COMPONENT-BASED TAILORABILITY: TOWARDS HIGHLY FLEXIBLE SOFTWARE APPLICATIONS

Volker Wulf

Institute for Information Systems
University of Siegen
Hölderlinstr. 3
57068 Siegen
volker.wulf@uni-siegen.de

Volkmar Pipek

Institute for Information Systems
University of Siegen
Hölderlinstr. 3
57068 Siegen
volkmar.pipek@uni-siegen.de

Markus Won

Institute for Computer Science III,
University of Bonn
Roemerstrasse 164,
53773 Bonn, Germany
won@cs.uni-bonn.de

Abstract

Flexibility is a central goal in modern software development. Component technologies are perceived as an important means to this end. So far, component technologies support software developers at design time. Especially the experiences made in the design of software for social systems (organisations, communities, …) have raised the issue of implementing 'use time' flexibility (or 'runtime' flexibility) into modern software systems. Approaches to capture this at the interface level do not offer the necessary flexibility. In this paper we present a combined effort at the levels of the interface, the architecture and (to some extent) the organisation using component-based technology to support non-programmers at runtime. Within our approach, appropriately designed component technology can enable non-programmers to tailor software applications according to changing requirements. A component model, called FLEXIBEANS, has been designed with the special notion of highly tailorable applications. The FREEVOLVE platform serves as an environment in which component-based applications can be tailored at runtime. To enable users to tailor their applications by recomposing components, we have developed three different types of graphical user interfaces. User tests indicate the need for additional support techniques beyond a coherent technical base and appropriate graphical interfaces. We present different types of support techniques and show how they can be integrated into the user interface, to enable users' individual and collective tailoring activities.

1. Introduction

The need for flexibility of software systems is well-known and well addressed in the research areas of applied software engineering. Driven by the need to be more efficient in software development, the approaches worked towards better reuse of code and increased comprehensiveness of software architectures (e.g. object-oriented programming, component-based systems, and lately service-oriented architectures). In the research field of Computer-Supported Cooperative Work (CSCW), the driving force of flexibilization is a different one: Software needs to be flexible to be adapted to new or changing work situations during use and in the context of use. It is also end-users, not professional designers, who typically take action to adapt their software infrastructure. Tools, techniques and methods, that have been developed to make an evolutionary software engineering process more efficient, do usually not consider this aspect. Recently component-technologies have gained considerable attention in this context (see e.g. Szyperski, 2002), since they offer 'Black Box' reusability (Ravichandran and Rothenberger 2003) by making the integration of third-party components without knowing/manipulating the code possible. This is extremely important for end-user-oriented

scenarios, since empirical studies showed that users may choose to buy/replace software over re-designing it (e.g. Robertson 1998). Component-based systems can therefore be an interesting starting point in the area of end-user-oriented tailoring.

However, flexibility of software artifacts was a major research issue in Human Computer Interaction from its very beginning. Since the individual abilities of specific users are diverse and developing constantly, suitability for individualization is an important principle for the design of the dialogue interface. In general, users were supposed to adapt the software artifacts according to their abilities and requirements (Ackermann and Ulich, 1987; Fischer et al. 1987; ISO 9241). However, the scope of flexibility realized in early implementations was limited to simple parameterization of the dialogue interface. While this line of thoughts gave the users of software artifacts an active role, high-levels of flexibility concerning the functionality of a system were originally not addressed.

Starting in the late 1980ies, industrial demands, resulting from the spread of personal computers and computer networks, led to research efforts on flexible information systems whose functionality can be modified by their users. While prior work aimed at the individual user, it was now a user group, an organization, or other social entities that needed flexibility in order to adopt computers for cooperative work. Henderson and Kyng (1991) worked out the concept of tailorability to name these activities, and stressed the importance of being able to deeply re-design and re-develop software during use and in the context of use. Software artifacts, commercial products as well as research prototypes, with a tailorable functionality have been developed. Regarding commercial products, spreadsheets and CAD systems were among the front riders. “Buttons” was one of the first highly tailorable research prototypes, where users could change the dialogue interface and functionality on different levels of complexity (MacLean et al. 1990).

With the emergence of networked applications to support collective activities such as communication, cooperation or knowledge exchange, the need for tailorable software artifacts still increased (Schmidt, 1991; Wulf and Rohde, 1995; Bentley and Dourish, 1995). However, the distributed nature of these systems and the potential interrelation of individual activities posed new challenges to the design of tailorable applications (see Oberquelle, 1993 and 1994; Wulf, et al., 1999; Stiemerling, 2000).

Empirical as well as design-oriented research has indicated different challenges in building tailorable systems (Mackay, 1990; MacLean et al., 1990; Nardi, 1993; Oppermann and Simm, 1994; Page et al., 1996; Wulf and Golombek, 2001). We have developed a tailorability framework to capture all relevant aspects.

A framework for end-user-oriented tailorability

The major challenges resulted from the studies and concept explorations in the field of HCI have been firstly to support re-design during use and secondly to allow end-users within their use contexts to take a leading role in re-designing their infrastructures. Especially for the second challenge, important refinements have been described:

1. *Support for tailoring on different levels of complexity:* MacLean et al. (1990) have already pointed out the problems, which would arise, if a considerable increase of users’ skills is required when trying to tailor a software artifact beyond simple parameterization (customization gulf). If users try to modify an application beyond parameterization, normally profound system knowledge and programming skills would be required. Therefore, tailorable applications should offer a gentle slope of increasing complexity to stimulate learning. Different levels of tailoring complexity also tackle the problem of different skill levels among the users.
2. *Support for cooperative tailoring*: Empirical research indicates that tailoring activities are typically carried out collectively (Mackay, 1990; Nardi, 1993; Wulf and Golombek, 2001). System administrators, power users[1] or gardeners are individuals who possess higher levels of technical skills or motivation. Users with less technical skills or motivation may benefit from receiving direct support or reusing tailored artifacts.

We developed a framework to capture all relevant aspects (cf. Fig. 1) which will also guide our presentation of almost a decade of research regarding these issues.

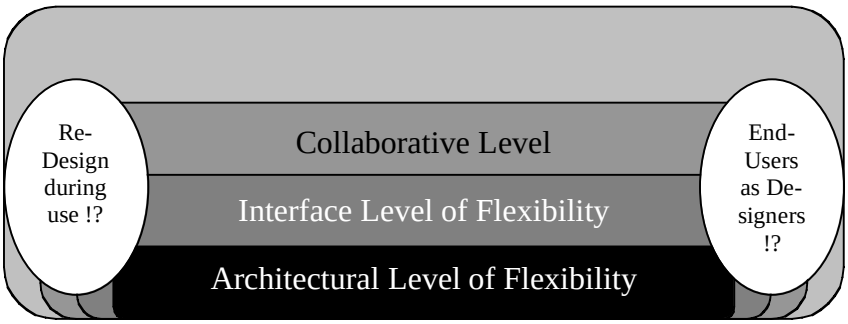

Figure 1: End-User-oriented Tailorability Framework

[1] Very experienced users who are not IT professionals (i.e. computer scientists) and who do not have any programming experience are here referred to as power users.

Our perspective distinguishes four levels of allowing and managing flexibility. For every level we discuss existing ideas against the background of the two core challenges, which are to develop and experiment with new approaches. First of all, the technological foundation of all efforts is of course a certain architectural flexibility. While it can be claimed that the technological flexibility of software artifacts is always given if the source code is available, we are looking for a pragmatic flexibility that respects the challenges of designing during use (or more technically: during run-time) and design by end-users. The pattern continues at the interface level: Easing programming tasks by providing appropriate interfaces has been addressed before, e.g. in research about 'Visual Programming', but providing the right abstraction level and granularity of concepts for an efficient re-design during use and by end-users with very heterogeneous skills poses specific challenges. At the collaborative level the analogy with software development methods starts breaking since collaborating users have very different backgrounds and goals compared to collaborating software designers. While for the professional software designer the focus is on the implementation of a given specification, for the heterogeneously skilled end-user who is designing-in-use, processes of understanding and sense-making of technologies become very important.

In this paper we describe and connect experiences regarding a number of prototypes we developed. All prototypes have been evaluated in laboratory settings or in real use settings, in any case with real end-users from different organizations. Altogether we had about 80 end-users involved in our research, and the evaluations were conducted using the heuristic evaluation method or the thinking aloud method (Nielsen 1993). For some of the prototypes that we were able to test in real world settings, we used questionnaires and semi-structured interviews. For the details about the evaluation of individual prototypes we have to refer the reader to the publications cited with the respective prototypes.

The purpose of this paper is to take a step back and to connect different research efforts to outline the dynamics and problems of providing tailorability for end-users. While many of the individual research efforts have been published before, we now want to shed light on the interrelations of research on the levels described above and on what questions remain open. We now discuss our approaches to end-user-oriented tailoring at different levels. In general we develop a stronger focus on those aspects that do not have a correlation in software engineering. We begin with describing the architectural level and the component-based tailoring platform FREEVOLVE and then proceed to the interface level to describe end-user-oriented solutions to more specific problems. Finally we discuss our work with regard to the work of others and draw some conclusions from it.

2. The architectural level: Component-based Systems as Tailoring Platforms

To some extent the discussion on component technologies in software engineering and the discussion on tailorable software artifacts in CSCW have a similar motivation: the differentiation and dynamics of the context in which software artifacts are applied. However, software engineering directs its attention towards the support of professional software developers during design time, while the concept of tailorability directs its attention towards users during runtime. In the introduction, we already made some points explaining why it is plausible to use component-based systems as a starting point for highly tailorable applications. We now begin with describing our understanding of component-based systems, and then discuss flexibilization research from the field of CSCW later to attain some requirements for our work. In the final two sections of our coverage of the architectural level we describe two specific problems we have overcome by developing the FREEVOLVE platform as a basis for our work on component-based tailorability.

2.1 Basic concepts of Component-based Systems

The term "component" is not used very consistently within the software engineering community. We refer conceptually to Szyperski's (2002) notion of components. He gives the following definition:

> *"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties" (Szyperski 2002, p. 41).*

This idea of accelerating (re-)design time by reusing strictly unitized codes originates in the very beginning in the software engineering discourse (McIllroy, 1968). Szyperski (2002) emphasizes on the economic potentials of collective and distributed software engineering processes. Component technology allows many software developers to apply the same software module in different artifacts. This allows the modules to become independent from each other and to maintain their state towards the outside, except for defined interactions via well-defined interfaces. The developers who apply a component created by somebody else do not necessarily need to understand the implementation details. They

may just access to the services provided by the component via the interfaces. The visibility of the component's implementation can reach from black boxing (no visibility of the source code) to white boxing (full accessibility of the code) with different levels of gray boxing (accessibility of parts of the code) in between.

Such an understanding of component technology has the potential to serve as a basis for the design of highly tailorable systems. Beyond parameterization and reprogramming, the composition of components provides a middle layer for the development of a gentle slope of tailoring complexity at an architectural level. Gray-boxing also covers the case of nested component structures which provide additional levels of complexity differentiation. Further aspects of other levels of managing flexibility already occur here: The level of encapsulation the component concept provides will come in handy on the collaborative level when end-users need to exchange their tailored components. At the interface level, visualization concepts which make the architectural concepts understandable to end-users may profit from the diversity which gray-boxing provides, while the challenge of visualizing behavior remains.

In software engineering, there are many environments which allow modifying or composing components at design time. They generate applications which are monolithic after being compiled. For instance, there are cases with Visual Age for Java (IBM 1998) or Visual Basic (Microsoft, 1996), where the component metaphor is not available at runtime for changing the application (unless it was anticipated and implemented for the application domain by the designers). There are also component-based systems that inspired our own work: The DARWIN system (Magee et al., 1995), for example, is based on a component model that emphasize on the need for comprehensive gray-boxing with a typed event-based interaction and hierarchical compositions. The process-oriented component enactment guaranteed a certain level of flexibility during use. However, the system is intended to be used by administrators, and the motivation to address the needs of end-users leads us to specifically ivestigate into the increase in comprehensiveness and the functional granularity in component architectures.

2.2 CSCW research towards component-based tailoring solutions

CSCW (Computer Supported Cooperative Work) is the research about concepts of and experiences with the development of groupware systems from which we obtain our requirements for a component-based approach to tailoring. A number of approaches for highly flexible groupware architectures have been developed, which we are going to discuss.

In the technological branch of CSCW research, tailorability is an important field of research, and the changing of software during use is one of the main aspects. The Oval system (Malone et al, 1992) is one of the earliest approaches designed for tailorability. The main idea of Oval is that there are only four types of software modules (Objects, Views, Agents, and Links) that can be used for building groupware applications. While the composition of these modules provides a lot of the functionality, it is not sufficiently fine-grained to permit applications building without system-level programming. PROSPERO (Dourish, 1996) is an object-oriented framework that can be used to compose CSCW applications. It offers a number of technical abstractions of CSCW-functionality (e.g. converging and diverging streams of cooperative work) that developers can use as a basis to develop specific applications rapidly. PROSPERO addresses the concerns of developers of CSCW systems, but it does not aim at tailoring by end-users. DCWPL (*Describing Collaborative Work Programming Language*, Cortes 1999) is a framework that allows separating computational and coordination issues when implementing groupware. Groupware applications consist of modules for computation and coordination. The computational modules are connected via coordinating modules which implement the multi-user aspects and are described in a DCWPL file. Thus, several language constructs may be used to describe session management, awareness support etc. As DCWPL files are interpreted during runtime, tailoring is possible by changing the code. Like PROSPERO, DCWPL does not offer a tailoring environment directed to users without programming experience. In the TACTS framework, Teege (2000) has worked out the idea of feature-based composition to tailor groupware. By adding a feature to a given software module, the functionality of an application can be changed during runtime. In his approach the underlying architecture only provides two basic communication styles: a broadcast to all components and a direct connection between two components. The architecture remains open regarding the structure of the messages sent between components. As a result, all semantics have to be defined within the scope of the communicating components. Wang and Haake (2000) present CHIPS, a hypermedia-based CSCW toolkit with elaborated abstraction concepts (role models, process models, cooperation modes, etc.) in a three-level modeling scheme (meta-model, model and instance) which allows users to describe and tailor their cooperation scenarios. Generally, the system which is based on an open hyperlink structure is extendable on any modeling level. Thus, it should be able to support every cooperation scenario that may occur. Wang and Haake (2000) focus on the notion of tailoring as a collaborative activity, and they use the meta-model to give the collaborative tailoring a reference framework.

In these systems and approaches, the authors have emphasized on the following benefits:

- they implement reusable and sharable structures,
- they prefer a building/construction metaphor over the 'text' metaphor of ordinary code,
- it is (with a varying intensity over the approaches) easy to group and re-group functionality at any time
- they define 'building entities' as well as their communication.

These considerations resemble what component-based systems offer at a deeper implementational level. While they would make the use of a component-based approach plausible, there are also some 'specialties' which are not necessarily related to a component-based approach:

- the implementation of domain-oriented building blocks and representations (e.g. Oval),
- the implementation of meta-models that guide tailoring (e.g. CHIPS),
- a defined but flexible component communication (e.g. TACTS).

These aspects reveal that the authors of the concepts aimed at aligning structures at the 'interface layer' with the 'architectural' structures of the building blocks of the CSCW applications. The successful implementation of these approaches and the encouraging impulses from the CSCW literature lead us to consider this as an important additional requirement for building highly tailorable systems.

We aim at building a component-based approach to tailorability which unites the best of both worlds, Software Engineering and CSCW. It should provide a well-founded basis for reuse, encapsulation and composition, while at the same time offer implementational flexibility to provide end-user-oriented interfaces. There are obvious convergences in the ideas and approaches of flexibilization research in CSCW and in component-based systems. However, there are also clear differences we need to deal with when applying component technology to the design of tailorable systems. By definition, tailoring is carried out after system's initialization by users who are not necessarily professional programmers. To allow composition after initialization or even during runtime, new concepts regarding the component model and tailoring platform have to be developed. Since users are the key actors, appropriate tailoring interfaces and an application-oriented decomposition of the software are needed. Moreover, technological mechanisms to support the sharing of tailored artifacts among the users need to be developed.

Three basic requirements for our approach are summed up:

- to provide an architecture for re-designing IT during the use phase
- to provide end-user- oriented concepts and interfaces, and
- to provide a strong congruency between architectural and interface concepts

We will now describe some aspects of how the FreEvolve architecture covers these requirements.

2.3 The FreEvolve Component Platform and Architecture

In dealing with the issues described above, we present results from research conducted at the University of Bonn and lately also at the University of Siegen. The design of tailorable groupware has been an important aspect of our work for almost a decade (e.g. Wulf, 1994; Stiemerling, 2000; Kahler, 2001; Wulf, 2001; Pipek, 2003; Won, 2003; Morch et al. 2004). Beyond component-based tailorability, we also experimented with alternative approaches, such as rule-based architectures and the extension of off-the shelf products. With regard to the latter, we will limit our presentation to those results which are applicable to component-based approaches.

As a technical foundation of their realization, Won (1998), Hinken (1999), and Stiemerling (2000) have developed the FLEXIBEANS component model and the FREEVOLVE tailoring platform[2]. Both were influenced by current technologies in software engineering, especially the JAVABEANS component model and its runtime environment BEANBOX.[3] However, due to the specific requirements of component-based tailorability for distributed systems, we had to refine the component model and to develop a distributed runtime and tailoring environment

2.3.1 Traceability and Intelligibility of Component Systems

Our first goal was to allow a general intelligibility for component system, more specifically, for a good end-user-oriented traceability of state changes of components. The work on the FLEXIBEANS model related to the requirement to achieve a high congruency between the architectural and the interface level.

In general, the atomic FLEXIBEANS components are implemented in Java, stored in binary format as Java class files, and packaged, if necessary, with other resources as JAR-files. One has to distinguish between the component and its instance. Every component can be instantiated in different compositions by different users at the same time. Each instance then has its own state. Like JAVABEANS, the interaction between components is event-based. The state of an instance of a component can only change in case that the instance possesses the control flow, or through interaction with a component that is in possession of the control flow. The composition of the components determines which

[2] In earlier versions the FREEVOLVE platform was called EVOLVE (cf. Stiemerling et al., 1999). The source code is available under GPL at: www.freevolve.org

[3] The component model of JAVABEANS and the source code of the BEANBOX are openly available from SUN and served as a base for early implementations.

instances of components can interact with each other (Stiemerling 1998). In our approach, tailoring on the level of component composition happens through the connection of ports. To allow tailorability at runtime, atomic and abstract (compound) components as well as their event ports have to be visualized at the user interface (see below).

The JAVABEANS component model is based on typed events. Thus, events of the same type (e.g. button click event) are always received on the same port. Incoming events have to be analyzed in the receiving component. This can be done by parsing the event's source or evaluating additional information which is sent with the event. Such an approach makes it difficult for users to understand the different state transitions resulting from events of different sources (e.g. click events from two different buttons). An alternative strategy could be the usage of dynamically generated adapter objects in order to distinguish between different event sources (e.g. click button). Such an adapter would forward different events of the same type (e.g. different button click events) to different handling methods according to the event source. Both strategies hide a part of the real components' interaction. An appropriate understanding of these strategies is essential for enabling the user to compose components appropriately. Therefore the FLEXIBEANS component model has been developed to allow named ports, which allow distinguishing ports according to their types *and* their names. Connections between components are only valid if the port's type and name match. Furthermore, well selected names of ports support an appropriate understanding of a component's semantics and its role within entire assembly. Ports of the same name could have different polarity in the sense that they could either emit or receive a certain type of event.

2.3.2 Composition Language

In traditional software engineering approaches, components are only visible at design time, and composing is done by means of a programming language. Since these approaches do not allow any change of the component structure at runtime, the component structure gets lost after compilation. If the reconfiguration of components during runtime is allowed, the first challenge would be the design of not only a language that describes the composition of atomic components, but also of concepts to maintain these representations of component networks. The CAT[4] component language as a part of the FreEvolve concept is not only developed to concerning these issues, but also allows describing compositions of atomic components in complex hierarchical structures. A CAT file, or in the distributed setting a set of CAT files (see Fig. 2), describes such composition.

To allow different levels of tailoring complexity, hierarchically nested component structures are supported by the CAT language. Complex components can be built and stored by composing a set of atomic (or complex) components. From the user's point of view there are two advantages: (a) the necessary number of abstract components to build the final application is lower and (b) their design can be more application-oriented. Thus, the necessary composition activities have a lower level of tailoring complexity than the composition activities at the atomic component level. The CAT language allows an arbitrary depth of nesting.

2.3.3 The Distributed Tailoring Platform

Our focus in this contribution is on supporting (end-) user interaction, neglecting some software-technological difficulties we had to face during the development of our approach. Those difficulties, especially during the implementation of the distributed features of our concepts, and the realization of the dynamic reconfiguration of component networks, are more extensively covered in (Stiemerling and Cremers 1998, Stiemerling, Hinken, and Cremers 1999, Stiemerling, Hinken, and Cremers 1999a). Here, we only give a brief presentation on our basic concepts. The FREEVOLVE platform allows tailoring distributed applications with a client-server architecture (see Figure 2).

[4] The syntax of the CAT (Component Architecture for Tailoring) is described in Stiemerling (1997). It draws on concepts already developed in port-based configuration languages such as DARWIN (cf. Magee et al. 1995) and OLAN (cf. Bellissard 1996).

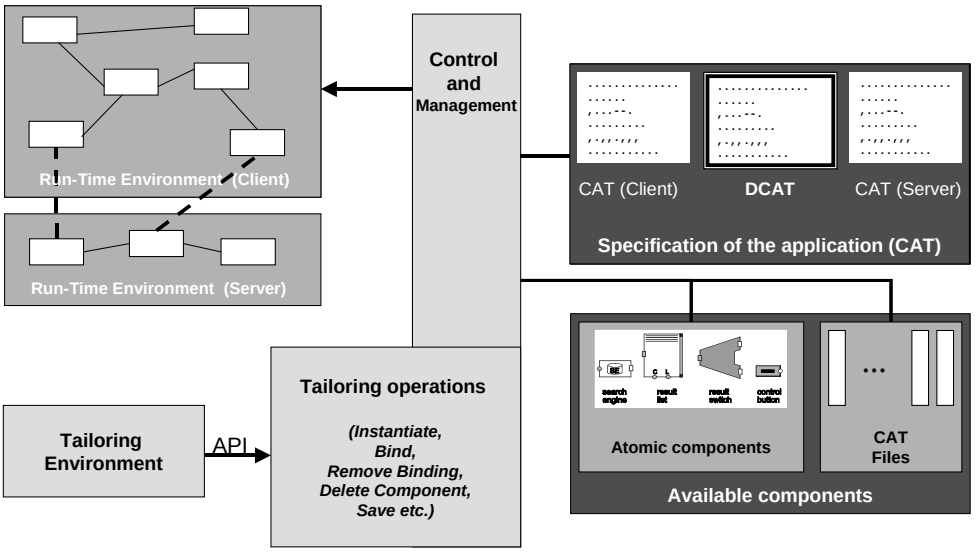

Figure 2: FREEVOLVE Platform

In the beginning, the tailorable application is stored persistently on the server. The atomic components are stored in JAR-files, and the current component structure is stored in a set of CAT files and *remote bind* files (DCAT files)[5]. The user management provides the appropriate CAT and DCAT files for each user starting their instance of the application. To support cooperative tailoring, any specific composition of the component can be shared by different users. The implementation does not allow run-time reconfiguration of the server-side composition.

During start-up, the CAT files on the server are being analyzed. All relevant components are represented at server side. When user A logs in and starts the client of the tailorable application, the client connects with the FREEVOLVE server. The server authenticates the user and sends the necessary components to the client. The client locally instantiates the atomic components and connects them according to the composition described in a CAT file. If the server application is not yet started, it is then instantiated in the same way. Finally, client and server are connected according to the DCAT information.

The CAT files for the client sides are stored centrally on the server which allows different users to run the same client by applying the same CAT file. Therefore, changes on the client side are transmitted to the server, stored persistently, and propagated to those active client machines that use the same client. However, with that type of distributed architecture typical synchronization problems occur: The propagation of tailoring activities to other users may lead to inconsistent system states in the FREEVOLVE platform. A specific protocol has been developed which guarantees in case of a breakdown, and a completely consistent version of the application can then be recovered (Stiemerling et al. 2000).

This architecture provides a well-organized structure of component-based CSCW applications even for multi-user setting. It allows tailoring during the applications runtime, and it prepares for a higher level of congruency at the interface level.

3. The Interface Level: Component Visualisations for Tailoring Platforms

There is a large body of work regarding the 'ergonomics' of programming, mainly in the field of Visual Programming (e.g. Shu 1988, Myers, 1990, Ambler and Leopold 1998, Pane et al. 2001). While many approaches aimed at supporting the professional programmer, e.g., by aiming at visualizing the whole program and all its facets, there is also some work referring to supporting less experienced programmers, which aims at visualizing only what is necessary, and only in an appropriate way.

[5] To allow tailoring of client and server independently, we use three CAT files to describe one application. Two CAT files are needed to describe each the client and the server configuration of the components. The third description file, called DCAT file, describes the remote interaction between client and server components. Thus, every client application can be tailored without affecting the (common used) server.

Repenning et al. (2000) built a simulation system in which cooperative agents can be programmed by end-users, and which offers a visual programming language that allows changing the behavior of single agents. The interaction between the agents remains hidden and cannot be tailored, which limits the flexibility of possible applications. The approach did not aim at providing a universal highly-tailorable infrastructure, e.g., the programming language Visual AgenTalk needs to be re-designed for each field of application in order to be understood more easily.

The Regis system (Magee et al., 1995) allows distributing configuration management. Its 2D environment implements many important techniques from Visual Programming. However, the underlying component model is based on the idea that a composition consists of few components only, which makes it less scalable.

Some of these tools use multiple views to focus on different aspects of the application. For instance, in most programming environments, there is a code view and a GUI view. In the field of end-user tailorability, Morch and Mehandjiev (2000) use this technique: Their system ECHOES allows tailoring applications that can be seen and changed in different representation views.

The HCI discussion has developed concepts which support the learning of the functionality of single user applications. These concepts are based on either structuring, describing, experimenting or exemplifying the use of certain functions (e.g. Carroll and Carrithers, 1984; Carroll, 1987; Yang, 1990; Howes and Payne, 1990; Paul, 1994). However, these approaches remain quite limited in the scope of competencies they are able to communicate, as they usually have only one function. Within the discussion on tailorability (starting with Henderson and Kyng (1991)), technology is not regarded to be the only important aspect for a user to be understood. By contrary, the complete socio-technical system, in which the user, the technology and the other users are involved is considered to be central.

In our research we made use of the knowledge from the Visual Programming community, but we will discuss that later on. Now we describe more dynamic aspects of end-user orientation. It is, from our point of view, not merely the visual experience that is the main factor for allowing end-users to familiarize easily with the manipulation of IT artifacts. It is in fact, the way how users are supported to detect, act and play safely with the configuration options which allows them to learn about and grow into the role of a 'lay programmer'.

We have collected the experience we and other researchers in the field of CSCW have made (see above), and followings are the main challenges for these 'dynamic' aspects:

- *Consistent Anchoring*: The options to tailor a software artifact need to be indicated consistently,
- *Intelligibility* (of composition structures): The current composition structure of a tailorable software artifact (component aggregate) has to be represented intelligibly, especially in the dynamic aspects,
- *Effect Visualization*: The effects of tailoring activities have to be easily perceivable for the user,
- *Fault Tolerance*: The tailoring environment should be fault tolerant in the sense that it indicates incorrect activities to the users and proposes advice.

We now describe concepts and prototypes dealing with these challenges. Firstly, we begin with the concept of Direct Activation which we developed to provide consistent anchors for tailoring functionality in component-based applications. Secondly we describe issues of congruencies between different aspects and layers of which we can elaborate on providing users an easy access to tailoring functionality. Thirdly we suggest 'Exploration Environments' to increase users' understanding of the dynamic aspects of the applications they tailor. Finally we address the issue of Fault Tolerance by suggesting additional constraint-based advisory structures to assure users that important dependencies within the component compositions will be guaranteed.

The work on the visualizations within the tailoring interface of FREEVOLVE was carried out by Won (1998 and 2003), Hallenberger (2000), and Krüger (2003). Additional features to support tailoring activities were realized by Engelskirchen (2000), Golombek (2000), Wulf (2001), Krings (2002) and Won (2003).

3.1 Easy Access to Tailoring Functions: The concept of Direct Activation

An empirical study of users of a word processor indicated that, finding the appropriate tailoring functions is a substantial barrier that either prevents tailoring at all or adds significantly to its costs (Wulf and Golombek 2001). Discussing our findings in the context of earlier work (see Mackay 1990; Page et al. 1996), we identified two rather different occasions when users want to tailor an application: (a) when a new version of the application is introduced, and (b) when the users' current task requires a modified functionality. The users need different patterns of support in order to find tailoring functions in both of these occasions.

For tailoring a newly intrduced version of an application, a survey of the given tailoring functions would be an appropriate means to tackle the finding problem. The users get informed about the scope of the new version's tailorability. When the users' current task requires a modified functionality, a context specific representation of the tailoring functions' access points seems to be appropriate. In such situation, the user typically knows which aspects of the application he wants to modify, since the current version of the function hinders his work.

In order to tackle the second case, we have developed the concept of *direct activation*. Tailoring is needed when users perceive a state transition that does not lead to the intended effects following a function's execution. In this case, users are typically aware of the function's anchor at the user interface. Therefore, the anchor of the tailoring function should be designed so that it 'relates' to the function to be tailored.

In our concept three ways of 'relating' tailoring and use are distinguished. Firstly, by *Visual Proximity to the use anchor*, where the visual representation of the anchor of the tailoring functions is placed closely to the anchor of the tailorable function. For example, in case certain parameters of the tailorable function have to be specified during activation, visual proximity can be reached by displaying the anchor of the tailoring functions next to the one for specifying the parameters (e.g. in the same window). If the tailorable function is executed without any further specification from the menu or via an icon (a 'use anchor'), the anchor for the tailoring function could then be placed next to the one of the tailorable function. However, this does not work for what we call 'triggered functions', functions that are not activated directly by the user, but initiated by some state changes in the application (e.g. email filters, Oppermann and Simm 1994). Since they do not provide a visible use anchor, but nevertheless cause perceivable effects at the interface usually, we suggest a *Visual Proximity to the effect anchor* here.

A third, completely different approach within our framework completely forgoes visualisations of anchors, but provides a *Consistent Mode Change* to the tailoring interfaces of functions instead. Mørch (1997) gives an example of a consistent mode to activate a tailoring function. In his system, a user can access different levels of tailoring functions by activating the tailorable function and pressing additionally either the "option", or "shift", or "control" button. Restricted to specific functions, the Microsoft context menu gives another example of how to design a *Consistent Mode Change* to activate tailoring functions. Whenever the display of a screen object may be tailored, a specific mouse operation on this object allows accessing the tailoring function.

To evaluate the effectiveness of the concept of *direct activation* in finding tailoring functions, we have implemented prototypes and carried out an evaluation study. The results of this study show that direct activation eases tailoring activities (see Wulf and Golombek, 2001a).

3.2 Visual Tailoring Environments: Exploiting Congruencies

The general design of the tailoring interfaces in component-based systems aims at allowing "natural" tailoring as Pane and Myers (2006) postulate. In fact, our goal is to create a visual tailoring environment, where users are able to match between the interface at runtime and the one at design time. So the users are supported to identify the aspects, which they want to tailor, easily.

The characteristics of components and the composition metaphor inherent in the concept make a graphical/visual tailoring interface appear appropriate for changing the component structure. Within the graphical tailoring environment, the component structure of the software artifact is displayed as follows: instances of components are visualized as boxes, ports are indicated as connectors at the surface of these boxes, and the binding between two ports is represented by a line between these surface elements. Tailoring activities consist of adding or deleting (instances of) components and rewiring their interaction. During the course of our work we have developed 2D (Stiemerling and Cremers 1998, Won 1998, Wulf 2000) and 3D versions (Stiemerling et al. 2001) of the visual tailoring environment. In the following we neglect the issue of parameterization of single components, which we have realized in all three of the tailoring environments by means of Direct Activation (see above).

While we heavily made use of well-known ideas from 'Visual Programming', our main consideration was more directed to find 'plausible congruencies' that would make it easier for end-users to achieve their tailoring goals. We begin with an example (see figure 2):

The first tailorable application we developed was a search tool for a groupware application. The tailorable aspects were restricted to the client side, while the groupware application itself had a client-server architecture. Figure 2 shows the 2D graphical environment to tailor the search tool.

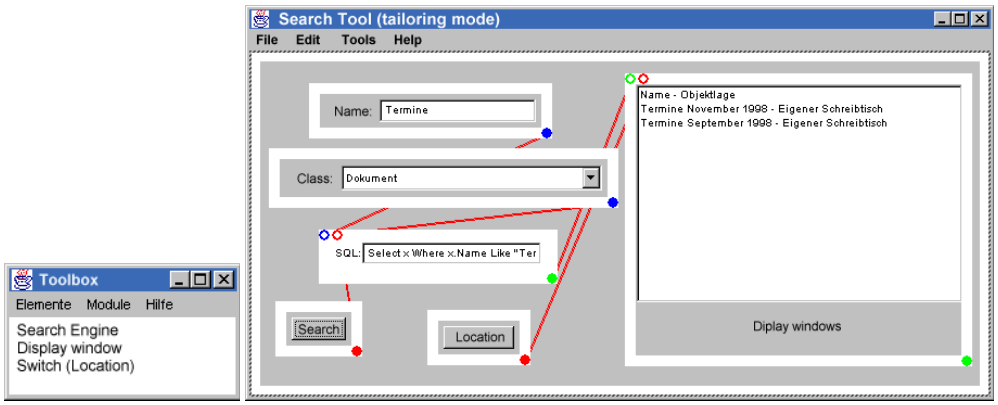

Figure 2: Two dimensional graphical tailoring environment

The general approach is pretty straightforward: The users were able to compose different variations of the search tool window, which basically consisted of different graphical elements to specify search queries and different graphical elements to display search results. To allow these tailoring activities, the search tool consisted of six types of atomic components. Four of these component types were visible during use (inquiry elements, start button, display elements) while the other two were visible while tailoring (the search engine and switches to direct search results towards specific graphical output elements). We made use of the categorizations of ports to represent them at the tailoring interface. The polarity of ports helped to distinguish between a component's input and output port: empty circles indicate input ports, filled circles indicated output ports. To support users in wiring the components appropriately, ports of the same type and name are given the same color, so that the users are hinted to fitting input and output ports by means of identical colors. Wired components are displayed by a connecting line between their corresponding ports. Abstract components are represented by a white frame around the atomic (or abstract) components they contain. If a power user has already designed several different abstract input and output components, other users can make first steps in constructing their personal search tool by combining two abstract components with each other.

Although the application resembles many other approaches to visual programming, it is important to us to clarify that we consciously aimed at exploiting 'plausible congruencies' here. The first congruency is the one between the metaphor world of components and their visualisation (*Component-Metaphor-Congruency*). Component-based systems provide a specific perspective on the way software systems are built and how they can be manipulated. Attached to this perspective is a certain language (components, ports, connectors, etc.), and a set of related concepts. From our point of view, the art of providing tailoring interfaces is to maintain this congruency, while a certain domain-orientedness (e.g. by using appropriate naming schemes) is providing at the same time. This allows users to understand tailoring options easily while they are still getting in touch with the concepts of the software architectures world they are working with.

The second congruency we exploited to support end-users is the *Architecture-Interface-Congruency* that we already mentioned at the architectural level (see above). It is, from our point of view, important that visualizations reflect the actual workings of the component architecture. Therefore, in the FLEXIBEANS concept we did not only provide typed and directed, but also named ports to make the information and control flow among components as intelligible as possible. The example above also shows that 'gray-boxing' which is also considered as important. However, other problems remain unsolved: in approach described above, it is still difficult to differentiate the scope of tailoring activities referring either to the client or to the server side.

The third important congruency we elaborated on is the *Tailoring-Use-Congruency*. The idea is simply that the tailoring interface users should still be able to recognize the familiar use interface with the corresponding visual anchors of functionalities. This, however, has proved to be quite difficult and led us to conduct experiment with different two- and three-dimensional (2D and 3D) interfaces. The 2D approach presented so far has the advantage of enabling users to match directly between the runtime (use) environment and the tailoring environment. When a user changes into tailoring mode, the visible components of the interface stays at the same place on the screen, while the invisible components, the ports and the connecting lines between them are added to the display. The components which are invisible during runtime (search engine, switches for the results) are displayed at the same locations where they have been placed during the prior tailoring activities. However, a strict adherence to the third congruency has the consequence that locations, where invisible components are placed during tailoring, could not be used by visible components during use. This approach does not make efficient use of the screen space as long as larger parts of the invisible functionality stay "behind" the user interface.

To overcome these problems, a 3D graphical tailoring interface, as presented in Figure 3, has been developed. An application here can be explored by spatial navigation ('flying through the model'). Components here are represented as three-dimensional boxes (with the component's name above them) which are located on a virtual plane. The ports are represented as rings around the components in order to facilitate connections from all directions. Like in the 2D case, the color indicates the type and name of the port. The polarity is expressed by the intensity of the color. The input port

is represented by darker shading, while the output port is indicated by lighter shading. Connections between components are represented by tube-like objects linking the corresponding rings.

Abstract components are represented like all other components by 3D boxes. However, the box's surface that encapsulates the containing components of lower hierarchical level becomes more transparent as the user navigates closer to it until the visual barrier finally disappears completely. However, the rings representing the ports of the abstract component remain visible. The user can now navigate or manipulate the inner-component structure. In our current implementation atomic components remain black boxes even if the user navigates into their neighborhood. A gray-box strategy would allow navigating into an atomic component and inspecting those aspects of the code that can be modified.

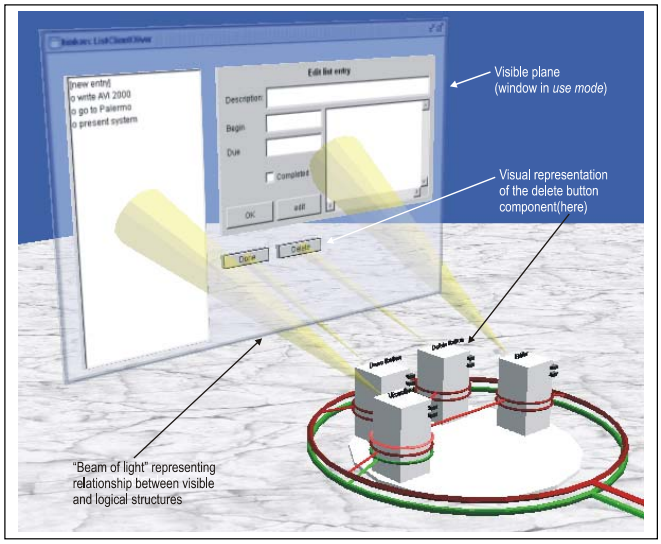

Figure 3: 3D graphical tailoring interface displaying a client's tailorable component structure

The distinction between client and server side is represented by a spatial arrangement which places the server in the center (represented by its tailorable component structure), while the different clients (represented by their tailorable component structure) are allocated in a semi-circle around the server.

In order to ease the transition from use into tailoring mode, we offer a reference between the visible components at the user interface and the invisible components "behind" the screen. If a user changes into the tailoring mode, the actual client's GUI window will be projected into the 3D world. Light beams will then connect the specific elements of the GUI interface to the client's component structure which they refer to. When the user starts tailoring and enters the 3D world, he will be located in front of the GUI projection and he will see the GUI of his regular interface. However, the semi-transparent plane allows observing the component structure on the highest level of abstraction as well. Following the beams of light he can navigate through the plane into the 3D space and explore or change the component structure.

With the development of the 2D- and 3D-solutions, it becomes obvious that the congruencies that we want to establish may conflict with each other. In our implementations it becomes obvious that the Architecture-Interface-Congruency is better to support in the 3D-environment while the Tailoring-Use-Congruency is better to implement in the 2D-environment. In user studies we identified several problems with the 3D tailoring environment. Users had difficulties to navigate, and even after some practices, the time for reaching certain points in the architecture was considered too long. Moreover, the spatial arrangements lead to situations, which users might feel that it is impossible to navigate, in a way that all relevant information (e.g. including the representation of the use mode) are visible on the screen. Our empirical evaluations also indicated that users have more problems in understanding the functionality and the use of invisible components (more abstract to them) than of the visible ones (this applied to the 2D- and the 3D-interfaces).

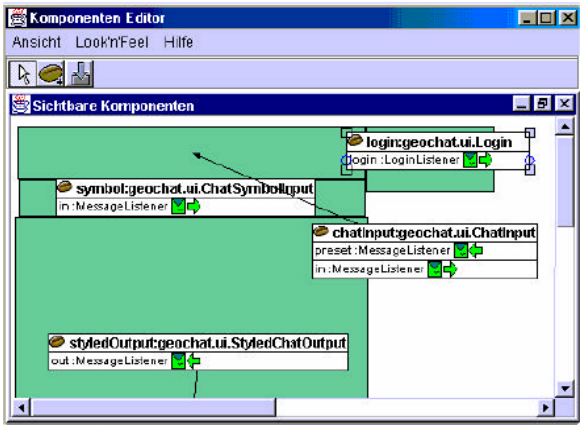

Figure 4: View of Visual Components

The latter result leads us to the development of a hybrid solution between 2D and 3D that aims at addressing both congruencies separately. In the third prototype we split the tailoring mode into three 2D-visualisations in separate windows. One window maps the Tailoring-Use-Congruency by visualizing visible components only (and making only them available for tailoring, see figure 4), whereas the other two map the Architecture-Interface-Congruency by providing a 2D-visualisation of the component bindings on the client side (Figure 5, Composition Editor), and by providing a tree view of the complete hierarchy of components (including server components, see Figure 7, Component Explorer). All three windows are synchronized in order to make sure that all of them represent the current state of composition.

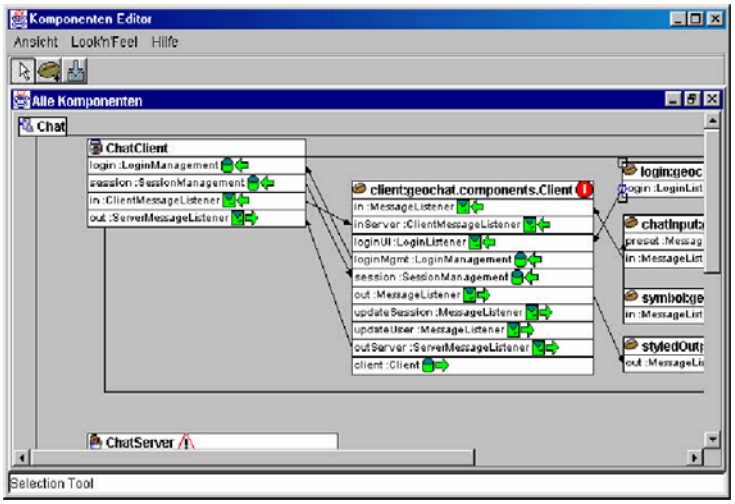

Figure 5: View of the Composition Editor

For easier orientation and navigation, the synchronization feature also covers the highlighting and marking of components, i.e., highlighting a component in one view leads to the component being highlighted in the other views simultaneously. The distinction of different views encourages users to use the views for different tailoring tasks: Resizing and/or re-arranging the components at the interface level are done in the editor for the visible components, architectural modifications are done by using the other editors. Changes of component parameters are possible in all three editors via context menus. In this approach, the notion of access control of the modification of components is also picked up by being able to deny access to the server-side components within the *Component Explorer*.

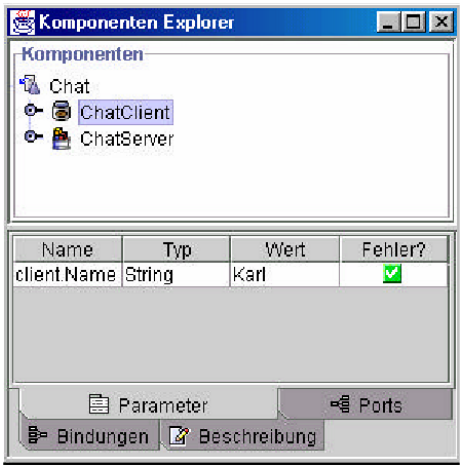

Figure 6: View of the Component Explorer

So far we have only carried out a very preliminary evaluation study (see Krüger, 2003) which indicates, at least the experienced users are able to work with all three different views provided by this tailoring environment.

3.3 Fault Management: Checking the Integrity of Component Compositions

Empirical studies indicate that the fear to break an application is one of the major obstacles in tailoring activities (Mackay, 1990). Thus, we have to consider the case that users may make mistakes when (re-)composing component structures. While these errors, on the one hand, threaten the functioning of the application (and even worse, the applications of others if the users work in a shared distributed environment), they can be regarded as opportunities for learning on the other hand (see Frese et. al., 1991). Although the differentiation of the ports (see FLEXIBEANS concept, typing, naming, directedness) help preventing certain misconnections among components already, we additionally developed technical mechanisms that detect errors actively in the composition of components (Won 2000, Won 2003).

Such mechanisms for integrity check should control the validity of the composition, indicate the source of an error, give hints, or even correct the composition. The rule-based integrity check which we present here consists of two different concepts: (a) constraints and actions and (b) an analysis of the event flow.

Rule-based integrity checks are well-known in data base management systems (Silberschatz et al., 2001). Rules are terms in first order logic that can be evaluated automatically. This technique can be used to add external conditions to the use of components. By restricting the use of a component, these constraints describe the “right use” of them. For instance, if we have a set of GUI components, we can formulate a constraint like “all interface components have to have the same look and feel”. If a user tailors an application (i.e. adding a new component) this condition can be checked.

The integrity constraints of all components are stored externally. Thus, they can be changed over time according to the users’ or the organization’s requirements. This separation may be useful in case standard components are added to the component framework. Parameters can be restricted with regard to the domain in which the component should be used. For instance, we introduce a watch component. This component is designed to work all over the world in order to support all time zones. Our domain in this case might be a regional organization that is set in Western Europe. Thus, only one or two time zones are needed and in this case, external constraints can ensure the time to be set to MET only.

Constraints include action descriptions. Action descriptions reach from providing simple text information which may help the user to correct a wrong input, or to describe an automated execution of solutions. Those action descriptions may cover all tailoring operations as discussed in our description of the architectural layer. Nevertheless, our goal here is not to compose applications automatically, but to ease the learning and understanding of tailoring activities. All the systems’ interaction mentioned so far are represented at the user interface as they happen. As soon as a constraint fires, the corresponding component will be marked and the user may get detailed information by clicking it.

A second technical mechanism is the check of event flow integrity. As mentioned before, the FLEXIBEANS component model allows event-based component interaction, in which events are passed between independent components. In most cases, events which are created and passed to another component can be regarded as important

information. Therefore an event that is produced, but not consumed, may indicate a problem or an unsolved issue. Here is a short example which explains that such an integrity rule might not be sufficient: Figure 7 (a) shows a three-component-composition consisting of a search engine (left), a filter component (middle) and an output window (right). If we mark consumers and producers in our constraint set, we can only ensure that the search engine and the output window are connected (in some way). What we want to ensure, however, is that each important event which is created is also consumed in the right way. Thus, the event flow needs to be checked additionally.

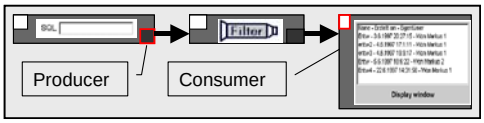

Figure 7: Example of an Event flow

In order to implement the concept of event flow integrity, we classify the ports into *essential* and *optional* ones: essential ports have to be connected to other components whereas optional ports may not be used. For instance, the output component of the search tool, which displays the found objects' names (see above), has two ports: one input port that is used for receiving search results from the search engine, and one output port that sends additional information (type, attributes) of selected search results to other components. In order to support users in building functional applications, the input port is classified as "essential", since it would be senseless to compose a search tool whose input window is not connected to any search engine. The output port here is classified as "optional", as one can use an output window without applying any additional information of the search results.

In order to deal with these dependencies, we have developed a 'regular providers/consumers' concept. For checking an event flow, all essential event ports have to be connected to 'regular providers' or 'regular consumers'. For instance, we may compose a search tool by connecting a search engine to a filter component (that filters some of the search result) that is connected to the output component. Here, the search engine has an essential output port. The filter component only passes events but it is not a consumer. Whereas the output component is a consumer of the search result and has an essential port that is finally connected. Thus, we have to differentiate ports according to their function within a composition (producer, consumer, pass-though) and their need to be connected (essential, optional).

This information is described in external XML files which can be changed by expert users only. The integrity check is carried out by translating the composition into a Petri-net and then analyzing it (van der Aalst et al., 1997; Won, 2003). In case the event flow analysis detects an error, the user is provided with corresponding information within his tailoring environment. Components and their ports are marked if they are essential but not connected correctly.

3.4 Providing safe sandboxes: Exploration Environments

Beyond a technical support during composition, we have also developed exploration environments to allow users to test the dynamic behavior of their tailored artifacts. As a result of empirical investigations, Mackay (1990) and Oppermann and Simm (1994) already pointed out the importance of explorative activities. While research in HCI has already led to different exploration mechanisms (e.g. undo function, freezing point, or experimental data), the distributed character of groupware poses new challenges. Users are often unable to understand the way groupware functions work, since they cannot perceive the effects of the functions' execution at the other users' interface (e.g.: the access rights granted to somebody else cannot be perceived by the owner). Only in cases where an application follows the WYSIWIS (What You See Is What I See) principle in a very strict manner (see Johansen 1988), users can perceive the effects of a function's execution on the interface of other users. In field studies (Wulf and Golombek 2001a), we learned that users try to overcome these problems by testing the results of their changes collectively. For instance, if users try to understand the current access right settings, they ask their colleagues to try to open specific documents. This kind of exploration is often disturbing and difficult to do, especially if the users are not collocated. To understand how tailorable functions are configured becomes even harder, in case their execution is not directly visible at the user interface. This type of changes in the configuration cannot be recognized directly. For instance, changes to an email filter can only be explored by sending a variety of different emails and tracking the recipient's mailbox.

We have developed the concept of exploration environments as an additional feature to support users in experimenting with tailorable groupware (Wulf 2000, Wulf and Golombek 2001). An exploration environment allows simulating the execution of a tailorable groupware function by means of a specific system mode, in which the learner's own user interface and the effects on other users' interfaces are simulated on the output device. If a function is executed in the exploration environment, the effects of its execution on the simulated user interfaces will be similar to the effects the "real" function's execution has on the "real" user interfaces. In most groupware systems users are able to see parts of the system only (their personal desktop, documents they are allowed to view etc.). Exploration environments have two main advantages: Firstly, a user can switch between different users' interfaces. By executing a function in the exploration environment and switching between the simulations of his owns and the other users' interfaces, a user can perceive how a newly tailored function works. Secondly, all data and documents of the exploration environment are accessible to the user. Thus, for instance, if a document cannot be found with a tailored version of a search tool, the user

will know that he has to change the configuration of the search tool as he can see that a corresponding file exists in the exploration environment.

The major challenge for exploration environments is that, with our approach we do not remain in the realm of technology. To allow credible and realistic expeditions into the possibilities of alternative technology configurations, it is necessary to model at least parts of the organizational system, e.g. users, departments, roles.

We have built specific exploration environments for three different tailorable groupware tools: an awareness service, a search tool for groupware, and a highly flexible access control for a shared workspace. To evaluate the effectiveness of exploration environments in tailorable groupware, we have carried out a field study and an experiment in a lab setting. The results of these studies indicate that exploration environments support tailoring activities in groupware (see Wulf 1999; Wulf and Golombek, 2001a).

4. Cooperative Tailoring

Component-based software engineering is based on the assumption that software development can be organized best in a collective and distributed manner. Component repositories together with monetary compensations for those who offer their source code for reuse are supposed to render software development more efficient (see Szyperski, 2002). On the technical side, the issue is being addressed by providing a certain level of encapsulation of functionality, and by allowing an easy exchange of components. Empirical studies on tailoring activities of end-users have also revealed a collective nature of tailoring (see Pipek and Kahler 2006). While monetary compensation does not play an important role in these collaborations among users, different patterns of cooperative activities have been found among users who differ in their commitment to and qualification for tailoring.

Regarding tailoring, we deal with two ideal types of social design relationships: A relation between professional designers and users, and a relation between users that cooperate in the reconfiguration of the technology they use. In the FREEVOLVE approach, we anticipate that a certain division of labor would go with this distinction: On the level of the atomic components, the code is provided by professional developers to users, while on the level of the abstract components, users cooperate by providing each other with pre-integrated abstract components. Especially referring to the second type of relationship, we experimented with the design of shared workspaces to exchange tailored artifacts among users. Moreover, additional features may add significantly to a component presentation for an intuitive reuse by others.

Engelskirchen (2000) and Wulf (2001) have developed a shared workspace to exchange abstract components within a groupware application; Höpfner (1998), Golombek (2000), and Kahler (2001) have developed a shared repository to exchange tailored artifacts, such as document templates or button bars, among users of a word processor. Stevens (2002) has worked on metaphors that make visible and invisible components better understandable to users.

Repositories in software engineering are expected to contain components which could construct a wide variety of different applications. The necessities of navigation, understanding of technological context, and choice of appropriate components are part of the professional knowledge of software engineers. Our experiences with end-users (Wulf 1999, Kahler 2001a, Stevens and Wulf 2002) revealed the need for a more application-oriented and user-oriented approach in designing repositories for the exchange of tailored artifacts. To allow a seamless transition between use and tailoring activities, the shared repository needs to be integrated into the tailoring environment and should be activated directly with only those components that are relevant to a certain tailoring context.

Repositories to exchange tailored artifacts can be understood as shared workspaces, where compositions of components can be exchanged. Shared workspaces have been discussed in CSCW research (e.g. Bentley et al., 1997). In our case, however, shared workspace functionality needs to be integrated in both the groupware application and the additional features for making the components intelligible. Their basic functionality consists of functions to upload and download compositions of components. Additional features allow specifying the visibility of compositions and defining access rights for different subgroups of users. Moreover, a notification service informs users as soon as relevant tailored artifacts are newly produced, modified or applied. Such service contributes to the mutual awareness of distributed tailoring activities and may encourage the emergence of a tailoring culture (see Carter and Henderson 1990). Since direct cooperation among the users should be supported as well, the tailoring environment may provide a function to mail tailored artifacts directly to specific users or groups of users.

We have identified several design aspects for cooperative tailoring tools that support users in understanding the technology and its tailoring options as well as their own tailoring activities. Now we will discuss the use of appropriate naming and classification schemes, following with the necessity of annotations, then we will address the social aspects of exploration environments.

4.1 Naming and classification schemes for components

Users are typically confronted with a variety of different atomic and abstract components. By offering numbers of atomic or abstract components, one has to provide meaningful names for them which are listed at the interface. Due to limitations of screen space, the users' first choice is based on a very sparse visual presentation of the listed items, where naming the components in a meaningful way has turned out to be of central importance for the efficiency of tailoring activities.

Naming may be still rather straightforward regarding atomic components because they usually have only limited functionality that is potentially easier to describe in a name. However with abstract components in a shared repository, the functionality will be more complex. Therefore, users have to agree on a shared vocabulary in describing tailored functionality. Some issues can be resolved by providing additional representations for invisible components, which explain their functionality (cf. Wulf 1999). It can be also useful to work with a defined set of metaphors that are related to the application field (cf. Stevens et al. 2006).

In order to structure the set of components, a classification scheme is essential. Again, the classification of atomic components can benefit from the simplicity of their purpose. For instance, in FREEVOLVE we group them according to their ports (names, types, input/output) because we assumes that such a classification indicates the roles they can play within the composition good enough (cf. Wulf 1999). Nevertheless, for classifying abstract components, there are many plausible categories, e.g. technical aspects like the type of ports they use, or the organizational unit of the author of a component (used as an indicator for use/task similarities covered in a set of components). However, the most important aspect with regard to naming and classification scheme is to recognize that they may evolve. Thus, it is important that names as well as classifications can emerge and develop as the users find and improve a common understanding.

4.2 Annotation of Components

A field study indicated that users need additional support in distinguishing components beyond naming and classifying. Hence, we generate possibilities to describe atomic and abstract components textually. To describe atomic components, we have created a hypertext-based help menu in which help texts briefly explain the components' functionality and screen shots are added if necessary.

Descriptions of abstract components have to be created by the users themselves. Therefore, we have implemented an annotation window which consists of the following text fields: "Name", "Creator," "Origin", "Description", and "Remarks". Since textual documentation of design rationales imposes extra burden, it is often omitted (c.f. Grudin, 1996). Thus, we tried to reduce the workload by providing automatic support in creating the descriptions where possible (cf. Wulf 1999). For instance, the "Creator" field is generated automatically by data taken from the user administration. The "Origin" field contains a reference if an abstract component is created by modifying an existing one. This reference is created automatically as well. The "Description" field clarifies the functioning of an abstract component. The text creation is supported in case the component results from the modification of an existing one. In this case the explanation of the original component is copied and put into italics to be modified. The "Creator" field has an important role. On the one hand, it may indicate the quality of the abstract component. A composition becomes more trustworthy if it is created by someone who is an accepted expert within her community. On the other hand her efforts become visible to the group, which leads to a higher degree of acceptance in many cases. Moreover, it may increase the motivation for sharing tailored artifacts as a service to a community of users.

4.3 Experimenting with components

Static descriptions already add to a better understanding of the functionality which components offer. Still it is a complex task to understand how a new component tailored by other users interacts with other known components. While the exploration environments described above support experimenting with completely assembled functionality, we have to find new approaches to support experimenting with atomic or abstract components. In case these modules do not cover an observable set of functionality, they cannot be executed in an exploration environment by themselves. Therefore, we have implemented an option which allows the users to store the "missing parts" together with the corresponding atomic or compound components. Together with the components themselves, the "missing parts" should provide a characteristic example for the component's use when building functionality. Those examples can then be executed in the exploration environment (cf. Wulf 1999).

5. Discussion

We have described concepts and prototype that we developed in order to provide tailoring means to end-users. While many prototypes have been published in more detail before, our overview allows us to take a step back and have a look at the big picture.

The purposes of our research are twofold: On the one hand we want to allow end-users to be able to tailor their technological infrastructures by themselves, and on the other hand, we want to allow them to do the tailoring at runtime. When we started with the research agenda, our choice of the component paradigm was motivated by its resemblance to

tailoring concepts that have been developed in the field of CSCW. During the course of our work we have developed more than 15 prototypes and have evaluated them with about 80 end-users from different organizations. The conceptual knowledge that we presented here emerged from this research.

As a general outcome of our research, we can comment that the task of tailoring that we aimed at supporting is much more complex than we originally thought. From our point of view it is also more complex than the task of (professional) programming. The reason is that tailoring is not mainly about product development (which is often the guiding metaphor in software engineering), but about infrastructuring. We take here Star and Bowkers' (2002) notion, which define an infrastructure as something that 'runs underneath' other structures, which usually is invisible and that becomes visible in the case of a breakdown. Infrastructures are used by people who are not experts in developing the infrastructure technologies. So they familiarize with the technological concepts driven by pure necessities of use. User experience breakdown situations (be it an actual technological breakdown or simply a perceived incongruence between the expected and the delivered service of an application) which force them to learn another part of the body of knowledge that is necessary to understand and orchestrate technology. In this perspective, tailoring is much more than a simple configuration of tools. It is an opportunity for use reflection and for learning about the (in-) abilities of today's technologies. The necessities of being competent regarding the technologies which form the infrastructure are forged into our structures of professionalization (addressed by targeted education efforts like University courses for becoming an IT expert or a software engineer). By contrary, infrastructural problems are either covered by more or less appropriate divisions of labor in an organization (system administrators, IT departments, etc.) or left to the initiative of individual users. We believe that software is a completely new type of infrastructural matter (compared to older infrastructures like power lines, railroad systems or water pipes) since it can offer flexibility to support the activities of 'infrastructuring'. We should aim at allowing every user to dig into technological matters exactly to the depth that looks appropriate to them. We call that 'to provide a gentle slope of complexity' for tailoring activities. We believe that with component-based tailorability, we are able to indicate one possible way of achieving this. Now we want to address three crucial aspects.

5.1 Exploiting Congruencies

Star and Bowker (2002) explicitly referred to the historic dimensions of infrastructures, and describe how new infrastructures emerge from older ones, and make use of the concepts found there. At the individual level, an awareness of historical aspects like earlier experiences and expertise are gained. We described that in the form of 'congruencies' that the architecture and the visualization of a component-based system should maintain. Our choice of using component-based approach proved to be successful since we were able to achieve these congruencies to a certain level. The Component-Metaphor-Congruency addressed the issue that the use of the component metaphor in programming is related to the notion of an, almost haptic, experience in building complex systems from simple building blocks. To achieve this congruency, it was necessary to combine a strong encapsulation of all concepts which describe a component with a clear indication of how components fit and work together.

Related to the Architecture-Interface-Congruency is a decision which underlied all our efforts: To provide a 'truthful' representation of the architecture at the interface, we need clear architectural concepts that are easy to represent at the user interface (e.g. the port description we provided in the FLEXIBEANS concept). We accept that for a large number of individual problems it would be easier for users to have an interface which is specifically designed for certain tailoring tasks. However, we believe that such a design would prevent users from building a deeper knowledge of the true complexity of the system and it may hinder the understanding of future breakdown situations. We believe that for bridging the gap between component-based structures and the intuitive understanding of users in an application field, an additional metaphor framework on top of the component network could provide benefits. But even in this case it has to be possible to trace issues into deeper levels of the application' software architecture. Our experiences also confirm that it is not necessarily a good idea to maintain a strong separation between the computational and the interface level of applications (c.f. Kuutti and Bannon 1993). The Use-Tailoring-Congruency helps users to enter the world of tailoring and to understand what a certain tailoring option is about.

The three congruencies address the end-user as a 'casual programmer', and allow him to interact with the infrastructure in a more powerful way.

5.2 Employing a Holistic Approach

We also like to emphasize here, that the completeness of the research (in terms of addressing the architectural level as well as the interface level and the social level) was not necessarily our intention in the beginning, but it proved to be essential during our evaluations. We simply cannot escape the embeddedness of users in their work context and their focus on their actual tasks (where tailoring is more a side task), and therefore we believe that scenarios which focus on providing help in just one specific situation do not cover the complexity of the problem. Apart from an incident-oriented support, we also have to address a strategic level of familiarizing with technologies. It needs to be supported at the individual as well as at the social level.

5.3 Supporting Appropriation Work

So far, we maintain an understanding of tailoring as a task to reconfigure or redesign software tools,. Henderson and Kyng (1991) went already beyond such an understanding by employing a perspective, where tailoring is not only the design of configurations, but the design of tools' 'usages'.. There is usually a certain set of tasks, roles, use scenarios, and conventions associated with the technological reconfiguration of tools. Under the term of 'appropriation work', we address the activities around the sense-making of technologies in an application field (Pipek 2005). Beyond learning about the functionality of an application, it deals with social activities related to the design of an appropriate usage of the application. An approach, which is aware of the infrastructural character of software artifacts has to provide support for these activities. In the FreEvolve approach, we address that issue by combining component repositories with exploration environments. Pipek (2001) suggested the use of 'use discourse environments' to provide a platform for these negotiations. One can imagine support along the following lines:

- *Articulation Support:* Support of technology-related articulations (real and online)

- *Historicity Support:* Visualise appropriation as a process of emerging technologies and usages, e.g. by documenting earlier configuration decisions, providing retrievable storage of configuration and usage descriptions.

- *Decision Support:* If an agreement is required in a collaborative appropriation activity, providing voting, polling, etc.

- *Demonstration Support:* Support showing usages from one user (group) to another user (group), providing necessary communication channels.

- *Observation Support:* Support the visualisation of (accumulated) information on the use of tools and functions in an organisational context.

- *Simulation Support:* Show effects of possible usage in an exemplified or actual organisational setting (only makes sense if the necessary computational basis can be established).

- *Exploration Support:* Combination of simulation with extended support for technology configurations and test bed manipulations, individual vs. collaborative exploration modes.

- *Explanation Support:* Explain reasons for application behaviour, fully automated support vs. user-user- or user-expert-communication.

- *Delegation Support:* Support delegation patterns within configuration activities; providing remote configuration facilities.

- *(Re-) Design support*: feedback to designers on the appropriation processes

The integration of these ideas is an important part of continuing the research described here.

5.4 Challenges on the technological level

We want to close our discussion by addressing open ends in technology-oriented research on tailorability. While our results stem from long-term research activities covering the design and implementation of technological innovations as well as their evaluation in laboratory-settings and field studies, many issues still remain open. We do not know yet which type of tailorable applications is best suited for component-based approaches compared to other software technical paradigms such as rule-based or agent-based ones. From a technological perspective, our experiences indicate

that applications or parts of them, with a control flow that can be presented in a rather linear order, are well suited for component-based tailorability.

Methods that allow finding appropriate modularizations of an application into atomic components are another issue for further investigation (cf. Stiemerling et al 1997; Stiemerling, 2000; Stevens and Wulf, 2002). The modularization must also be meaningful to users (cf. Stevens et al. 2006). For different classes of applications, we need to find meaningful metaphors to communicate the meaning of individual components. The CoCoWare platform (Slagter et al., 2001) allows users to compose their own component-based groupware applications. In this platform, each component is in itself a small application, e.g. a session control or a conference manager component. While their work is, from a software-technological perspective, closely related to us, the main difference conserns the granularity of the components. While in CoCoWare components represent whole applications, FLEXIBEANS are conceptualized to be more fine-grained. On the one hand this allows for more flexibility, on the other hand tailoring becomes more complex.

Extending our approach, one can also imagine introducing additional levels of tailoring complexity by gray or even glass-boxing atomic components. Thus, selected aspects of the code or even the whole code of an atomic component could become modifiable by certain users. With regard to the user interface for tailoring, one has to investigate whether a single interaction paradigm is sufficient for component-based tailorability or whether the interaction paradigm of the tailorable application needs to be taken into account for the design of the interface of the tailoring environment (cf. Nardi 1993).

Regarding the support of cooperative tailoring activities, we need to think of additional features which support users in selecting appropriate atomic or abstract components out of a larger set of components. Ye (2001) has developed a recommender system which supports software developers to share and reuse source codes via a repository. We believe that similar functionalities will be valuable for collaborative tailoring activities. Moreover, we will be able to learn from the open source movement and the discussion on social capital to design shared repositories in an appropriate manner (cf. Huysman and Wulf, 2004; Fischer et al., 2004).

The final point we want to make here is concerning other software development paradigms. One important issue is the development of appropriate tailoring platforms for peer-to-peer architectures and mobile systems (cf. Alda and Cremers, 2004). Our approach may fall short there since we use a client-server architecture. However, peer-to-peer architectures face additional challenges regarding synchronization and availability.

Nevertheless, with the emergence of software-oriented architectures (SOA), new promises and challenges occur. Our experiences showed how difficult it is in providing an infrastructure that addresses the two main goals of end-user orientation and runtime tailorability within service networks. SOA promise, on the one hand, a higher level of flexibility since the concept is not related to an operating system or programming language (like component-based systems are); on the other hand the current practice of SOA addresses only the use by programmers, but not end-users. This can be interpreted as a slightly worse starting situation compared to our work with components. Otherwise, services resemble aspects of the component-based architectures: Services are independent of each other, stateless, self-describing and provide standardized interfaces (Bieberstein et al. 2006, Brown et al. 2002). But services also provide a process perspective on applications, not just an object-/component perspective (Jones und Morris 2005). However, the promise of a new level of flexibility is already being undermined by an inflationary use of the term ‘service-orientation’ (Doernhoefer 2006). The standardization efforts are counteracted in practice by the emergence of different service worlds (e.g. SOAP web services vs. OSGi architecture). Nevertheless, it will be interesting to see whether and how the concepts developped for component-based systems translate into the SOA world.

6. Conclusion

We have presented our work on how the concept of component-based tailorability can be made intelligible and manageable for end-users. Due to the specific requirements of users whose main interest is not in software development, the requirements for the design of the user interface are distinct from typical developer-oriented IDEs in software engineering. We worked out a component-based approach by evaluating our experiences with the FREEVOLVE platform. In addition, we developed a number of prototypes which covered important side issues. These issues covered constraint-based integrity checks so as to help users to detect tailoring errors as well as the provision of exploration environments, so that they can familiarize with the dynamic aspects of tailored component networks.

We suggested a holistic approach to component-based tailorability by addressing the architectural level as well as the interface and the social/collaborative level. We described how we used different ‘congruencies’ (Component-Metaphor-Congruency, Architecture-Interface-Congruency and Tailoring-Use-Congruency) as guidelines to provide a comprehensive tailoring environment complemented with the interface concept of direct activation. We discussed the

role of these congruencies in the development of three different tailoring interfaces. Furthermore, we addressed the social level by providing shared repositories of tailored components, and connecting them to exploration environments. Finally, we related our findings to the issue of developing technology in an 'infrastructure-aware' way. It is necessary to consider technology as an infrastructure for users, something which remains invisible until breakdown, but also something extremely urgent to know about in case of breakdown. Such a perspective promotes several notions for the support of 'appropriation work' which users perform typically when making sense of technologies. Perceiving groupware as an infrastructure also means that the levels of qualification, interest, and dedication will be different among users involved in tailoring an application and that they will vary over time. The infrastructure issue connects to earlier discussions about the need of an emerging tailoring culture within the field of application (see Carter and Henderson 1990).

Along these lines of thought, we need to gather new experiences on how to connect tailoring activities with processes of organizational development and change (see Wulf and Jarke 2004). In order to improve flexibility and efficiency of business processes, the exploitation of tailorability needs to be integrated into the ongoing processes of organizational change. Therefore, we have developed the framework of Integrated Organization and Technology Development which connects tailorability with planned processes of organizational and technological development (cf. Wulf and Rohde 1995). However, we need to investigate more on emergent change processes and the role of tailorability in groupware appropriation (cf. Orlikowski and Hofman 1997; Andriessen, Hettinga; Wulf, 2003).

7. References

Ackermann, D.; Ulich, E.: The chances of individualization in human-computer interaction and its consequences. In: Frese, M.; Ulich, E.; Dzida, W. (eds), Psychological issues of human computer interaction in the work place, North Holland, Amsterdam 1987, pp. 131-146

Alda, S. and Cremers, A.B.: Towards Composition Management for Peer-to-Peer Architectures, in: *Proceedings of the Workshop Software Composition (SC 2004)*, affiliated to the 7th European Joint Conference on Theory and Practice of Software (ETAPS 2004). Barcelona, Spain. April 2004

Ambler, A.; Leopold, J.. Public Programming in a Web World. Visual Languages. Nova Scotia, Canada, 1998.

Andriessen, J. H. E.; Hettinga, M.; Wulf, V. (eds): Special Issue on Evolving Use of Groupware, in: Computer Supported Cooperative Work: The Journal of Collaborative Computing (JCSCW), Vol. 12, No. 4, 2003

Bellissard, L.; Atallah, S. B.; Boyer, F.; Riveill, M.: Distributed Application Configuration, in: Proceedings of the 16th International Conference on Distributed Computing Systems, Hongkonk, IEEE-Press, 1996, pp. 579-585

Bentley, R. und Dourish, P.: Medium versus Mechanism. Supporting Collaboration Through Customisation, in: Marmolin, H., Sundblad, Y., and Schmidt, K. (eds.), Proceedings of the Fourth European Conference on Computer Supported Cooperative Work - ECSCW '95, Kluwer, 1995, pp. 133-148.

Bentley, R., W. Appelt, U. Busbach, E. Hinrichs, D. Kerr, K. Sikkel, J. Trevor, and G. Woetzel: Basic Support for Cooperative Work on the World Wide Web, in: International Journal of Human Computer Studies 46, 1997, pp. 827-846.

Bieberstein, Norbert; Bose, Sanjay; Fiammante, Marc; Jones, Keith; Shah, Rawn (2006): Service-Oriented Ar-chitecture Compass. Pearson, 2006.

Blythe, M.A., Overbeeke, K., and Monk, A.F.: *Funology*. Kluwer Academic Publishers, Dordrecht, NL, 2004, 320 p.

Brown, A.W., S. Johnston, and K. Kelly (2002). Large-scale, using service-oriented architecture and component-based development to build web service applications. Rational Software White Paper TP032, 2002.

Carroll,J. M.: Five Gambits for the Advisory Interfaces Dilemma, in: Frese, M.; Ulich, E.; Dzida, W. (eds): Psychological Issues of Human Computer Interaction in the Work Place, Amsterdam 19987, pp. 257 – 274.

Carroll, J. M.; Carrithers, C.: Training Wheels in a User Interface, in: Communications of the ACM, Vol. 27, No. 8, 1984, pp. 800 – 806.

Carter, K.; Henderson, A.: Tailoring Culture, in: Hellman, R.; Ruohonen, M.; Sorgard, P. (eds): Proceedings of the 13th IRIS, Reports on Computer Science and Mathematics, No. 107, Abo Akademi University 1990, pp. 103 - 116

Cortes, M.: A Coordination Language For Building Collaborative Applications, Journal of Computer Supported Cooperative Work, 1999.

Dittrich, Y., Dourish, P., Mørch, A., Pipek, V., Stevens, G., and Törpel, B. "Special Issue on Supporting Appropriation Work," International Reports on Socio-Informatics (IRSI) (2:2) 2005, p 84. http://irsi.iisi.de/

Doernhoefer, Mark (2006): Surfing the net for software engineering notes. ACM SIGSOFT Software Engineer-ing Notes, Volume 30, Issue 6, S. 5-13 November 2005. ISSN:0163-5948

Dourish, P.: Open implementation and flexibility in CSCW toolkits, Ph.D. Thesis. London: University College, 1996.

Engelskirchen, T.: Exploration anpassbarer Groupware, Master Thesis, University of Bonn, 2000

Fischer, G.; Girgensohn, A.: End-User Modifiability in Design Environments. In: Proceedings of the Conference on Computer Human Interaction (CHI ´90), April 1 - 5, 1990, Seattle, Washington, ACM-Press, New York 1990, pp. 183-191

Fischer, G., Lemke, A.C., and Rathke, C. "From design to redesign," International Conference on Software Engineering, Monterey, California, United States, 1987, pp. 369-376.

Fischer, G.; Scharff, E.; Ye, Y.: Fostering Social Creativity by Increasing Social Capital, in: Huysman, M.; Wulf, V. (eds): Social Capital and Information Technology, MIT-Press, Cambridge, MA 2004 in press

Frese, M., Irmer, C. und Prümper, J.: Das Konzept Fehlermanagement: Eine Strategie des Umgangs mit Handlungsfehlern in der Mensch-Computer Interaktion (‚The concept of „Fault Management“: A strategy of dealing with activity faults in Human-Computer-Interactions'), in: Software für die Arbeit von morgen (‚Software for tomorrow's work'), C. Skarpelis, Ed. Berlin: Springer Verlag, pp. 241-252, 1991.

Golombek, B.: Implementierung und Evaluation der Konzepte „Explorative Ausführbarkeit“ und „Direkte Aktivierbarkeit“ für anpassbare Groupware (‚Implementation and Evaluation of the concepts of „Explorative Execution“ and „Direct Activation“ for tailorable Groupware'), Master Thesis, University of Bonn, 2000

Grudin, J.: Evaluating Opportunities for Design Capture, in: Moran, J. P.; Carroll, J. M. (eds.): Design Rationale: Concepts, Techniques and Use, LEA, Hillsdale 1996

Hallenberger, M.: Programmierung einer interaktiven 3D-Schnittstelle am Beispiel einer Anpassungsschnittstelle für komponentenbasierte Anpassbarkeit (‚Implementation of an interactive 3D-interface in the example of a tailoring interface for component-based tailorability'), University of Bonn, Master Thesis, 2000

Henderson, A. and Kyng, M., “There's No Place Like Home: Continuing Design in Use,” in Design At Work - Cooperative Design of Computer Artefacts, J. Greenbaum and M. Kyng, Eds. Hillsdale, New Jersey: Lawrence Erlbaum Associates, Publishers, 1991, pp. 219-240.

Hinken, R.: Verteilte Anpassbarkeit für Groupware - Eine Laufzeit und Anpassungsplattform (‚Distributed Tailoring of Groupware – A run-time and tailoring environment'), University of Bonn, Master Thesis, 1999.

Howes, A.; Paynes, S. J.: Supporting exploratory learning, in: Proceedings of INTERACT'90, North-Holland, Amsterdam 1990, pp. 881 – 885.

Huysman, M.; Wulf, V. (eds): Social Capital and Information Technology, MIT Press, Cambridge MA 2004

IBM: Visual Age for Java, Version 1.0, 1998.

ISO 9241: Ergonomic requirements for office work with visual display terminals (VDTs) Part 10: Dialogue Principles

Johansen, R.: Current User Approaches to Groupware, in: Johansen, R. (ed): Groupware, Freepress, New York 1988, pp. 12-44

Jones, Steve; Morris, Mike (2005): A methodology for Service Architectures. OASIS Draft, http://www.oasis-open.org/committees/download.php/15071/A%20methodology%20for%20Service%20Architectures%201%202%204%20-%20OASIS%20Contribution.pdf, referenced in January 2006

Kahler, H.: Supporting Collaborative Tailoring, Ph.D.-Thesis, Roskilde University, Denmark, Roskilde, 2001.

Kahler, H.: More than WORDSs: Collaborative Tailoring of a word processor, in: Journal on Universal Computer Science (j.ucs), Vol. 7, No. 9, 2001a, pp. 826-847.

Krings, M.: Erkennung semantischer Fehler in komponentenbasierten Architekturen (‚The recognition of semantic errors in component-based architectures'), University of Bonn, Master Thesis, 2002.

Krüger, M.: Semantische Integritätsprüfung für die Anpassung von Komponenten-Kompositionen (‚Semantic Integrity Checking for tailoring component aggregates'), University of Bonn, Master Thesis, 2003.

Kuutti, K., and Bannon, L. "Searching for Unity among Diversity: Exploring the Interface Concept," interchi'93, ACM Press, 1993, pp. 263-268.

Mackay, W.E.: Users and customizable Software: A Co-Adaptive Phenomenon, PhD Thesis, MIT, Boston (MA), 1990.

MacLean, A., Carter, K., Lövstrand, L., und Moran, T.: User-tailorable Systems: Pressing the Issue with Buttons, in: Proceedings of the Conference on Computer Human Interaction (CHI '90), April 1-5, Seattle (Washington), ACM-Press, New York, 1990, pp. 175-182

Magee, J., Dulay, N., Eisenbach, S., und Kramer, J.: Specifying Distributed Software Architectures, in: Proceedings of 5th European Software Engineering Conference, Barcelona, 1995.

Malone, T. W., Lai, K.-Y., and Fry, C.: Experiments with Oval: A Radically Tailorable Tool for Cooperative Work, in: Proceedings of CSCW., Toronto, Canada, ACM Press, 1992, pp. 289-297.

McIlroy, D.: Mass-produced software components, in: Proceedings of Conference on Software Engineering, Garmisch-Partenkirchen (D), North Atlantic Treaty Organization (NATO), 1968.

Microsoft: Visual Basic, Version 4.0, 1996.

Mørch, A. I.: Method and Tools for Tailoring of Object-oriented Applications: An Evolving Artifacts Approach, PhD-Thesis, University of Oslo, Department of Computer Science, Research Report 241, Oslo, 1997.

Mørch, A.I. and Mehandjiev, N.D.: Tailoring as Collaboration: The Mediating Role of Multiple Representations and Application Units. Computer Supported Cooperative Work 9(1), 2000, pp. 75-100.

Mørch, A.I; Stevens, G.; Won, M.; Klann, M.; Dittrich, Y.; Wulf, V.: Component-based Technologies for End User Development, in: Communications of the ACM, Vol. 47, No. 9, 2004, pp. 59 - 62

Myers, B. A.: Taxonomies of Visual Programming and Program Visualization, Journal of Visual Languages and Computing, vol. 1, 1990, pp. 97-123.

Nardi, B. A.: A Small Matter of Programming - Perspectives on end-user computing, MIT-Press, Cambridge et al., 1993.

Nielsen, J.: Usability Engineering Academic Press, Boston, MA, USA, 1993.

Oberquelle, H.: Anpassbarkeit von Groupware als Basis für die dynamische Gestaltung von computergestützter Gruppenarbeit (‚Tailorability of Groupware as a basis for a dynamic design of Computer-Supported Cooperative Work'). In: Konradt, U.; Drisis, L. (eds.): Benutzeroberflächen in der teilautonomen Arbeit (‚User Interfaces in semi-autonomous work'), Köln 1993, pp. 37-54.

Oberquelle, H.: Situationsbedingte und benutzerorientierte Anpassbarkeit von Groupware (‚Situational and user-oriented tailorability of groupware'), in: Hartmann, A.; Herrmann, Th.; Rohde, M.; Wulf, V. (eds.), Menschengerechte Groupware, Stuttgart, 1994, pp. 31-50.

Oppermann, R.; Simm, H.: Adaptability: User-Initiated Individualization, In: Oppermann, R. (ed.): Adaptive User Support – Ergonomic Design of Manually and Automatically Adaptable Software, LEA, Hillsdale, NJ, 1994.

Orlikowski, W. J.; Hofman, J. D.: "An Improvisational Model for Change Management: The Case of Groupware Technologies"; in: Sloan Management Review (Winter 1997); 1997, pp. 11-21.

Page, S.; Johnsgard, T.; Albert, U.; Allen, C.: User Customization of a Word Processor. In: Proceedings of CHI '96, April 13.-18 1996, pp. 340-346.

Pane, J.F.; Myers, B.A.; Ratanamahatana, C.A.: Studying the language and structure in non-programmers' solutions to programming problems, International Journal of Human-Computer Studies, v.54 n.2, p.237-264, Feb. 2001

Pane, J. F.; Myers, B. A.: More Natural Programming Languages and Environments, in: Lieberman, H.; Paternó, F.; Wulf, V. (eds): End User Development, Kluwer Dordrecht 2006, in press

Paul, H.: Exploratives Agieren („Explorative Agency"), Peter Lang, Frankfurt/M 1994.

Pipek, V. "From Tailoring to Appropriation Support: Negotiating Groupware Usage," in: Faculty of Science, Department of Information Processing Science (ACTA UNIVERSITATIS OULUENSIS A 430), University of Oulu, Oulu, Finland, 2005, p. 246.

Pipek, V.; Kahler, H.: Supporting Collaborative Tailoring, in: Lieberman, H.; Paternó, F.; Wulf, V. (eds): End User Development, Kluwer Dordrecht 2006, in press

Ravichandran, T., and Rothenberger, M.A. "Software reuse strategies and component markets," Communications of the ACM (46:8) 2003, pp 109-114.

Repenning, A.; Ioannidou, A.; Zola, J.: "AgentSheets: End-User Programmable Simulations"; in: Journal of Artificial Societies and Social Simulation, 3(3), 2000.

Robertson, T. "Shoppers and Tailors: Participative Practices in Small Australian Design Companies," Computer Supported Cooperative Work (CSCW) (7:3-4) 1998, p 205 221.

Schmidt, K.: Riding a Tiger or Computer Supported Cooperative Work. In: Bannon, L., Robinson, M. and Schmidt, K. (eds): Proceedings ECSCW '91, Kluwer, Dordrecht, 1991, pp. 1-16.

Shu, N.C.: Visual programming. Van Nostrand Reinhold Co., New York, NY, USA, 1988

Silberschatz, A., Korth, H., and Sudarshan, S.: Database System Concepts, Osborne McGraw-Hill, 2001.

Star, S.L., and Bowker, G.C. "How to infrastructure," in: Handbook of New Media - Social Shaping and Consequences of ICTs, L.A. Lievrouw and S. Livingstone (eds.), SAGE Pub., London, UK, 2002, p. 151 162.

Stiemerling, O.: CAT: Component Architecture for Tailorability, Working Paper, Department of Computer Science, University of Bonn, 1997.

Stiemerling, O.: FLEXIBEANS Specification V 2.0, Working Paper, Department of Computer Science, University of Bonn. 1998

Stiemerling, O.: Component-based Tailorability, Ph.D. Thesis, Department of Computer Science, University of Bonn, Bonn, 2000. Available at: http://www.freevolve.de/Dissertation.pdf

Stiemerling, O.; Cremers, A. B.: Tailorable Component Architectures for CSCW-Systems, in: Tyrell, A. M. (ed.): Proceedings of the 6th Euromicro Workshop on Parallel and Distributed Processing, IEEE-Press 1998, pp. 302-308.

Stiemerling, O.; Hallenberger, M.; Cremers A. B.: 3D Interface for the Administration of Component-Bases, Distributed Systems, in: Fifth International Symposium on Autonomous Decentralized Systems, March 26 - 28, 2001, Dallas, Texas, IEEE Press 2001, pp. 119-126..

Stiemerling, O.; Hinken, R.; Cremers, A. B.: The EVOLVE Tailoring Platform: Supporting the Evolution of Component-based Groupware, in: Proceedings of EDOC'99, IEEE Press, Mannheim, Sept. 27.-30., 1999, pp. 106-115.

Stiemerling, O.; Hinken, R.; Cremers, A. B.: Distributed Component-based Tailorability of CSCW Applications, in: Proceedings of ISDS'99, Tokio, Japan, IEEE-Press, 1999a, pp. 345-352.

Stiemerling, O.; Kahler, H. and Wulf, V.: How to Make Software Softer - Designing Tailorable Applications. In Proceedings of 2nd Conference on the Design of Interactive Systems, Amsterdam (NL), ACM Press, 1997, pp. 365-376.

Stiemerling, O.; Won, M.; Wulf, V.: Zugriffskontrolle in Groupware - Ein nutzerorientierter Ansatz (‚Access control in Groupware – a user-centered approach'), in: WIRTSCHAFTSINFORMATIK, 42. Jg., Nr. 4, 2000, pp. 318-328.

Steven, G.: Komponentenbasierte Anpassbarkeit - FlexiBeans zur Realisierung einer erweiterten Zugriffskontrolle (‚Component-based Tailorability – Using Flexibeans to implement an extended access control system'), University of Bonn, Master Thesis, 2002.

Stevens, G.; Wulf, V.: A New Dimension in Access Control: Studying Maintenance Engineering across Organizational Boundaries, in: Proceedings of ACM Conference on Computer Supported Cooperative Work (CSCW 2002), ACM-Press, New York, 2002, pp. 196 – 205

Stevens, G.; Quaisser, G; Klann, M.: Modulizing Software for Tailorability - An Industrial Case Study, in: Lieberman, H.; Paternó, F.; Wulf, V. (eds), End User Development, Kluwer, Dordrecht 2006, in press

Szyperski, C.: Component Software: Beyond Object Oriented Programming, 2nd Edition, Addison Wesley, London, 2002.

Slagter, R., Biemans, M, & Ter Hofte, G.H. Evolution in use of groupware: Facilitating tailoring to the extreme, in: M. Borges, J. Haake and U. Hoppe (eds.), Proceedings of the 7th International Workshop on Groupware (CRIWG 2001), 6-8 September 2001, Darmstadt, Germany, 2001.

Teege, G.: Users as Composers: Parts and Features as a Basis for Tailorability in CSCW Systems, CSCW, Kluwer Academic Publishers, 2000, pp. 101-122.

van der Aalst, W., D., H. und Verbeek, H. M. W.: A Petri-Net-based Tool to analyze workflows, in: Proceedings of Petri Nets in System Engineering (PNSE'97), Hamburg, Universität Hamburg, 1997, pp. 78-90.

Wang, W., and Haake, J.M. "Tailoring Groupware: The Cooperative Hypermedia Approach," International Journal of Computer-Supported Cooperative Work (9:1) 2000.

Won, M.: Komponentenbasierte Anpassbarkeit - Anwendung auf ein Suchtool für Groupware (‚Component-based Tailorability and ist application on a groupware searching tool'), University of Bonn, Master Thesis, 1998.

Won, M.: Supporting End-User Development of Component-Based Software by Checking Semantic Integrity, in: ASERC Workshop on Software Testing, 19.2.2003, Banff, Canada, 2003.

Won, M.: Checking integrity of component-based architectures, CSCW 2000 Philadelphia USA, Workshop on Component-Based Groupware, 2000.

Wulf, V.: Anpaßbarkeit im Prozeß evolutionärer Systementwicklung (‚Tailorability within the process of evolutionary system development'); in: GMD-Spiegel, Vol. 24, 3/94, pp. 41 – 46, 1994

Wulf, V.: "Let's see your Search-Tool!" - Collaborative use of Tailored Artifacts in Groupware, in: Proceedings of GROUP '99, ACM-Press, New York, 1999, pp. 50-60.

Wulf, V.: Exploration Environments: Supporting Users to Learn Groupware Functions, in: Interacting with Computers, Vol. 13, No. 2, 2000, pp. 265-299.

Wulf, V.: Zur anpassbaren Gestaltung von Groupware: Anforderungen, Konzepte, Implementierungen und Evaluationen (‚On the design of tailorable Groupware: Requirements, Concepts, Implementations and Evaluations'), GMD Research Series, Nr. 10/2001, St. Augustin, 2001 (zugleich: Habilitation Universität Hamburg 2000).

Wulf, V. und Golombek, B.: Direct Activation: A Concept to Encourage Tailoring Activities, Behavior and Information Technology, Vol. 20, No. 4, 2001, pp. 249-263.

Wulf, V.; Golombek, B. (2001a): Exploration Environments – Concept and Empirical Evaluation, in: Proceedings of GROUP 2001, ACM-Press, New York, 2001, pp. 107-116.

Wulf, V.; Jarke, M.: The Economics of End-user Development, in: Communications of the ACM, Vol. 47, No. 9, 2004, pp. 41 - 42

Wulf, V.; Rohde, M.: Towards an Integrated Organization and Technology Development; in: Proceedings of the Symposium on Designing Interactive Systems, 23. - 25.8.1995, Ann Arbor (Michigan), ACM-Press, New York 1995, pp. 55 – 64.

Wulf, V.; Stiemerling, O.; Pfeifer, A.: Tailoring Groupware for Different Scopes of Validity, in: Behaviour & Information Technology, Vol. 18, No. 3, 1999, pp. 199 - 212

Yang, Y.: Current Approaches and New Guidelines for Undo-Support Design, in: Proceedings of INTERACT'90, North-Holland, Amsterdam 1990, pp. 543 – 548.

Ye, Y.: *Supporting Component-Based Software Development with Active Component Repository Systems.* Ph.D. Dissertation, Department of Computer Science, University of Colorado at Boulder, Boulder 2001.