosson, M. B. and Quiñones, M. A. P. (2006): End user development of web applications, in *End User Development*, H. Lieberman, F. Paternò and V. Wulf (Eds.), 1st edition, Springer Verlag, Dordrecht, Netherlands, pp. 161-182.

Rosenbaum, S. (1989): Usability evaluations versus usability testing - when and why? *IEEE Transaction on Professional Communication*, 42(4), pp. 210-216.

Sánchez-Jankowski, M. (2002): Representation, responsibility and reliability in participant-observation, in *Qualitative Research in Action*, T. May (Ed.), 1st edition, SAGE Publishers, Trowbridge, Wiltshire, UK, pp. 144-160.

Sanoff, H. (2007): Editorial - special issue on participatory design, *Design Studies,* 28(3), pp. 213-215.

Schuler, D. and Namioka, A. (1993): *Participatory Design - Principles and Practices*, 1st edition, Lawrence Erlbaum Associates, Hillsdale, NJ, USA.

Schümmer, T. and Slagter, R. (2004): The Oregon software development process, in *Proceedings of the 5th International Conference on Extreme Programming and Agile Processes in Software Engineering (XP 2004)*, Berlin/Heidelberg, Germany, pp. 148-156.

Schümmer, T., Lukosch, S. and Slagter, R. (2005): Empowering end-users - a pattern-centered groupware development process, in *Proceedings of 11th International Workshop on Groupware (CRIWG 2005) - Groupware: Design, Implementation, and Use*, Porto de Galinhas, Brazil, pp. 73-88.

Shapiro, D. (2005): Participatory Design - the will to succeed, in *Proceedings of the 4th Decennial Conference on Critical Computing - between Sense and Sensibility*, Aarhus, Denmark, pp. 29-38.

Silverman, D. (2001): *Interpreting Qualitative Data; Methods for Analyzing Talk, Text and Interaction*, 2nd edition, SAGE Publishers, Trowbridge, Wiltshire, UK.

Sommerville, I. (2001): *Software Engineering*, 6th edition, Pearson Education Limited, Harlow, UK.

Star, S. L. and Griesemer, J. R. (1989): Institutional ecology, translations and boundary objects - amateurs and professionals in Berkeley's museum of vertebrate zoology, 1907-39, *Social Studies of Science,* 19(8), pp. 387-420.

Stevens, G., Quaisser, G. and Klann, M. (2006): Breaking it up - an industrial case study of component-based tailorable software design, in *End-User Development*, H. Lieberman, F. Paternò and V. Wulf (Eds.), 1st edition, Springer, Dordrecht, Netherlands, pp. 269-294.

Stiemerling, O. (2000): *Component-Based Tailorability,* Doctoral Dissertation, Bonn University, Bonn, Germany.

Stiemerling, O., Cremers and Armin, B. (1998): Tailorable component architectures for CSCW-systems, in *Proceedings of the 6th Euromicro Workshop on Parallel and Distributed Programming*, Madrid, Spain, pp. 302-308.

Stiemerling, O., Kahler, H. and Wulf, V. (1997): How to make software softer - designing tailorable applications, in *Proceedings of the Symposium on Designing Interactive Systems (DIS'97)*, Amsterdam, Netherlands, pp. 365-376.

Subramanian, M. (1999): *Network Management - An Introduction to Principles and Practice*, 1st edition, Addison-Wesley, Reading, MA, USA.

Svahnberg, M. (2003): *Supporting Software Architecture Evolution; Architecture Selection and Variability,* Doctoral Dissertation, Department of Software Engineering and Computer Science, Blekinge Institute of Technology, Ronneby, Sweden.

Svahnberg, M. (2005): A taxonomy of variability realization techniques, *Software - Practice and Experience,* 35(8), pp. 705-754.

Szyperski, C. (2002): *Component Software beyond Object-Oriented Programming*, 2nd edition, Addison-Wesley, London, UK.

Tidwell, J. (2006): *Designing Interfaces - Patterns for Effective Interaction Design*, 1st edition, O'Reilly, Sebastpol; CA, USA.

Trigg, R. and Bødker, S. (1994): From implementation to design - tailoring and the emergence of systematization in CSCW, in *Proceedings of the Conference on Computer Supported Cooperative Work (CSCW 94)*, Chapel Hill, NC, USA, pp. 45-54.

de Vreede, G. J., Davison, R. and Briggs, R. O. (2003): How a silver bullet may lose its shine - learning from failures with Group Support Systems. *Communications of the ACM,* 46(8), pp. 96-101.

Wake, W. C. (2002): *Extreme Programming Explored*, 1st edition, Addison-Wesley, Upper Sadle River, NJ, USA.

Waldo, J. (1999): The Jini architecture for network-centric computing, *Communication of ACM,* 42(7), pp. 76-82.

Wang, W. and Haake, J. M. (2000): Tailoring groupware - the cooperative hypermedia approach, *Computer Supported Cooperative Work (CSCW),* 9(1), pp. 123-146.

Want, R., Fishkin, K. P., Gujar, A., et al. (1999): Bridging physical and virtual worlds with electronic tags, in *Proceedings of the 1999 Conference on Human Factors in Computing Systems (CHI '99)*, Pittsburg, PA, USA, pp. 370-377.

Weiser, M. (1991): The computer for the century, *Scientific America,* 265 (3), pp. 94-104.

Wesson, J. and Cowley, L. (2003): Designing with patterns - possibilities and pitfalls, in *Proceedings of the 2nd Workshop on Software and Usability Cross-Pollination: The role of Usability Patterns, INTERACT 2003*, Zürich, Switzerland.

Winograd, T. (2000): Interaction spaces for the 21th century computing, in *HCI in the New Millennium*, J. Carroll (Ed.), 1st edition, Addison Wesley, pp. 259-276.

Wulf, V. and Rohdein, M. (1995): Towards an integrated organization and technology development, in *Proceedings of the Symposium on Designing Interactive Systems (DIS'95)*, Ann Arbor, Michigan, USA, pp. 55-64.

Yin, R. K. (2003): *Case Study Research - Design and Methods*, 1st edition, SAGE Publications, Thousand Oaks, CA, USA.

Zimmerman, C. (1996): Reflections on adaptable real-time metalevel architecture, *Journal of Parallel and Distributed Computing,* 36(1996), pp. 81-89.

List of Figures

List of Tables

Appendixes

Introduction to Examples

A company establishes contracts with the subcontractors. The contracts are a base for prices and discounts for the delivered material. Dependent on different market forces the contracts are updated and renegotiated. A contract always consists of contract type, sub contractor, regular price plan and time period the contract is valid for. Specific discounts and a specific price plan that differ from the regular price plan for a time period can be added to the contract. The contracts are stored in the Company's "Contract Handler" and the content of the contracts is used by different software systems.

The contract official has to create new contract types and fill the contracts with data that is valid for a specific contract.

Contracts are renegotiated and new contracts are created as a response to what happens on the market.

What has to be done in the Contract Handler when a contract is changed can differ. Four ways of creating new contract types are illustrated in Examples 1-4

Example 1

When the contract official shall create a new contract he can choose to create

- A basic contract (consisting of contract type, sub contractor, regular price plan and time period)
- A contract with discounts (that contains all a basic contract consists of and a component for discount)
- A contract with a specific price plan (that contains all a basic contract consists of and specific price plan that differs from the regular price plan for some time period) or
- A contract with both specific price plan and discounts.

When the choice is made a user interface starts. The interface represents the chosen contract type and the contract official may fill in the data that is required.

When the contract is renegotiated and thereby changed a new contract is created as above. In this case, four types of contracts are preprogrammed in the Contract Handler. What happens in the system when a new contract is created is that the contract official chooses one of the contract types that shall be used when the chosen contract type is shown.

Example 2

When the contract official creates a new contract he first decides what kind of contract he wants to create. If it is a contract with discounts, he first chooses new contract in the user interface. He then automatically gets a basic contract on the monitor. Then he clicks on the button marked "Add" and can thereafter choose if he wants discounts and a specific price plan too. He chooses discounts and when this choice is made the interface is extended with a component representing the discounts. The contract official can then fill in the data required.

When the contract is renegotiated and thereby changed, a new contract is created as above. In this case there are three different contract modules in the program.

- Basic contract
- Discount module
- Specific price plan module.

When the contract official chooses to create a contract by choosing a basic contract and a discount module, a relationship between the modules are created. This relationship is represented in the program in form of code. This code determines what is shown, in this case both basic contract and the discount component.

Example 3

When the contract official shall create a new contract he first decides if he wants to create a new contract from start or if he wants to build on an existing contract. Maybe he has created a contract with discounts earlier and now he needs a contract that contains both discount and a specific price plan. In the interface he can choose to start from a contract that combines a basic contract and discount that he or someone else created earlier. He then automatically gets a basic contract with discount on the monitor. Thereafter he clicks the "add" button and he adds a specific price plan. When this is done the interface is extended with a specific price plan component and the contract official can fill in the data.

When the contract official has created a new contract type by relating some of the basic modules, as in Example 2, the combination is regarded by the program as a new composite module and is treated in the same way as the basic components. Every time a new type of contract is created a new module is created (in this case only three ways of combining the modules exist. This means that only six modules can exist in the system).

In this case there are four different contract modules in the program.

- Basic contract
- Discount module
- Specific price plan module.
- Personal modules

Example 4

In the examples above the contract official has a limited number of choices. But suppose there is a need for a component in the contract that handles the delivery guarantee. As the Contract Handler is described in Examples 1-3 above this possibility does not exist. But the Contract Handler in this example makes it possible for the contract official to make modules from the start. When the contract official shall make a contract that contains delivery guarantee he chooses "Create Module" and an interface opens showing an empty module. The contract official can fill the empty module with predefined parameters and even write some code to make the module work as desired. He saves his "delivery guarantee module" and then chooses to create a new contract. A basic contract is shown on the monitor and he chooses "add" and his new "delivery guarantee module" is shown among the modules to choose between. He can choose to add the new module and then the interface is extended with a part of delivery guarantee. Then the contract official can fill in the data.

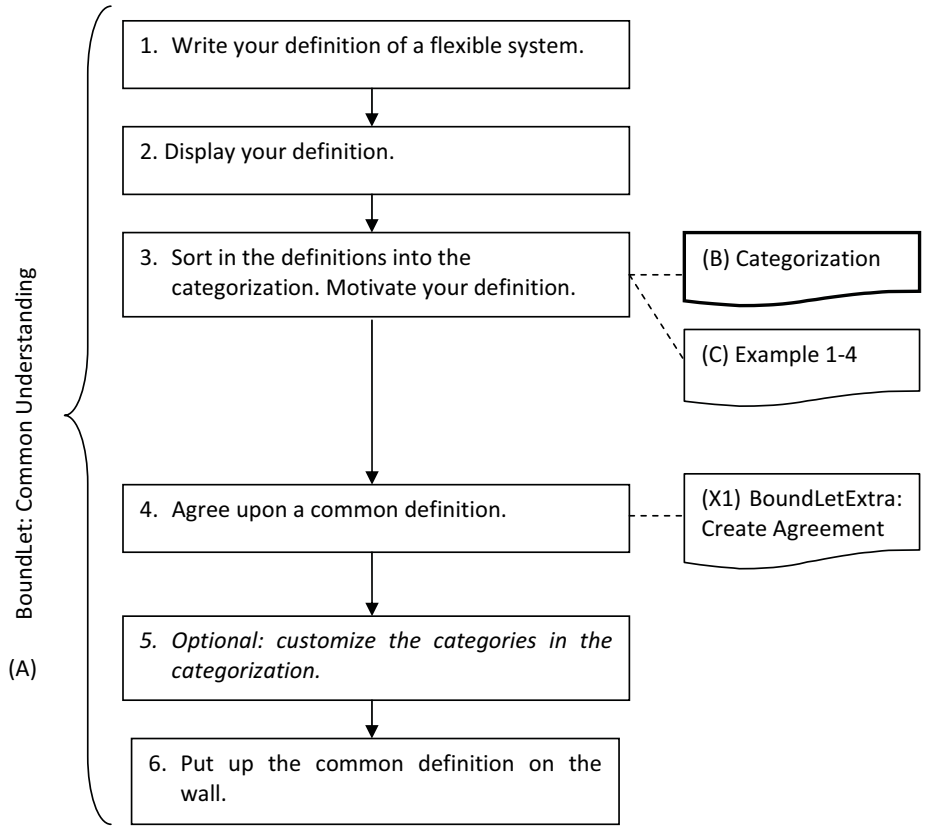
What happens when the contract official creates the new module by choosing between parameters and writing some code is that the program interprets the extension and transforms it to a coherent code that is encapsulated by the code that constitutes the 'empty' module. As all modules (preprogrammed or self made) have the same shell they can be treated in the same way.

CATEGORIZATION TOOL

DOCUMENTS

(A)	BoundLet: Common Understanding (1 page)	*BoundLet*
(B)	Categorization (1 page)	*Artefact*
(C)	Example 1-4 (5 page)	*Support documents*
(X1)	BoundLetExtra: Create Agreement [1]	*Support document*

OVERVIEW of WORKFLOW



[1] BoundLet that frames how to reach an agreement in the group. Not included in the appendix.

BOUNDLET: COMMON UNDERSTANDING

A

Input and Output

INPUT: Diverse opinions and experiences of flexible software systems.

OUTPUT: A unified definition of what a flexible software system means to the group.

Choose this tool…

- When there is confusion within the group about what flexible software systems mean and about which flexibility is needed in the software.
- When the group feels they need a common basis to work from.
- When the group starts a cooperation around flexibility in software.
- Even when there is no common concrete example to base the definition on.

Overview

- The participants first individually define to themselves what flexibility means. Thereafter each person's definition is shown to the others and together the group classifies the definitions. Then the participants start to negotiate about what the definition should be, to reach a definition all participants can agree upon. (also see workflow on the first page)

Artefact

- Categorization

Instructions

1. Define what a flexible software system means to you. Write down your definition.
2. When everyone is finished you must display your definition for everybody to see.
3. Sort all the definitions into suitable categories in **Categorization (B)**
   a. Use Example 1 to 4 (C) if you need a common example for the participants to relate to.
   b. Motivate and explain your definition when it is going to be sorted.
   c. Why do you think of flexibility in this way?
4. Use **BoundLetExtra: CreateAgreement (D**) to agree upon a common definition and to clarify differences in opinions.
5. Specify the descriptions of the categories in **Categorization** if this is needed for the categories to work in the specific context or situation.
6. Write down the common definition on a large paper and put it up on the wall.

Rules

- Everybody must write down their definition.
- Everybody must motive how they think about flexibility.
- Everybody must be active in the discussion to create a common definition.

Experiences

- *The group can make references to their positive and negative experiences here.*

CATEGORIZATION OF END-USER TAILORING

B

	User Perspective	System Perspective
Customization	***Set parameter values*** (The end-user makes small changes, e.g. sets parameter values.)	***Interpretation of existing code*** (Parameter Values are interpreted and used in existing code.)
Composition	***Link different existing components*** (The end-user relates different existing components to each other.)	***Definition of relationships between components*** (The relationships between the components are defined by a composition language. (It does not matter which programming language)) ***Code Generation*** *(optional)*
Expansion	***Creation of new component*** (The end-user creates a new component.)	***Definition of relationships between components*** (Components are integrated into the software by the implementation language and the new component does not differ from the pre-existing components. The composed component is used as a starting point for further tailoring.) ***New and predefined components are treated uniformly*** ***Code Generation*** *(optional)* (The software may generate code that is added to the pre-existing code, or incorporate the new component into the application in some other way.)
Extension	***Insertion of code*** (The end-user adds code to the software.)	***New code is added*** (New code (implemented by the end-user) is added to the pre-existing code.) ***Code Generation*** *(optional)* (The application may also generate code to integrate the end-user's code into the software.)

EXAMPLES INTRODUCTION

C:1

A company establishes contracts with the subcontractors. The contracts are a base for prices and discounts for the delivered material. Dependent on different market forces the contracts are updated and renegotiated. A contract always consists of contract type, sub contractor, regular price plan and time period the contract is valid for. Specific discounts and a specific price plan that differ from the regular price plan for a time period can be added to the contract. The contracts are stored in the Company's "Contract Handler" and the content of the contracts is used by different software systems.

The contract official has to create new contract types and fill the contracts with data that is valid for a specific contract.

Contracts are renegotiated and new contracts are created as a response to what happens on the market.

What has to be done in the Contract Handler when a contract is changed can differ. Four ways of creating new contract types are illustrated in Examples 1-4.

EXAMPLE 1 CUSTOMIZATION

C:2

This type of program contains the following possibilities to make changes:

When the contract official shall create a new contract he can choose to create

- A basic contract (consisting of contract type, sub contractor, regular price plan and time period)
- A contract with discounts (that contains all a basic contract consists of and a component for discount)
- A contract with a specific price plan (that contains all a basic contract consists of and specific price plan that differs from the regular price plan for some time period) or
- A contract with both specific price plan and discounts.

When the choice is made a user interface starts. The interface represents the chosen contract type and the contract official may fill in the data that is required.

When the contract is renegotiated and thereby changed a new contract is created as above. In this case, four types of contracts are preprogrammed in the Contract Handler. What happens in the system when a new contract is created is that the contract official chooses one of the contract types that shall be used when the chosen contract type is shown.

EXAMPLE 2 COMPOSITION

C:3

This type of program contains the following possibilities to make a change:

When the contract official creates a new contract he first decides what kind of contract he wants to create. If it is a contract with discounts, he first chooses new contract in the user interface. He then automatically gets a basic contract on the monitor. Then he clicks on the button marked “Add” and can thereafter choose if he wants discounts and a specific price plan too. He chooses discounts and when this choice is made the interface is extended with a component representing the discounts. The contract official can then fill in the data required.

When the contract is renegotiated and thereby changed, a new contract is created as above. In this case there are three different contract modules in the program.

- Basic contract
- Discount module
- Specific price plan module.

When the contract official chooses to create a contract by choosing a basic contract and a discount module, a relationship between the modules are created. This relationship is represented in the program in form of code. This code determines what is shown, in this case both basic contract and the discount component.

EXAMPLE 3 EXPANSION

C:4

This type of program contains the following possibilities to make a change:

When the contract official shall create a new contract he first decides if he wants to create a new contract from start or if he wants to build on an existing contract. Maybe he has created a contract with discounts earlier and now he needs a contract that contains both discount and a specific price plan. In the interface he can choose to start from a contract that combines a basic contract and discount that he or someone else created earlier. He then automatically gets a basic contract with discount on the monitor. Thereafter he clicks the "add" button and he adds a specific price plan. When this is done the interface is extended with a specific price plan component and the contract official can fill in the data.

When the contract official has created a new contract type by relating some of the basic modules, as in Example 2, the combination is regarded by the program as a new composite module and is treated in the same way as the basic components. Every time a new type of contract is created a new module is created (in this case only three ways of combining the modules exist. This means that only six modules can exist in the system).

In this case there are four different contract modules in the program.

- Basic contract
- Discount module
- Specific price plan module.
- Personal modules

EXAMPLE 4 EXTENSION

C:5

This type of program contains the following possibilities to make a change:

In the examples above the contract official has a limited number of choices. But suppose there is a need for a component in the contract that handles the delivery guarantee. As the Contract Handler is described in Examples 1-3 above this possibility does not exist. But the Contract Handler in this example makes it possible for the contract official to make modules from the start. When the contract official shall make a contract that contains delivery guarantee he chooses “Create Module” and an interface opens showing an empty module. The contract official can fill the empty module with predefined parameters and even write some code to make the module work as desired. He saves his “delivery guarantee module” and then chooses to create a new contract. A basic contract is shown on the monitor and he chooses “add” and his new “delivery guarantee module” is shown among the modules to choose between. He can choose to add the new module and then the interface is extended with a part of delivery guarantee. Then the contract official can fill in the data.

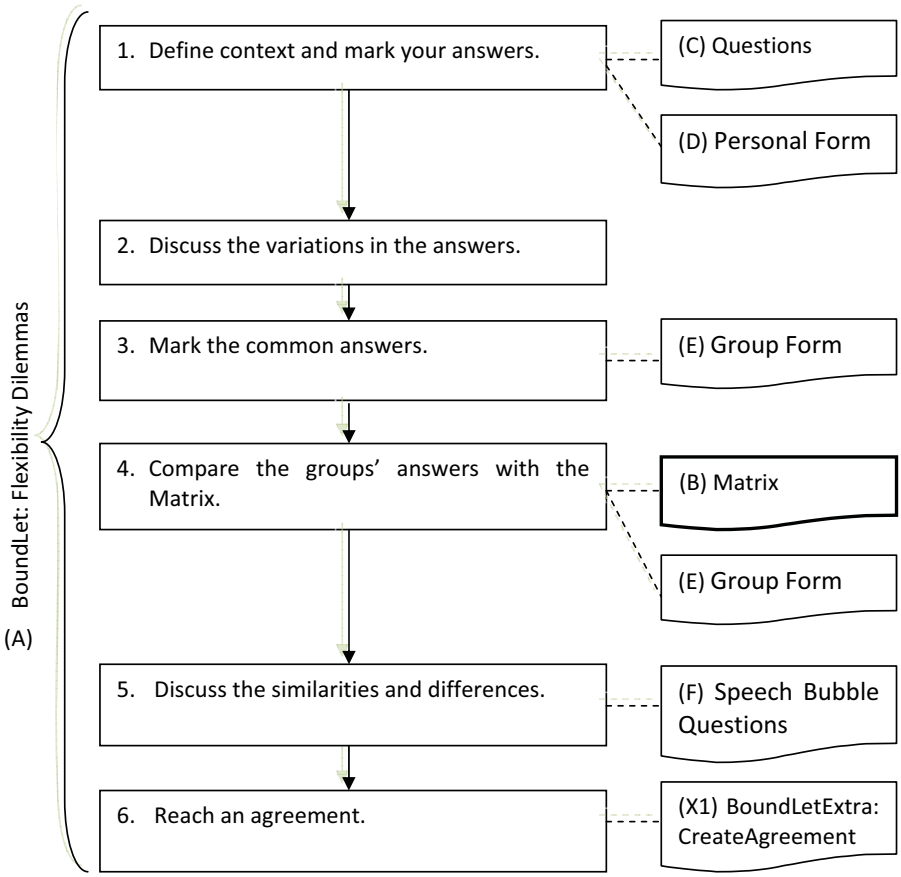
What happens when the contract official creates the new module by choosing between parameters and writing some code is that the program interprets the extension and transforms it to a coherent code that is encapsulated by the code that constitutes the ‘empty’ module. As all modules (preprogrammed or self made) have the same shell they can be treated in the same way.

MATRIX TOOL

DOCUMENTS

(A)	BoundLet: Flexibility Dilemmas (1 page)	*BoundLet*
(B)	Matrix (1 page)	*Artefact*
(C)	Questions (1 page)	*Support document*
(D)	Personal Form (1 page)	*Support document*
(E)	Group Form (1 page)	*Support document*
(F)	Speech Bubble Questions (1 page)	*Support document*
(X1)	BoundLetExtra: Create Agreement [1]	*Support document*

OVERVIEW of WORKFLOW



[1] BoundLet that frames how to reach an agreement in the group. Not included in the appendix.

BOUNDLET: FLEXIBILITY DILEMMAS

A

Input and Output

- A joint definition of what a flexible software system means.
- A joint case to work with.
- An understanding of the context and flexibility needed and also which compromises may be necessary to make.

Choose this tool…

- When the group needs to explore what type of flexibility to implement.
- When the participants need to discuss and probe deeply into the forces that influence the choice of flexibility.
- When there is a need to map out and understand the compromises that have to be made.
- When the group have a case, and a task to create new flexible functionality.
- When the group needs guidance in which flexibility that should be chosen for the task.

Overview

- The participants start by thinking individually about the context and the flexibility needed and try to pinpoint the need, based on the questions. Then the group does the same thing. The next step is to compare the group's answers and opinions with the Matrix. The Matrix can guide the discussion of which flexibility to use, and the advantages and disadvantages come up to the surface and illuminate which compromises have to be made. (also see workflow on the first page)

Artefact

- Matrix

Instructions

1. Define from **questions** the form of the context for the flexible functionality that shall be implemented.
   a. Mark your answer in the **Personal Form**. Deal with one question at a time and judge it as low (L), medium (M) or high (H).
2. Discuss in the group how the answers vary and the reasons for differences in the answers.
3. Mark the joint answer in the **Group Form.**
4. When all the questions are dealt with the groups judgements are compared with the **Matrix**. The numbers of corresponding answers is summarised in the **Group Form.**
5. The result is discussed with the help of the **Speech Bubble Questions**.
6. **BoundLetExtra: CreateAgreement** is used to reach an agreement of what type of flexibility to try to build for the functionality.

Rules

- In the discussion, the word passes around the table, to make it possible for everyone to express their opinions.

Experiences

- *The group can make references to their positive and negative experiences here.*

MATRIX B

Characteristics		Customization	Composition	Expansion	Extension
Business Changes	Frequency of change	M	M	H	H
	Anticipation of change	H	M	L-H[2]	L
	System support of change	L	M	M-H	H
Usability Issues	User control	H	M-H	M-H	?[3]
	Transparency	H	M-H	M-H	?
	Realization speed	H	H	M	M-H
	Frequency of use	L	H	-	-
	User competence	-	-	M-H	H
Software Attributes	Fault tolerance	H	M-H	M	L
	Complexity	L	L- M	M	H

Business changes

Frequency of change – how often the business changes occur, frequently or infrequently

Anticipation of change – to what extent it is possible to anticipate the business changes

System support for change – how well the software has to support business changes

Usability issues

User control – how much control the users have to have of what happens in the software

Transparency – how easy it should be for the users to know if the result is correct.

Realization speed – how fast it should be to realize the changes in the software.

Software attributes

Fault tolerance– to which degree the software has to prevent mistakes.

Complexity– how complex the software could be

[2] Users thought the example was highly suitable for anticipated changes, developers thought the example was not so suitable for such situations.

[3] Dependent of how user control is interpreted the value can be either H or L. Should the control be in the software or in the user knowledge?

QUESTIONS

C

Business changes

Frequency of change – How often do the business changes occur, frequently/infrequently?

Anticipation of change – To what extent is it possible to anticipate the business changes?

System support for change – How well does the software support business changes?

Usability issues

User control – How much control do the users have to have of what happens in the software?

Transparency – How easy should it be for the users to know if the result is correct?

Realization speed – How fast should it be to realize the changes in the software?

Software attributes

Fault tolerance – To which degree does the software have to prevent mistakes?

Complexity – How complex is the software allowed to be?

PERSONAL FORM

D

Judge how your context and environment relates to the characteristics?
(L=Low, M=Medium, H=High, ?= not sure)

Characteristics		Personal Matrix
Business Changes	Frequency of change	
	Anticipation of change	
	System support of change	
Usability Issues	User control	
	Transparency	
	Realization speed	
	Frequency of use	
	User competence	
Software Attributes	Fault tolerance	
	Complexity	

GROUP FORM

E

Judge how your context and environment relates to the characteristics?
(L=Low, M=Medium, H=High, ?= not sure)

Characteristics		Group Matrix
Business Changes	Frequency of change	
	Anticipation of change	
	System support of change	
Usability Issues	User control	
	Transparency	
	Realization speed	
	Frequency of use	
	User competence	
Software Attributes	Fault tolerance	
	Complexity	

Number of correspondences in the matrix	
Customization	
Composition	
Expansion	
Extension	

SPEECH BUBBLE QUESTIONS

F

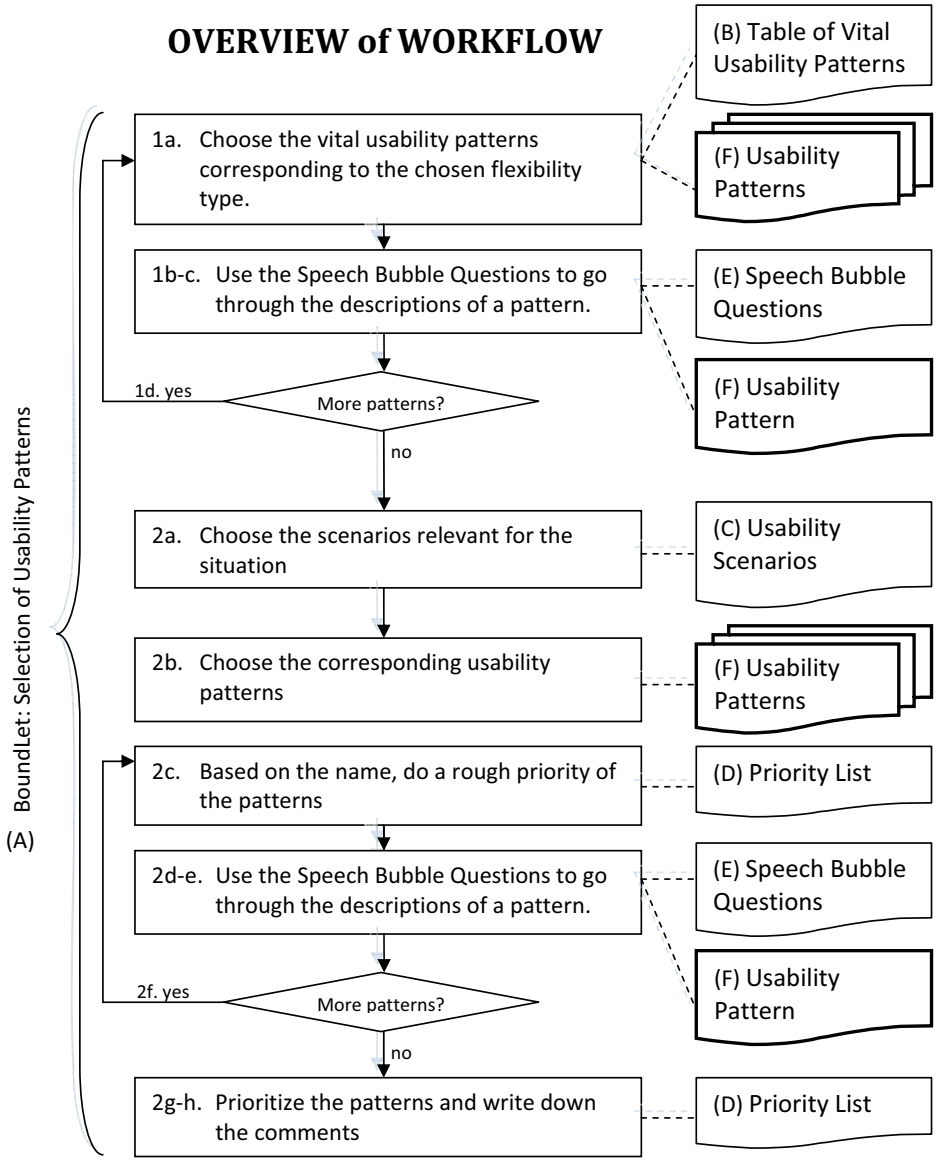
- Does the Matrix point towards a suitable flexibility type? If not, what is the reason?
- Is the answer unambiguous?
- Is the answer ambiguous?
- What compromises must be made?
- What are the advantages and disadvantages of the alternatives?
-
-
-

USABILITY PATTERN TOOL

DOCUMENTS

(A)	BoundLet: Selection of Usability Patterns (1 page)	*BoundLet*
(B)	Table of Vital Usability Patterns (1 page)	*Support document*
(C)	Usability Scenarios (1 page)	*Support document*
(D)	Priority List (1 page)	*Support document*
(E)	Speech Bubble Questions (1 page)	*Support document*
(F)	Usability Patterns[1]	*Artefact*

OVERVIEW of WORKFLOW



[1] Only the patterns structure is available in the appendix.

BOUNDLET: SELECTION OF USABILITY PATTERNS

A

Input and Output

INPUT: A chosen flexibility type
Tentativ, basic architecture
OUTPUT: A collection of prioritized usability patterns that can be used in the design of the flexible functionality.

Choose this tool…

- When the group is new to the use of patterns.
- When the ability to understand and use patterns needs to be trained.
- In the initial stage when user participation in the technical design process is introduced.
- When the goal is primarily to make the software usable.

Overview

- Users and developers together work through the usability patterns, starting with the most important. The aim is to explore the consequences of the use of the specific pattern and to understand and agree on which design decisions to make. (Also see the workflow on the first page)

Artefact

- Usability Patterns with architectural impact

Instructions

1 - VITAL PATTERNS

a. Based on the flexibility type chosen the **vital usability patterns** (see table) **are selected**.
b. By using the pattern's description work through the **Speech Bubble Questions** together.
c. Work through the parts of the pattern description that have not been considered. Base the work on the maturity of the group. A more mature group can go deeper into the pattern description. Does it add anything to the assessment?
d. Move to the next pattern by returning to b).

2 – OTHER PATTERNS

a. Work through the other **usability scenarios** and choose those that are relevant for the situation.
b. Choose the **usability patterns** corresponding to the usability scenarios
c. Based on the name, make a preliminary prioritization. Which pattern is most important? Start with that pattern.
d. By using the pattern's description work through the **Speech Bubble Questions** together.
e. Work through the parts of the pattern description that have not been considered. Base the work on the maturity of the group. A more mature group can go deeper into the pattern description. Does it add anything to the assessment?
f. Choose the next pattern. Repeat h to j until all the patterns are worked through.
g. When all the patterns in the selected collection are worked through, prioritize the patterns together in the group.
h. Write down the prioritizations and comments in the **priority list**

Rules

- In the initial phase the focus must be on the usability pattern that is of vital importance for the type of flexibility chosen (see Table of Vital Usability Pattern).
- In the discussion word is passed around the table to make it possible for everybody to express their opinion.
- The opinions that are revealed must be weighed together and be used actively to make the process proceed (deliberation).

Experiences

- *The group can make references to their positive and negative experiences here.*

TABLE OF VITAL USABILITY PATTERNS

B

Category	Usability Scenario	Pattern
Customization	*Checking for correctness* *Supporting undo* *Providing good help*	***Form/Field validation*** ***Undo*** ***Wizard, Context-sensitive help, Standard Help, Tour*** ***User profile***

Category	Usability Scenario	Pattern
Composition	Checking for correctness Supporting undo Providing good help *Working in an unfamiliar context*	Form/Field validation Undo Wizard, Context-sensitive help, Standard Help, Tour User profile ***Workflow model***

Category	Usability Scenario	Pattern
Expansion	Checking for correctness Supporting undo Providing good help Working in an unfamiliar context *Observing system state*	Form/Field validation Undo Wizard, Context-sensitive help, Standard Help, Tour User profile Workflow model *Status indication*

Category	Usability Scenario	Pattern
Extension	Checking for correctness Supporting undo Providing good help Working in an unfamiliar context Observing system state *Verifying resources*	Form/Field validation Undo Wizard, Context-sensitive help, Standard Help, Tour User profile Workflow model **Status indication** *Alert*

USABILITY SCENARIOS

C

1. **Checking for correctness** [2]
2. **Supporting undo**
3. **Providing good help**
4. **Working in an unfamiliar context**
5. **Observing system state**
6. **Verifying recourses**
7. Aggregating data
8. Aggregating commands
9. Cancelling commands
10. Using applications concurrently
11. Maintaining device independence
12. Evaluating the system
13. Recovering from failure
14. Retrieving forgotten passwords
15. Reusing information
16. Supporting international use
17. Leveraging human knowledge
18. Modifying interfaces
19. Supporting multiple activities
20. Navigating within a single view
21. Working at the users' pace
22. Predicting task duration
23. Supporting comprehensive searching
24. Operating consistently across views
25. Making views accessible
26. Supporting visualization
27. Supporting personalization

[2] Explanations to the scenarios are not included in the appendix.

PRIORITY LIST

D

Rough prioritization	
Priority	**Pattern**
1	
2	
3	
4	
5	
6	
7	

Priority	Pattern	Comments
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		

SPEECH BUBBLE QUESTIONS

E

- What impact does the pattern have on use?
- What impact does the pattern have on the architecture?
- Does the use of the pattern mean that compromises have to be made?
- How can this pattern be realized?
-
-

USABILITY PATTERN <NAME>

F

Usability Pattern for End-user tailorable software[3]		
Introductory description		
• **Name**		
• **Ranking**		*The author's confidence in the pattern*
• **Tailoring Categories**		*Which categories of tailoring the pattern is suitable for*
• **Illustration**		
Overall description of problem and solution		
• **Problem**		
• **Forces**	• Environment and task	*Forces from environment and task that influence the choice of solution.*
	• Human desires and capabilities	*Forces from human desires and capabilities that have an impact on the choice of solution.*
	• State of the software	*Forces generated by the system state, for example software is sometimes unresponsive*
• **General Solution**		
Detailed description of solution		
• **Specific Solution**		*Example of prior design decisions that influence the choice of solution. The forces are specific for the situation.*
	• Prior design decisions	
• **Diagrams**		
• **Consequences**		
• **Danger spots**		
• **Sample code**		*A short example of how to implement the pattern. Written in the language used at the company or in C++ since this is well known.*
• **Examples**		*Examples of features in applications where the pattern is used*
• **Related patterns**		

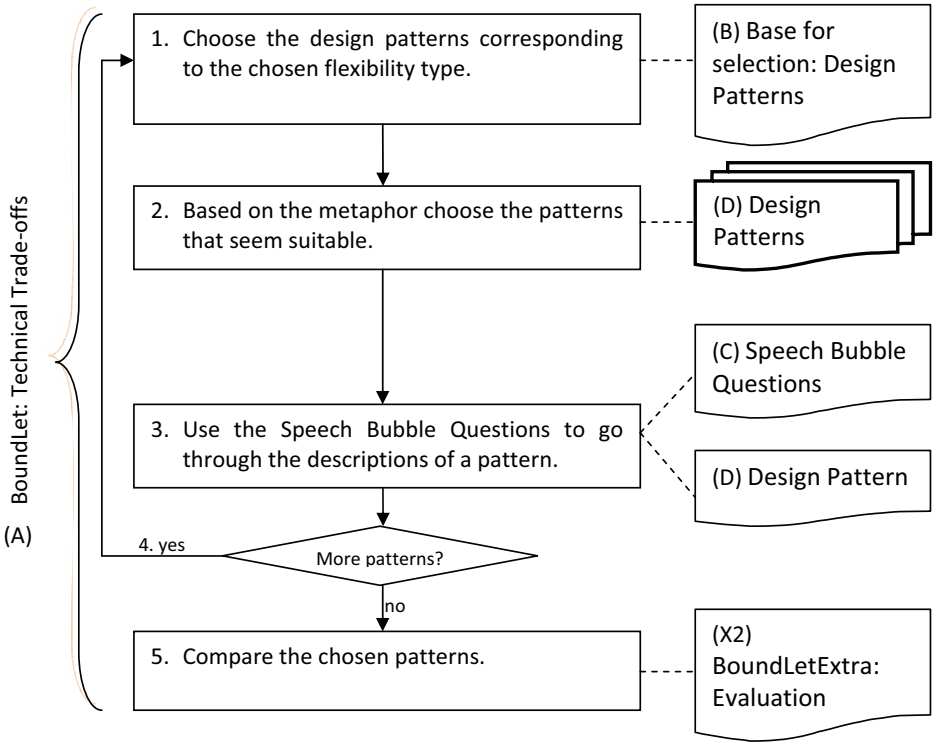
[3] Only the pattern structure is available in the appendix.

DESIGN PATTERN TOOL

DOCUMENTS

(A)	BoundLet: Technical Trade-offs (1 page)	*BoundLet*
(B)	Base for selection: Design Patterns (1 page)	*Support document*
(C)	Speech Bubble Questions (1 page)	*Support document*
(D)	Design Patterns[1]	*Artefacts*
(X2)	BoundLetExtra: Evaluation [2]	*Support document*

OVERVIEW of WORKFLOW



[1] Only the patterns structure is available in the appendix.
[2] BoundLet that frames how to compare and evaluate the patterns. Not included in the appendix.

BOUNDLET: TECHNICAL TRADE-OFFS

A

Input and Output

INPUT: Chosen type of flexibility.
Familiarity with patterns.
OUTPUT: Suggestions for design patterns that can be used for the design of flexible systems.

Choose this tool…

- When the group is used to working with patterns and all participants feel comfortable in such situations.
- When the users think it is interesting to learn more about underlying techniques and their consequences for use.

Overview

- Users and developers together create an overall goal for what is needed. Patterns are selected bases on the corresponding metaphor. Each pattern is discussed on the basis of a set of questions. The patterns are compared and a first choice is made. (See also the workflow on the first page)

Artefact

- Design Patterns

Instructions

1. Dependent on the type of flexibility that is to be implemented a collection of design patterns are chosen. Use **Base of selection of Design Patterns (B)**
2. Based on the pattern metaphors the patterns that best match the idea of the software system are chosen.
3. The participants work through the pattern by using the pattern description of the **Design Pattern.** The **Speech Bubble Questions** may help in the work.
4. Continue with the next pattern to get an overview and understanding of the different patterns.
5. Compare the patterns. Use **BoundLetExtra: Evaluation.**

Rules

- In the discussion words is passed around the table to make it possible for everyone to contribute to the discussion.
- All the participants' opinions are valuable.
- It happens easily that the developers take over, as they already possesses technical skill, but this must be prevented so that nobody feels inferior.
- The opinions that are revealed must be considered and actively used to make the process proceed. (deliberation)

Experiences

- *The group can make references to their positive and negative experiences here.*

BASE FOR SELECTION OF DESIGN PATTERN

B

Categorization of tailoring	Explanation	
customization	1.Choosing specialisation within a component	
	2.Choosing between different operations	
composition	3.Choosing components	
expansion	6.Creating a new component by connecting several components	4. *Adding* connector
extension	5.Creating a new component by subclassing	

Category	Design Pattern	Type of 'change'
customization	Strategy	1
	Template Method	1,2
	Command	2
composition	Decorator	3
extension	Adapter	4
	Façade	4
	Abstract Factory	5
	Prototype	5
	Interpreter	5
	Proxy	5,6
expansion	Adapter	4
	Façade	4
	Proxy	5,6
	Builder	6
	Composite	6
	Mediator	6

SPEECH BUBBLE QUESTIONS

C

- What impact does the pattern have on use?
- What impact does the pattern have on maintenance?
- What are the advantages and disadvantages?
- Does using the pattern mean that compromises have to be made?
- How should the pattern be realized?
- Go through the remaining parts of the pattern that appear to be relevant. Do they add anything to the judgment?
-

DESIGN PATTERN <NAME>

D

Design Pattern for End-user tailorable software[3]		
Introductory description		
• **Name**		
• **Ranking**		*The author's confidence in the pattern*
• **Tailoring Categories**		*Which categories of tailoring the pattern is suitable for*
• **Illustration**		
Overall description of problem and solution		
• **Problem**		
• **Forces**	• Environment and task	*Forces from environment and task that influence the choice of solution.*
	• Human desires and capabilities	*Forces from human desires and capabilities that have an impact on the choice of solution.*
	• State of the software	*Forces generated by the system state, for example software is sometimes unresponsive*
• **General Solution**		
Detailed description of solution		
• **Specific Solution**	• Prior design decisions	*Example of prior design decisions that influence the choice of solution. The forces are specific for the situation.*
• **Diagrams**		
• **Consequences**		
• **Danger spots**		
• **Sample code**		*A short example of how to implement the pattern. Written in the language used at the company or in C++ since this is well known.*
• **Examples**		*Examples of features in applications where the pattern is used*
• **Related patterns**		

[3] Only the pattern structure is available in the appendix.

ABSTRACT

In most business areas today, competition is hard and it is a matter of company survival to interpret and follow up changes within the business market. The margin between success and failure is small. Possessing suitable, sustainable information systems is an advantage when attempting to stay in the front line of the business area. In order to be and remain competitive, these information systems must be up-to-date, and adapt to changes in the business environment. Keeping business systems up-to-date in a business environment that changes rapidly and continuously, is a huge challenge.

This thesis is concerned with end-user tailorable software. Tailorable software makes it possible for end users to evolve an application better to fit altered business requirements and tasks. In the view of tailorable software taken in this thesis, the users should be seen as co-designers, as they take over the design of the software when it is in use. In this work, it is important that the users are aware of the possibilities and limitations of the software.

However, tailoring is not enough, because the tailoring capabilities are always limited, meaning that tailoring cannot support completely unanticipated changes. The tailoring capabilities must therefore be extended, and tailoring activities must be coordinated with software evolution activities performed by professional developers. This allows the system to adapt continuously to a rapidly changing business environment and thereby live up to the intention of the system. Studies so far have tended to look at evolution from either a user perspective or a system perspective, resulting in a gap between development and use. This thesis takes an overall stand and states that it is possible to benefit from both the user and system perspectives, through collaboration between users, tailors and developers.

This thesis also presents a set of tools to support collaboration on equal terms between users and developers, in the technical design process of evolving the tailorable software and extending the tailoring capabilities. The toolkit aims at building a common understanding of tailoring, supporting democratic agreements and a common understanding of what kind of tailoring to implement. It makes it possible for the users to take part in technical design decisions and have a better understanding of trade-offs and system boundaries.

All of the research is based on field studies including participatory observations, interviews and workshops with users and developers. These studies led to the creation of prototypes and tools that act as mediating artefacts when exploring the research questions.

The contribution of the thesis is twofold. Firstly, the thesis elucidates the need for a cooperative design process to ensure that end-user tailorable software remains useful and sustainable. Secondly, the thesis suggests a toolkit with four different tools to support such a cooperative design process.

ISSN 1653-2090
ISBN 978-91-7295-130-3

2008:03