AI & Society (2000), 14, pp. 120-141

The EVOLVE Project: Component-Based Tailorability for CSCW Applications

Oliver Stiemerling and Armin B. Cremers
Department of Computer Science III
University of Bonn
Römerstraße 164
53117 Bonn
Germany
{os | abc}@informatik.uni-bonn.de

Abstract

Tailorability is generally regarded as a key property of groupware systems due to the dynamics and differentiation of cooperative work. This article investigates the use of software components as a generic architectural concept for designing tailorable groupware applications. First, the issues raised by this approach are discussed in the context of an exploratory experiment during which component-based tailorability was applied to a real tailoring problem in the POLITeam project. The experiment's results led us to concentrate on questions concerning the support of distributed CSCW applications. As a consequence, we have developed the EVOLVE platform whose design concepts are described. Furthermore, a concrete example for the application of the approach to the design of a tailorable distributed coordination tool is given. We discuss related work, summarize the current state of the component-based tailorability approach and propose venues of further research.

1. Introduction

While only a few years ago computers were mainly used by highly educated information technology specialists, today almost everybody has to deal with them. As a consequence, *tailorability* to diverse and changing requirements is becoming an increasingly important quality of software systems. End users are tired of organizing their work around the antics and technological idiosyncrasies of software designed for large market segments or based on incomplete or fallacious knowledge about the application domain. They require software matching their particular work situation, preferences, and style. We call a system *tailorable*, if it can be changed by end users (or others) to achieve this match (some authors – e.g. (Oberquelle 1994) – differentiate between *tailorability* and *configurability*, with the latter referring to changes to the system after development but before its first use. This distinction is not relevant in the context of this paper). Tailorability is certainly no panacea for creating perfectly fitting software systems. However, it gives end-users (or more generally: all non-developers involved) a stronger role in the system lifecycle by permitting some decisions which usually have to be made during the design-phase to be transferred into the use-phase.

In CSCW (Computer Supported Cooperative Work) systems tailorability is especially important, because the group dimension adds a number of requirement-dependencies on factors which are less or even not at all relevant for single user application design. Regard, for instance, country specific laws or even company specific agreements governing the access to sensitive (e.g. personnel related) documents. If a company-wide groupware systems cannot be appropriately adapted to restrict access to these documents according to the applying laws and agreements, the success of the system's introduction is in serious doubt. Another example are notification (or awareness) services in groupware. They are supposed to make users aware of activities which are relevant for the coordination of cooperative work (e.g. a users is notified,

whenever a certain document - perhaps containing company rules - is edited). Since – depending on his or her role in the organization and the nature of the assigned tasks – the user's need to be aware of certain events can vary substantially, a non-tailorable awareness service could cause a flood of unwanted notifications or a scarcity of needed ones (or even both at the same time).

Additionally, the introduction and subsequent tailoring of groupware systems has a high potential for conflict (see e.g. Wulf 1995), which – apart from CSCW-“induced” technical issues like distribution and concurrency – is one of the challenges in the design and implementation of groupware tailorability. Regard, for instance, the example of tailoring a workflow system which has been in use for some time and thereby upsetting the established division of labor within one department, e.g. by assigning certain tasks differently. As soon as some members of the organization perceive themselves as losers in this development, the adaptation becomes a source of conflict and confrontation within the organization.

However, tailorability can also alleviate or even avoid conflicts in the context of groupware projects. If, for instance, a rigid system imposes a certain style of cooperation (e.g. highly structured workflows) across several – perhaps rather differentiated – groups within one organization, some groups may be forced to work in an unsuitable fashion. Conflicts are the obvious consequence. A tailorable system permitting the adaptation of the cooperation style for certain groups, i.e. for a certain scope of validity see (see Wulf et al. 1999), can alleviate such conflicts. Alternatively, Herrmann (1995) suggested in the context of workflow management systems to offer negotiation mechanisms within the groupware system itself which permit users affected by adaptations to reject, accept, modify or comment on them.

Because of the relevance of tailorability for the design of CSCW systems, a lot of current research in tailorability is motivated by and conducted in the context of complex groupware projects, which is also the background of the work presented here (the PoliTeam project, see Klöckner et al. 1995). Abstracting from concrete projects and applications, design for tailorability raises three major questions (Stiemerling et al. 1997):

How can we determine the necessary points and degree of tailorability? This question concerns the design process and is driven by three factors (Trigg 1992): fluidity, diversity, and uncertainty of requirements. Especially the first and third factor involve the future use of the system and thus we will never have a complete methodological solution. However, current requirements capturing and design techniques can be extended to take anticipated future changes into account (see e.g. the change case methodology by Ecklund et al. 1996).

How can we control and implement adaptations after initial system development? Assuming we have a tailorable system, this questions concerns the way, how the need for an adaptation is discovered, how an appropriate adaptation is developed, how the decision for or against the adaptation is made, and how it is finally implemented. (Kühme et al. 1992) suggest these four factors as basis for a taxonomy of adaptation control mechanisms. They distinguish, which step is controlled by the user or by the system. We call a system *adaptive*, if the control resides mostly with the system, and *user tailorable* otherwise. In the latter case, user interfaces for tailorability become a major issue. Furthermore, tailoring software systems has been identified as a cooperative activity, involving persons of different technical skill levels and domain knowledge (see e.g. Mackay 1990, or Oberquelle 1994). These group aspects raise new questions concerning tailoring rights and exchange of adaptations between users and groups of users.

How can we support tailorability in the software architecture? Regardless whether adaptations are controlled by user or system, they have to be supported on the software technical level. An at least partial specification of the system has to be available in effectively manipulatable form (Stiemerling and Cremers 1998). Some systems (e.g. LINKWORKS which

is used in the POLITeam project as groupware platform) are partially implemented in application specific high-level languages. The implementation can be made accessible for system administrators or even end users in order to permit the tailoring of the system. Traditional initialization files can be seen in this context as very simple (parameter) languages.

When designing a specific application, these three questions – especially in user centered design processes – are usually addressed in the order given above. Our goal, however, is to develop a generic approach to tailoring which can be applied to a variety of software systems. From this perspective, the first two questions rely on the choices made to address the third: The type and power of a possible tailoring user interface is constrained by the underlying architecture, as is the design of adaptive control mechanisms. The architectural level also determines which and how flexibility requirements can be accommodated. Therefore, it appears sensible to first address the question of appropriate architectural concepts and then design adaptation control mechanisms and development methodologies specific for the chosen architecture.

According to these thoughts, the EVOLVE project at the University of Bonn has developed an approach which provides generic (i.e. application independent) tailorability based on the concept of hierarchically structured component architectures. The second section of this paper describes the basic concepts of component-based tailorability and raises the questions addressed in the project. Section three describes the results of a first exploratory experiment applying component-based tailorability in the POLITeam project. The section also relates the different aspects of the approach to the current literature and identifies issues motivating the work presented in the following sections. In section four the architecture of the EVOLVE platform is described, together with an example for a distributed tailorable CSCW application. Section five presents a cross-section of CSCW research systems which explicitly address the issue of tailorability. Section six summarizes and gives an overview of future work.

2. Component-based Tailorability

Component-oriented programming (see Szyperski 1998) is motivated by the successes classical engineering disciplines like electronic or mechanical engineering have had with building complex artifacts from standardized components (e.g. transistors, resistors, cogs, or screws). Taking into account this motivation, a software component can be defined as "*a unit of composition with contractually specified interfaces and explicit context dependencies only. Components can be deployed independently and are subject to composition by third parties.*" (Szyperski and Pfister 1996, p. 130).

Components have to be distinguished from objects. The latter are concerned with structure and behavior (in the form of attributes and methods), the former with composability. The interface definition of an object only contains the services (methods) provided, while the object might additionally require services of other objects. These required services are not part of the explicit interface description, but are implicit in the implementation of the methods (in the form of calls to methods of other objects).Thus the "*explicit context dependencies only*" condition of the above definition is not met. Furthermore, components can consist of a large number of objects, and are thus potentially more abstract than objects, i.e. on a higher level of granularity.

In the context of our work, components are also different from processes, because they do not necessarily have to be active entities, but can become active or passive, depending on how processes or threads traverse the component system. In the field of distributed systems, this distinction is often blurred.

We say that a system built from components has a *component architecture*. On the component level such a system can be described using an appropriate *composition language* (see Nierstrasz and Meijler 1995). The possible form of the composition language depends on the underlying component model. Later on in this paper we describe CAT (Component Architecture for Tailorability) which is the composition language we are using in our work (for a specification of the language see Stiemerling 1997).

Advantages which component oriented programming hopes to provide include facilitation of reuse (especially by third parties), speed-up of development processes, reduction of development costs, and higher quality (standard components are less prone to exhibit errors if they have already been used and tested in prior projects).

Components have been successfully employed to support the design of graphical user interfaces. Application builders like MICROSOFT VISUAL BASIC (component model: VBX, OCX, see Microsoft 1996) or LOTUS BEANMACHINE (component model: JavaBeans, see JavaSoft 1997) provide often used generic visual design elements (e.g. buttons, text-boxes, combo-boxes), which are configured and composed during the design process to yield domain-specific applications.

The notion of components, however, has been applied to areas of software engineering other than GUI-design, as well. Formal component models and composition languages have been employed to describe the architecture of complex (even distributed) systems (see e.g. Wright described in Allen and Garlan 1994, or Darwin described in).

An advantage of viewing complex systems as compositions of components is that one can reason about the system on a high level (*high* in the sense of *conceptually close to the application domain*, or *further away from implementation details*). The level depends on the granularity of the components employed. The current discussion, however, exploits this advantage mostly in the design process. During the design process the component structure is often lost and with the final system the users are confronted with a monolithic application. In the EVOLVE project we are investigating the use of components for tailorability *after* initial development (also see Stiemerling and Cremers 1998, Stiemerling et al. 1999). Figure 1 shows the different uses of components for development and tailorability:

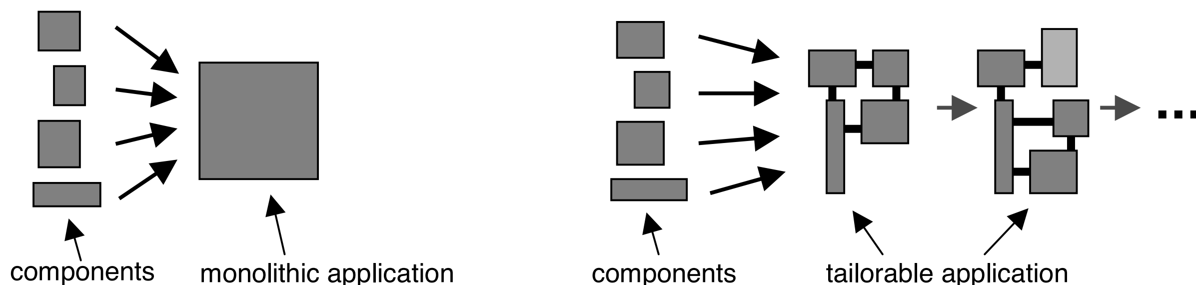


Figure 1: (left) components are used only during development, resulting in a monolithic application; (right) the component structure of the application is maintained after initial development, resulting in an application providing component-based tailorability.

Component architectures are attractive for tailoring, because they support a number of different tailoring interfaces, from simple parameterization (Henderson and Kyng 1991), over visual programming (Nardi 1993), to programming by modification of examples (Nardi 1993, Mørch 1997). If the architecture consists of multiple layers of nested components (hierarchical component architecture), tailoring operations at several different levels of abstraction and complexity are possible. Components on the higher levels of the hierarchy can

be closer in semantics to the application domain (e.g. the *bookkeeping component* of a business software package), while components further down can be more technically oriented (e.g. the *TCP/IP-protocol component*). Thus, a hierarchical component architecture can provide appropriate levels of tailoring for both a bookkeeper and a system administrator.

The collaborative aspect of tailoring is supported by the component approach, as well. Mackey noted in her dissertation (Mackay 1990) that users of computer systems often exchange configuration files, allowing others to share good adaptations which, for instance, make certain task more efficient or build a nice desktop environment. Components can easily be shared, because they are – by definition (compare Szyperski and Pfister 1996) – made for assembly by third parties.

Another reason why components are attractive for tailoring is that they seem to allow a clear separation of application-dependent concerns (encapsulated within the components of the framework) and tailoring functionality (provided by an application independent tailoring and run-time environment in form of generic compositional tailoring operations).

When going beyond component-oriented programming towards component-based tailorability, several issues arise anew or have to be rethought. In the EVOLVE project we investigate the following four questions which we believe to be central to component-based tailorability:

- What is an appropriate *component model* for component-based tailorability
- How do we build a *platform* allowing the flexible deployment of software components?
- How does one *design* applications suited for component-based tailorability?
- What are appropriate *user interfaces* for component-based tailoring?

We have formulated and explored these questions by applying the component-based tailoring approach to a real tailoring problem in the POLITeam project. This project is concerned with providing electronic support for the cooperative work of the distributed German government in Bonn and Berlin (see Klöckner et al. 1995). The example application was the search tool employed in the POLITeam system to search for documents (e.g. Microsoft Word documents) in the shared document base. In (Stiemerling and Cremers 1998, Stiemerling 1998) we have reported in detail on this work. The next section gives an overview of the results of this first application of the approach in relation to the four questions posed above.

3. The Application of Component-Based Tailorability in the POLITeam Project

The first subsection section briefly outlines the search tool tailoring problem and the technology used for the first version of the tool. The second subsection discusses the experiences made during the initial design process (finding the appropriate decomposition) and the workshop evaluation of the 2D user interface. The third subsection deals with the more technical issues of the component model and the tailoring platform.

The search-tool tailoring problem

In several workshops the POLITeam users articulated very diverse opinions about how the search-tool is supposed to present results and how the privacy of document-owners can be protected by limiting the possible search space of the tool. Another important issue was the handling of search results. Some users preferred a link (a basic POLITeam concept supporting the sharing of documents) to the found documents, while others wanted to create a copy of it. It became obvious that there was no one-best-way solution to the design problems, thus the search-tool constituted an excellent example application for exploring the component-based

tailoring approach. The range of different requirements is documented in (Kahler 1996) and served as basis for the experiment (see question one in the introduction).

We have implemented the tailorable search tool using the JAVABEANS component model see (see JavaSoft 1997). While this component model proved sufficient for the simple search tool example, we will describe how we have extended JAVABEANS in order to allow the support of more complex, distributed applications.

As platform, we have employed the JavaSoft BEANBOX which is supplied in source code together with the component model as an example development environment. By modifying an existing development environment we focussed on and learned about the technical differences between component-based system development and component-based tailorability (see figure 1). The modification also concerned the user interface which in the original BEANBOX was mainly aimed at developers and proved to be not very usable.

Empirical results

This subsection discusses these aspects of component-based tailorability which involved the end users of the POLITeam search tool, namely the participatory design process during which the decomposition of the search tool's functionality was determined and the user interface for tailoring the tool.

Designing applications suited for component-based tailorability: finding the appropriate decomposition

The hierarchical decomposition of a software system is application-dependent and should reflect the flexibility needed in the respective field of application. Thus, for any application the questions of which functionality should be encapsulated within a component, what granularity the most primitive components should have, and how these primitive components are hierarchically composed to finally yield the application have to be answered individually. Furthermore, we believe that from different perspectives there may be different sensible decompositions of the same application, e.g. that if we aim at decomposition for distribution we end up with a different result than what we get when we decompose for tailorability. Currently we view every perspective as an additional constraint on the space of possible decompositions which has to be taken into account.

Prior work on the POLITeam search tool (Kahler 1996) employed methods from the field of Participatory Design (PD, for an overview see e.g. Greenbaum and Kyng 1991, or Grønbæk et al. 1995). They were quite useful, since the active involvement of all prospective users of a groupware system helps in finding the nuances in usage differences and dynamics, which make up the need for tailorability. One user, for instance, who was responsible for preparing the weekly vote in the German state representative body (the Bundesrat), wanted the search tool to present the search results split in two groups, according to a certain document creation date. The reason for this requirements was the fact that laws are voted on twice in the German Bundesrat separated by a specific time interval. The user needed to quickly distinguish between documents relating to the first and the second vote.

Other users articulated the need to group the result by the found documents' location within the virtual desktop environment of the POLITeam system in order to be able to distinguish between documents from their own private desktop and other documents found in shared work spaces. Since access to documents in the system could be regulated by moving documents to shared spaces, this distinction proved important for efficient cooperation support. These (and other) requirements indicated the need for flexible grouping strategies for search results. As a consequence, we based the decomposition of the result presentation part

on a data flow metaphor (the data obviously being the search results). The lower part of figure 3 give the reader an impression of the application of this metaphor. The result stream is produced by the search engine (the component marked with "SQL"). It arrives at a switch-component which – in this case – splits up the result stream according to the first letter of the documents name. Other switch-components we implemented split up the result stream according to the documents' location or creation date. A switch component has two output ports which can be connected to other switch-components to further refine the grouping or to an output window which displays the search results (see figure 3). In a similar fashion, we developed the rest of the component framework, resulting in a "component language" which covered all diversities and dynamics of requirements discovered during the participatory design process. It is our experience that once the designers have gained some insight into the diversities and dynamics of a certain field of application, the decomposition of the application's functionality into components follows quit naturally.

While the PD methods worked quite well for requirement elicitation for one specific field of application, the literature reports on other approaches to the problem of finding appropriate decomposition for easy modifiability.

OVAL (Malone et al. 1995), for instance, represents an early approach to find *the one* decomposition of groupware functionality into components. The authors present a set of only four basic building blocks (Objects, Views, Agents, Links = OVAL) which they claim encapsulate functionality central to many groupware applications. They support this claim with an experiment during which a number of different existing CSCW applications were composed within OVAL. The result was that most applications could be build with "*only modest amounts of system level programming*" (p. 197) or with omitting of irrelevant functionality. While the authors interpret this result as success for their approach of "*radical tailorability*", it also shows clearly that any approach attempting to find "the one" decomposition for a vast number of different application classes is not suitable as basis for component-based tailorability.

In his dissertation Henri ter Hofte (1998) presents a very general approach which not so much aims at finding *the one* decomposition, but develops *structuring guidelines* for component-based groupware. These guidelines are much less restrictive than giving a concrete decomposition, since they are only supposed to support a groupware developer in finding a decomposition for a specific application. We do not discuss details of these guidelines here as they are rather deeply rooted in the special terminology of the models ter Hofte employs. However, note that his approach is based on the systematic structural analysis of a number of existing groupware applications and thus embraces a lot of what is know about groupware functionality today.

Users interfaces for component-based tailoring

The initial attractiveness of using components as basis for tailorability was fueled in part by the intuition that the process of tailoring component-based applications could be as easy (and perhaps as enjoyable) as playing with LEGO bricks. However, our experiences with software components and end users are rather ambivalent. In the context of the POLITeam search tool design we conducted a workshop with users from one of our fields of application in order to discuss the component-based design of the tool. We made two qualitative observations:

First, the end users had no problem whatsoever understanding the concept of software components. They readily grasped the LEGO metaphor and not only discussed different compositions of search tool components with us, but also demanded new components with additional or different functionality. Note that they did not simply talk about required functionality but *components* with a specific functionality. Thus, we are confident that in

principle software components can be employed explicitly as underlying concept for tailoring interfaces.

Second, the end users had problems with actually executing tailoring operations. Currently, we have implemented a 2D visual programming style interface which is depicted in figure 3:

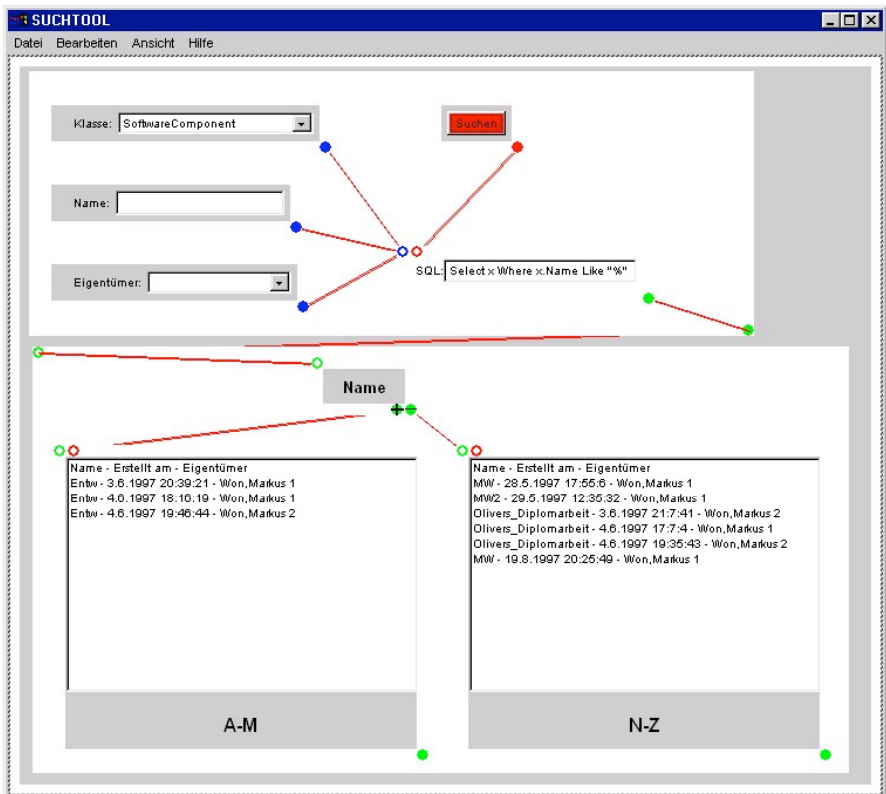


Figure 3: A 2D visual programming interface for tailorability. The white boxes represent different levels of the component hierarchy. Using the mouse and a tool box (not shown here) the tailoring user can change component connections, instantiate, delete, and position components.

While the end users managed to solve small tailoring problems during the workshop (along the lines of: "*Now try to add another output-list to the search tool.*"), they felt not really comfortable with the interface. Points of criticism were the visualization of components which were invisible during regular use of the tool (e.g. a component that managed the connection to the database of the groupware system), the empty spots in the regular use interface caused by these invisible components, and the cumbersome, mouse-based manipulation operations in general. We concluded that current visual programming techniques (at least our implementation of them) do not adequately support component-based tailorability by non-experienced end users.

An appropriate user interface for tailoring should allow the end user to quickly switch between use and tailoring mode in order to be able to adapt the system when the need for an adaptation arises. Furthermore, the tailoring mode should start out presenting the application's component structure in a way which is close to the use mode, i.e. for instance visual components should be arranged geometrically in the same way as in the regular use interface. The move from the known and understood to the new and unknown (the application's component-based implementation) should be as gradual and gentle as possible. A major problem is the arrangement and visualization of usually invisible components. A button is easily recognizable for an end user; but what about a database connection component? Furthermore, the hierarchical structure of an application should be presented in a way that end

users can move between different levels of granularity and perform manipulations on arbitrary levels.

We are currently working on a visual tailoring interface which is based on 3D representation of an application's component-structure and attempts to solve or at least alleviate the problems stated above (see section 6, "Summary and Future Work").

Technical issues

It was mentioned at the beginning of this section that we employed the JavaBeans component model and a modified version of the Java BeanBox as underlying technology for the component-based search tool application. These choices were made in order to quickly produce a first working version of an application providing component-based tailorability and gain empirical results. This subsection discusses the questions of component model and platform from a more theoretical and technical perspective. However, the considerations presented here were motivated by the experiences made during the implementation of the search tool application.

Component model: from JavaBeans to FlexiBeans

Since component-based tailoring necessitates maintaining the component structure of an application after initial development, one has to regard components as blueprints for parts of the running systems. A component can occur in several instances within the final system (for simplicity we sometimes say only "component", when it is clear from the context that actually an instance of that component is meant). A component instance has its own state. The state can either change when the component instance is in possession of a control flow, i.e. a processor is traversing the control structures (code) which are associated with the instance's state; or it can change through interaction with another component instance which is in possession of a control flow. The composition of the application determines which component instances can interact. Our work focuses on component models which provide the notion of named, typed, and polarized ports as basis for interaction. The polarization of a port determines the role it can play in a two-sided, asymmetric interaction (e.g. source or receiver in a data flow-like connection). Figure 2 shows a component which has three ports (the filled and empty circles at the edges of the rectangle). Two components can be connected when they each have a port of the same type and different polarity. In the context of a specific component, port names distinguish different ports of the same type and polarity (see *output_high* and *output_low* figure 2):

Figure 2: This example component provides three ports of type <Sound signal>. One is named *input* and has a different polarity than the other two, named *output_high* and *output_low*.

The *input* port in figure 2 could, for example, be connected to the *output_high* port of another instance of the same component. Port-based component models like this have also been used as conceptual basis for the Darwin (Magee et al. 1995) and Olan (Bellissard et al. 1996) configuration languages.

However, there are other possible component models. (Teege 1999), for instance, discusses *feature composition* as another basis for the design of tailorable applications. Features are components which do not have to be explicitly composed by connecting ports, but are simply

added to a set of features which makes up the final artifact. The feature composition approach has the advantage that possible user interfaces (which only have to support the adding and removing of elements) can be very simple to use. However, the downside is, that a set of components – while being a simpler concept – cannot contain as much information (i.e. support as many alternatives) as a net of explicitly connected components. This becomes obvious, when considering different compositions of the same set of basic component instances. Because of this advantage we employ a port based model as basis for our work[1].

We also call the type of a port an *interaction primitive*. An interaction primitive specifies how two components connected via ports of that type interact. Industrial component models like JAVABEANS usually offer one basic interaction primitive (or a small set of these), because components have to be designed by human programmers who are adverse to spend time learning a lot of different interaction styles, but rather construct more complex interactions employing simple, well-understood primitives. The JAVABEANS model, for instance, provides events as primitives. Events permit a push-like interaction, transferring a reference to a stateful entity (event object) together with the control flow to another software component. The other component is supposed to return the control flow.

In the context of the EVOLVE project, we have addressed the questions of what interaction primitives are appropriate for the approach of component-based tailoring with the help of a formal model (based on data space theory, see Cremers and Hibbard 1978, Cremers and Hibbard 1986). This model provides a mathematical notion of interacting, hierarchically decomposable computational entities and permits the precise investigation and comparison of different interaction primitives. Based on the theoretical investigation we have modified the JAVABEANS component model in order to support our approach. In addition to events, our FLEXIBEANS component model (for a detailed description, see Stiemerling 1998) offers shared objects as a "pull-like" interaction primitive without transfer of control flow. Shared objects are by nature a symmetric interaction style, because two components sharing an object can potentially both induce the same state-transitions on this object. In FLEXIBEANS, however, one component in the interaction is assigned the responsibility to instantiate the shared object in the beginning. Therefore, we model shared objects as asymmetric component connections, as the reader will notice in the example given in section four.

One can show with the help of the formal model that together, events and shared objects permit the construction of arbitrary, complex interaction styles. Another modification concerns the naming of ports. JAVABEANS only provides typed event ports, which are not sufficient e.g. for implementing the two named ports of the same type in the component shown in figure 2. The third extension is the integration of JAVA RMI (Remote Method Invocation) into the model in order to permit an arbitrary distribution and interaction of components across the Internet. This feature of FLEXIBEANS is especially important when using the approach to design tailorable groupware applications which are often – almost by nature – distributed.

Platform: from local to distributed

In the search tool application we employed the modified BEANBOX as platform, which was possible, because the search tool is essentially a local application with only one remote

---

[1] However, we agree with (Teege 1999) that these two types of composition are not mutual exclusive and can be combined to provide different kinds of tailorability within one application.

component connection to the POLITeam server. The effects of the tailoring operations were restricted to the computer running the search tool. While this architecture was sufficient for the search tool application, most groupware applications are distributed and thus demand a different supporting platform (this obvious need for distribution is not actually a result of the search tool application. The locality of the search tool was an anticipated limitation we were prepared to accept in the first experiment in order to quickly gain empirical results). We have identified the following issues or requirements for a platform for distributed component-based tailorability:

Internet- or network-based. A distributed application should run on the Internet or other networks. The prerequisites for this are given by the design of the FLEXIBEANS component model. Components at any level of granularity can be arbitrarily distributed across a network, with, for instance, a *button* component on a Sun in Tokyo deleting an item in a *list* component on an NT machine in Bonn. Bandwidth and synchronicity issues obviously have to be taken into account by the designers of the initial component set and the composers of the final application who also determine the distribution of component instances across the net.

Minimal maintenance. The system should be easy to install and maintain. Especially adding new clients and users should not be more hassle than sending somebody an email. We believe that in a world of virtual enterprises and short-term inter-company project groups, cooperation support systems have to be set-up and dismantled on the spur of a moment. The BSCW system (Bentley et al. 1997) developed at GMD is an example for an Internet-based system which offers this quality.

Decentralized and cooperative tailoring. Every user of the system potentially tailors. Thus, on the technical level, support for tailoring operations originating from every point of the network has to be provided. Furthermore, cooperative tailoring of jointly used application components should be supported by taking into account concurrent tailoring activities and providing basic facilities for coupling tailoring interfaces (e.g. broadcast mechanisms for tailoring events).

Support for different types of tailoring interfaces. Since we are not yet sure about appropriate user interfaces for tailoring, the system should support a generic technical adaptation interface not unlike an API (application programming interface). It can be used to “plug-in” different types of user interfaces for tailoring.

In section four we describe the basic concepts and the architecture of the EVOLVE system which is oriented at these issues and requirements.

Summarizing the results of the first application of the approach

The search tool implementation showed that the component-based tailorability approach is feasible and can be applied to a real world groupware application. Concerning the four questions raised in the section two, the permits the following observations:

- In finding the *appropriate decomposition* of the search tool functionality, Participatory Design methods proved quite useful. Intensive user involvement improves the designer’s insights into the diversity and dynamics of a specific field of application.
- Concerning *user interfaces for tailoring*, the concept of components was understood by our end users. However, the 2D user interface developed for the experiment did not satisfy all users and posed some conceptual (e.g. geometrical) problems.
- While the JAVABEANS *component model* proved sufficient for the implementation of the search tool example, in order to apply the approach to whole groupware applications, one has to have additional support for distributed software components. The FLEXIBEANS

component model is designed to provide this support, in addition to new interaction primitives (shared objects) and named ports.

- Analogously to the modified component model, the *platform* has to support the distribution of software components. Furthermore, the platform should provide an API-like interface for plugging in different user interface for tailorability (or other mechanisms for controlling adaptations) at any point in the network.

Based on these results, we decided to address the remaining open issues by developing a platform for the distributed deployment and composition of FLEXIBEANS. As the result of these efforts, the EVOLVE platform is described in the following section.

4. The EVOLVE Platform

All parts of the EVOLVE platform are implemented in the JAVA programming language and are supposed to run on every operation system for which a JAVA Virtual Machine of the appropriate version (1.2) exists. The tailoring interface (see next section) requires the JAVA 3D application programming interface. Figure 4 gives a schematic overview of the different elements of the EVOLVE platform:

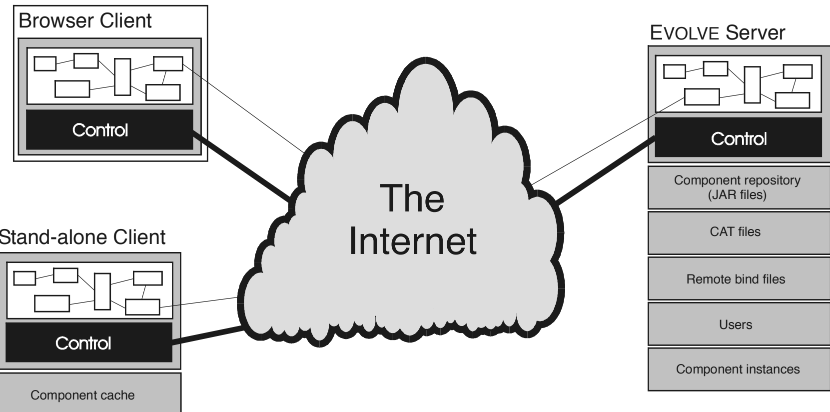


Figure 4: Basic elements of the EVOLVE platform

The EVOLVE server (right side of figure 4) is implemented as a stand-alone application, while the browser can either run as an application (lower left) or within a browser (upper left). The latter alternative has the advantage that no installation on the client computer is required if a browser is available. All code is loaded into the browser when accessing the EVOLVE server web page (a consequence of the *minimal maintenance requirement* stated in the last section). However, due to the Java security mechanisms the application cannot access certain resources on the local computer which restricts the number of application types that can sensibly be run within a browsers. There are ways (digital signatures of network loaded code) to circumvent this problem which we have not implemented yet. If the client can access the local file system (as a stand-alone client can, see lower left of figure 4), it is possible to cache components between session and thus improve start-up performance. The EVOLVE server encompasses the following elements (as indicated in figure 4):

The *component repository* contains all FLEXIBEANS which can be used to build EVOLVE applications. The FLEXIBEANS are packaged into individual JAR-files (Java ARchive) in binary format together with all resources needed for execution (e.g. auxiliary classes or bitmaps).

The *CAT-files* describe the hierarchical component architecture of the application. They refer to the FLEXIBEANS of the component repository as *atomic* (or implemented) components.

These atomic components are composed yielding *complex* components which in turn can again be part of a composition. Figure 5 shows a simple example of a CAT-file:

```
i_component FlexiBeanA {
   required Port1 Datastream;
   provided Port2 Datastream;
   config_parameter Parameter1 Integer;
}

a_component ComplexComponentA {
       required Input Datastream;
    provided Output Datastream;

       subcomponent ComponentA FlexiBeanA;
       subcomponent ComponentB FlexiBeanA;

       bind Input → ComponentA.Port1;
       bind ComponentA.Port2 → ComponentB.Port1;
       bind ComponentB.Port2 → Output;
}
```

Figure 5: (left) Simple CAT-file describing the serial composition of two instances of the atomic (implemented) component `FlexiBeanA` yielding a complex component `ComplexComponentA`. (right) Graphical representation of the example CAT-file.

The `required` and `provided` keywords indicate the polarity of the port. The `config_parameter` determines the parameterization space of a component. The specification of an abstract component additionally contains the `subcomponent` instances and their composition and connection to the ports of the abstract component via the `bind` keyword.

In contrast to the DARWIN and OLAN configuration languages mentioned before, in CAT a distributed application is defined by a set of CAT-files and not by a single specification. Each CAT-file describes a part of the application (conceptually an abstract component) which resides on one physical computer in the network. As a consequence, the `bind` commands within a CAT specification only concern local interactions.

Remote bind files describe the way components on different computers in the network are connected. A remote bind file is specific for two CAT files, i.e. it prescribes how the named ports of one component are to be connected to the named ports of the other component. Essentially, it is a list of `bind` commands which only concern remote connections.

The server also handles the *user management*. The user management contains all data necessary for authentication during log in.

CAT-files describe abstract components of an application which can be running in several *component instances* (e.g. for several users on different client computers). Thus, in addition to the list of users and passwords, it maintains for each user a list of references to CAT-files. This list points to all parts of the application which are to be instantiated on the client when the user logs into the system. The list of references also describes to which server-side components the client components are to be connected and which remote bind file is to be used for this connection. If a server component is not yet running when a client wants to connect, then it is immediately instantiated.

The *control* elements of the server and of all the clients are responsible for maintaining the component structure of the application. They offer – not yet on the user interface level but similar to an API (application programming interface) – a number of tailoring operations for runtime manipulation of the component structure. New components can be added, i.e. instantiated at every point in the network, components can be deleted or “rewired”, and abstract components can be copied. This set of tailoring operations is *complete* in the sense

that any valid (i.e. statically correct with respect to port types etc.) specification of an application based on an arbitrary component framework can be transformed into every other valid specification. The API is employed to plug different adaptation control mechanisms (e.g. user interfaces, see question two of the introduction) into the EVOLVE platform.

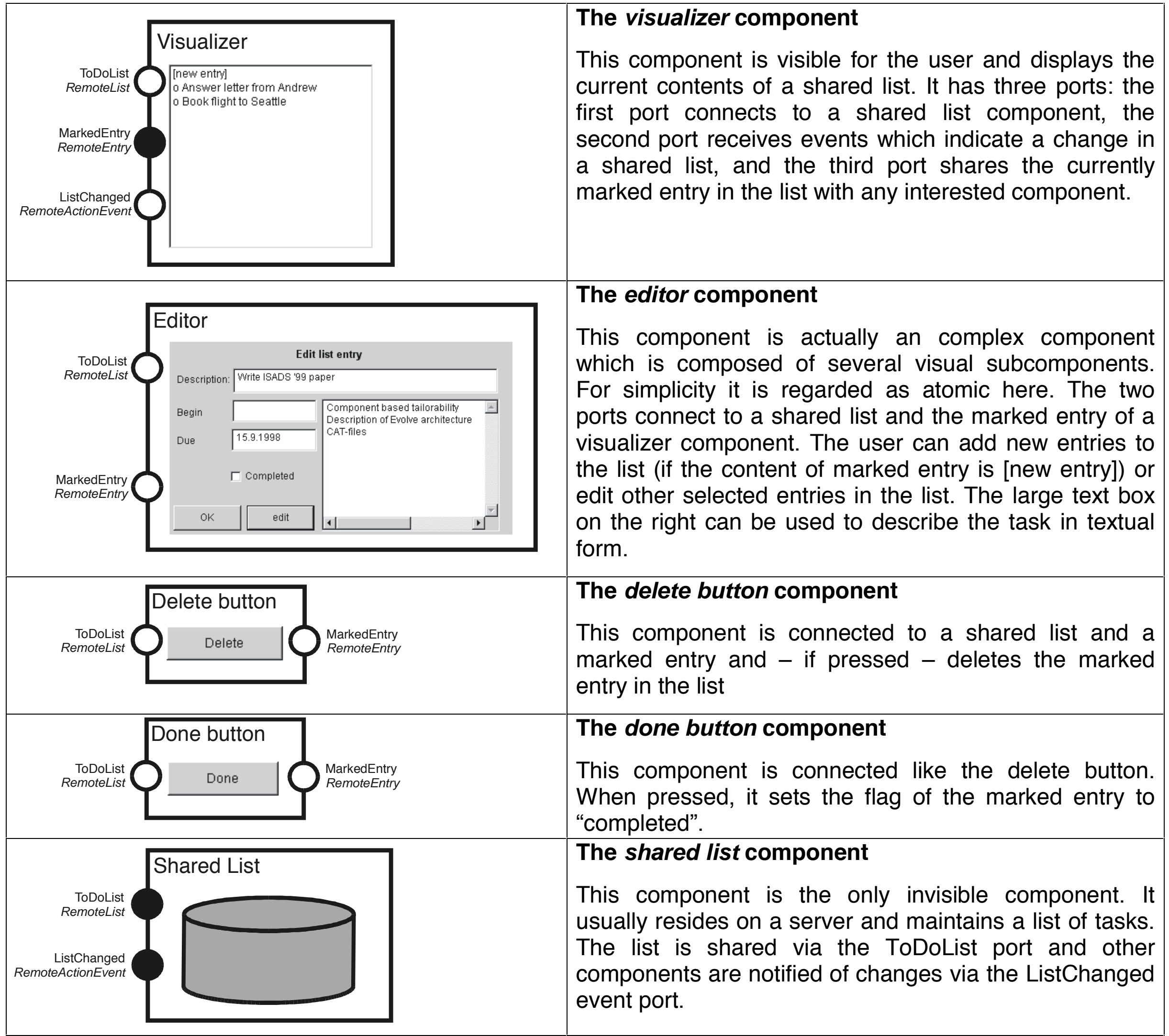
On first sight, the decision to define the application using a set of CAT-files and a list of user-specific component instances and connections induces a lot of complexity which could be avoided when simply specifying the whole application as one big CAT-file including distribution information (e.g. by providing a location pointer for atomic component instances). However, our approach has the advantage that it distinguished explicitly between local and remote connections allowing for light-weight components which only can be connected to local components without causing the JAVA RMI overhead (several new threads). Another advantage is the fact that the effect of changes to client CAT-files is restricted to users sharing the same file. Thus, our approach constitutes a natural basis for implementing control mechanisms concerning access to and scope of tailoring operations. Furthermore, it is easy to share or copy distinct parts of the application between different clients and users which supports the cooperative tailoring activities mentioned in the beginning. In the following we want to discuss a simple example which shows how the approach can be applied in the tailorable design of a coordination tool. We do not employ the search tool example here, because the search tool is essentially a non-distributed application.

Example CSCW-application: Shared To-Do Lists

Shared to-do lists (see e.g. Kreifelts et al. 1993) support coordination of work activities in small groups (2-10 persons). They contain entries which describe a task to be done, its title, begin, deadline, and a flag indicating the status of completion (in progress, completed). Depending on the structure of the group and its work habits, there can be distinct group members who add tasks, perform tasks, check for completion, and monitor or distribute work. In groups with a flat hierarchy, everybody might be allowed to add, mark as done, delete, and monitor tasks. In more hierarchical (or autocratic) organizations, only the manager might be allowed to add tasks, while subordinates are only supposed to indicate completion. As groups evolve, members fluctuate, and group tasks change, the configuration of the application has to change, new lists have to be introduced for subgroups and old lists become obsolete.

The shared to-do list application presented here is highly simplified for the purpose of this paper. However, applications with the same basic functionality are used in IT support departments, call centers, and generally for coordination in small groups which are confronted with a lot of short-term, well-defined tasks. We use shared to-do lists as an example to demonstrate the implementation of the concepts developed above, because they are simple distributed applications and exhibit a clear interdependence between evolving group activities and tailoring. We do not use the search tool application here, because of its primarily local nature.

The simplified shared to-do list framework consists of four visible and one invisible component. Table 1 gives an overview and informally describes the semantics of each component. The components rely on three different interaction types: a remote shared object `RemoteList` (shared objects are discussed above in subsection *component model*) which contains a list of tasks, another remote shared object `RemoteEntry` which contains a single list entry, and a remote event `RemoteActionEvent` which is used to notify other components of changes in a specific to-do list.



Component	Description
Visualizer	**The *visualizer* component** This component is visible for the user and displays the current contents of a shared list. It has three ports: the first port connects to a shared list component, the second port receives events which indicate a change in a shared list, and the third port shares the currently marked entry in the list with any interested component.
Editor	**The *editor* component** This component is actually an complex component which is composed of several visual subcomponents. For simplicity it is regarded as atomic here. The two ports connect to a shared list and the marked entry of a visualizer component. The user can add new entries to the list (if the content of marked entry is [new entry]) or edit other selected entries in the list. The large text box on the right can be used to describe the task in textual form.
Delete button	**The *delete button* component** This component is connected to a shared list and a marked entry and – if pressed – deletes the marked entry in the list
Done button	**The *done button* component** This component is connected like the delete button. When pressed, it sets the flag of the marked entry to “completed”.
Shared List	**The *shared list* component** This component is the only invisible component. It usually resides on a server and maintains a list of tasks. The list is shared via the ToDoList port and other components are notified of changes via the ListChanged event port.

Table 1: Components of the shared to-do list framework

When deploying the component framework of table 1 within the EVOLVE platform, the different components are placed in the component repository and can thus be referred to in CAT-files. Figure 6 shows an example application build with the component framework:

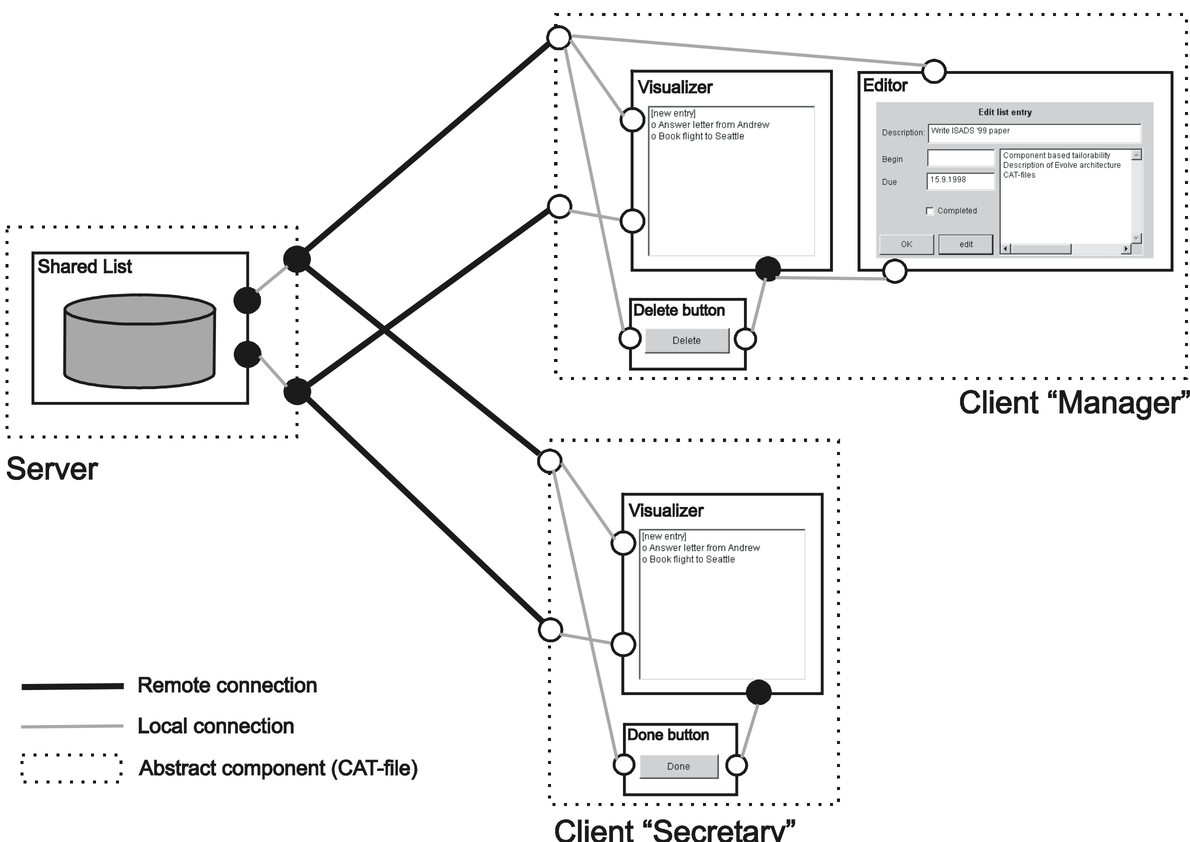


Figure 6: Simple example application providing a shared to-do list between a manager and a secretary.

The application depicted in figure 6 is specified using three CAT-files and one remote bind file. One CAT-file simply contains the one server component *shared list*, while the other CAT-files describe the client components *manager* and *secretary*. Both client components can use the same remote bind file, since the remote connections are conceptually the same.

The composition shown in figure 6 is the initial state of the application after installation. Now one can imagine many ways in which the application could evolve as an effect of changes in the group and work practices. For instance, if a second secretary joins the group, another instance of the secretary CAT file would have to be introduced. Or it could be decided that the secretary deletes finished tasks directly. In this case an instance of the *delete button* component would have to be added to the client. These manipulations can be executed by employing the tailoring operations offered by the control modules of the server and all of the clients.

We want to point out again, that the shared to-do list example is highly simplified. However, it shows how our approach supports the adaptation of a distributed application on the technical level.

5. Related Work

The high relevance of tailorability within the field of CSCW has led to a rich body of research and a substantial number of research prototypes. This section cannot give a complete survey of all systems, but focuses on specific ones exemplary for certain approaches reflecting the multiple facets of the question.

The OVAL system by Malone and colleagues (Malone et al. 1995) mentioned in section three represents one of the earlier attempts to design tailorable groupware. As discussed above, one

difference to the EVOLVE approach lies in the fact that OVAL is based on a small, fixed set of components. While these components capture a lot of the functionality needed for cooperation support, they are not fine-grained enough to permit all necessary tailoring operations without system-level programming. However, it should be possible to design a set of FLEXIBEANS components which encapsulate the functionality of objects, views, agents, and links of OVAL and use this set together with other FLEXIBEANS components.

PROSPERO by Paul Dourish (Dourish 1996) is an approach which aims more at groupware developers than end users. The system is implemented as a highly adaptable toolkit based on the reflective programming language CLOS (see Kiczales et al. 1991). Contrary to the "genericity" approach of OVAL, PROSPERO is based on the understanding that especially in cooperative work not all eventualities of the circumstances of use can be anticipated and encapsulates in one fixed toolkit or set of components. Therefore, it employs the approach of *open implementation* (see Kiczales 1996) in order to permit developers to inspect and manipulate the implementation of toolkit functionality in the face of unanticipated requirements in a specific groupware project. The primary difference to the EVOLVE approach is the focus on the development phase of the software lifecycle. Prospero does not have to deal with issues like run-time tailorability or tailoring user interfaces.

ARIADNE by Simone and Schmidt (1998) is a rather abstract model of a tailorable groupware systems. It consists of three layers, the α-, β-, and γ-layer. The latter defines the fundamental "grammar", i.e. the expressiveness of a language which is used on the β-layer to define coordination mechanisms. These mechanisms are instantiated in the running system which is described by the α-layer. In contrast to EVOLVE, the layers in ARIADNE do not concern different levels of tailorability of the system (like hierarchically structures component architectures) but different meta-levels of description. In EVOLVE, the γ-layer would be the JAVA programming language, the β-layer the set of implemented FLEXIBEANS components, and the α-layer the (CAT-) specification of the instantiated system.

DCWPL by Cortes (1999) represents an approach which clearly separates computational from coordination issues by capturing the latter in a special purpose language (DCWPL = *Describing Collaborative Work Programming Language*). The language offers constructs for session management, conflict resolution, awareness support and other groupware relevant issues. A complete groupware application consists of a number of computational modules (e.g. in C++) which at some points rely on coordination decisions which are specified separately in a DCWPL program. This program is not compiled but interpreted in a special environment, thus permitting run-time changes to e.g. floor control policies in a synchronous groupware application. The design of the language draws a fixed and definite line concerning which parts of a groupware application are tailorable during run-time and which are not. In contrary to component-based (or open implementation) approaches unanticipated requirements are difficult to address. However, the DCWPL language is quite powerful and expressive and thus probably is able to cover a lot of requirements. However, this expressiveness can also be a disadvantage, as it makes the language rather complex and thus places it probably beyond the capabilities of even some system administrators.

A number of efforts apply concepts from object-oriented programming to the design of tailorable groupware systems. The use of design patterns (Gamma et al. 1995) has received specific attention (e.g. the *adapter* pattern (Trevor et al. 1994), the *mediator* pattern (Syri 1997), and the *extensibility* pattern (Hummes and Merialdo 1999)). The primary difference of these approaches to EVOLVE is the fact that the tailoring functionality has to be explicitly integrated with the application functionality in the development phase.

The *feature composition* approach to tailorability in groupware by Teege (1999) was already mentioned in the section three. Its major difference to EVOLVE lies in the component model

which is based on sets of components rather than networks of explicitly connected components. As pointed out earlier, a combination of both approaches is possible and might yield useful system architectures.

These rather different approaches represent a cross-section through the CSCW community's efforts to address the challenges of tailorability in system design. Making groupware tailorable on a technical level is about identifying typical reusable functionality and structures (see e.g. OVAL or DCWPL), and designing software architectures which exhibit a high degree of flexibility for designers (see PROSPERO, and ARIADNE) and other tailors (see OVAL and DCWPL). Groupware-specific questions of user interface design (e.g. how to design interfaces for tailoring distributed groupware applications) are still not well-addressed in the current discussion. We hope that component-based approaches like EVOLVE permit the construction of interfaces through which even non-expert users can look and act "behind the stage" of a groupware application and tailor it to their specific preferences and requirements.

6. Summary and Future Work

This article develops the component-based tailoring approach for CSCW applications. It discusses four questions raised by the approach in the context of a first exploratory application during which component-based tailorability was employed to solve a real tailoring problem in the POLITeam project. The application's results concerning the support of distributed CSCW applications resulted in the modification of JAVABEANS yielding the FLEXIBEANS component model and the subsequent development of the EVOLVE platform. The design concepts of the EVOLVE platform were presented together with a distributed example CSCW application.

In order to address the problems in the area of user interfaces for tailoring, we are currently developing a 3D version of such an interface. The users can cooperatively navigate through a 3D representation of the distributed component world and locally and remotely manipulate compositions. Apart from the implementation of 3D user interface, its integration into the EVOLVE platform and usability tests with end users, future work on the approach also concerns the investigation of its scope of applicability. The implemented *shared to-do list* application served well to test and demonstrate the basic concepts of the approach (e.g. the distributed *CAT-files* and the *remote-bind files*). However, concerning the number and size of its components, it is rather "lightweight" and does not permit sound conclusions concerning the scalability of the approach. We will address this questions by constructing a dummy component framework, with components of different sizes, interaction types, and numbers of ports. These components will be composed into test applications which we will use to measure the execution time of start-up and tailoring operations. We expect the results to indicate the limits of the approach and to suggest bottlenecks which can be the subject of further investigations.

Another open issue is the lack of an appropriate design methodology, supporting the groupware developer in a coherent process of eliciting, representing, and validating dynamic and diverse requirements (compare questions three in section two). Further support is needed in transforming these requirements into an appropriate decomposition of application functionality for implementation in the FLEXIBEANS component model. These problems are best addressed empirically by applying the approach to a variety of tailoring problems. Apart from the development of an appropriate design methodology we expect the experiences gained by the application of component-based tailorability to give some indication whether the approach is also limited with respect to the type of target applications. Are there certain applications which cannot easily be made tailorable with the help of component-based tailorability?

7. Bibliography

Allen, R. and Garlan, D., "Formalizing Architectural Connection," in: Proceedings of 16th International Conference on Software Engineering., Sorrento, Italy, 1994.

Bellissard, L., Atallah, S. B., Boyer, F., and Riveill, M., "Distributed Application Configuration," in: Proceedings of 16th International Conference on Distributed Computing Systems., Hong-Kong, IEEE Computer Society, pp. 579-585, 1996.

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

Cortes, M., "A Coordination Language For Building Collaborative Applications," *Journal of Computer Supported Cooperative Work*, Special Issue on Tailorability and Cooperative Systems, pp. (to appear), 1999.

Cremers, A. B. and Hibbard, T. N., "Formal Modeling of Virtual Machines," *IEEE Transactions on Software Engineering*, vol. 4, 5, pp. 426-436, 1978.

Cremers, A. B. and Hibbard, T. N., "Subspaces: Factorization and Communication," in: Proceedings of Computational And Combinatorial Methods in Systems Theory, C. I. Byrnes and A. Lindquist, Eds., Stockholm, Sweden, Elsevier Science Publishers B.V. (North-Holland), pp. 383-396, 1986.

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

Ecklund, E. F., Delcambre, L. M. L., and Freiling, M. J., "Change Cases: Use Cases that Identify Future Requirements," in: Proceedings of OOPSLA '96., CA, USA, ACM Press, pp. 342-358, 1996.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*: Addison-Wesley Publishing Company, 1995.

Greenbaum, J. and Kyng, M., "Design at Work," . Hillsdale, New Jersey: Lawrence Erlbaum Associates, Publishers, 1991.

Grønbæk, K., Kyng, M., and Mogensen, P., "Cooperative Experimental System Development - Cooperative Techniques Beyond Initial Design and Analysis," in: Proceedings of Third Decennial Conference: Computers in Context: Joining Forces in Design., Aarhus, Denmark, Aahrus University, pp. 20-29, 1995.

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

Herrmann, T., "Workflow Management Systems: Ensuring Organizational Flexibility by Possibilities of Adaptation and Negotiation," in: Proceedings of COOCS '95., Milipitas, California, ACM-Press, pp. 83-94, 1995.

Hummes, J. and Merialdo, B., "Design of extensible component-based groupware," *Journal of Computer Supported Cooperative Work*, Special Issue on Tailorability and Cooperative Systems, pp. (to appear), 1999.

JavaSoft, "JavaBeans 1.0 API Specification," , Version 1.00-A ed. Mountain View, California: SUN Microsystems, 1997.

Kahler, H., “Developing Groupware with Evolution and Participation, a Case Study,” in: Proceedings of PDC '96, J. Blomberg, F. Kensing, and E. Dykstra-Erickson, Eds., Cambridge, Massachusetts, USA, ACM Press, pp. 173-182, 1996.

Kiczales, G., “Beyond the Black Box: Open Implementation,” *IEEE Software*, vol. 13, 1996.

Kiczales, G., Rivières, J. d., and Bobrow, D. G., *The Art of The Metaobject Protocol*. Cambridge, Massachusetts: The MIT Press, 1991.

Klöckner, K., Mambrey, P., Sohlenkamp, M., Prinz, W., Fuchs, L., Kolvenbach, S., Pankoke-Babatz, U., and Syri, A., “POLITeam --- Bridging the Gap between Bonn and Berlin for and with the Users,” in: Proceedings of ECSCW '95, H. Marmolin, Y. Sundblad, and K. Schmidt, Eds., Stockholm, Sweden, Kluwer, pp. 17-32, 1995.

Kreifelts, T., Hinrichs, E., and Woetzel, G., “Sharing To-Do Lists with a Distributed Task Manager,” in: Proceedings of ECSCW '93, G. d. Michelis, C. Simone, and K. Schmidt, Eds., Milan, Italy, Kluwer Academic Publishers, pp. 31-46, 1993.

Kühme, T., Dieterich, H., Malinowski, U., and Schneider-Hufschmidt, M., “Approaches to Adaptivity in User Interface Technology: Survey and Taxonomy,” in: Proceedings of IFIP '92, 1992.

Mackay, W. E., “Patterns of sharing customizable software,” in: Proceedings of CSCW '90., Los Angeles, CA, ACM Press, pp. 209-221, 1990.

Mackay, W. E., “Users and Customizable Software: A Co-Adaptive Phenomenon”, Ph.D. thesis, in *School of Management*: Massachusetts Institute of Technology, 1990.

Magee, J., Dulay, N., Eisenbach, S., and 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,” *ACM Transactions on Information Systems*, vol. 13, 2, pp. 177-205, 1995.

Microsoft, “Visual Basic,” , 4.0 ed, 1996.

Mørch, A., “Method and Tools for Tailoring of Object-oriented Applications: An Evolving Artifacts Approach”, PhD-Thesis, in *Department of Computer Science* Oslo: University of Oslo, 1997.

Nardi, B. A., *A Small Matter of Programming - Perspectives on End User Programming*. Cambridge, Massachusetts: The MIT Press, 1993.

Nierstrasz, O. and Meijler, T. D., “Research Directions in Software Composition,” *ACM Computing Surveys*, vol. 27, 2, June 1995, pp. 262-264, 1995.

Oberquelle, H., “Situationsbedingte und benutzerorientierte Anpaßbarkeit von Groupware,” in *Menschengerechte Groupware - Software-ergonomische Gestaltung und partizipative Umsetzung*, A. Hartmann, T. Herrmann, M. Rohde, and V. Wulf, Eds. Stuttgart: Teubner, 1994, pp. 31-50.

Simone, C. and Schmidt, K., “Taking the distributed nature of cooperative work seriously,” in: Proceedings of 6th Euromicro Workshop on Parallel and Distributed Processing., Madrid, Spain, IEEE Press, pp. 295-301, 1998.

Stiemerling, O., “CAT: Component Architecture for Tailorability,” Department of Computer Science, University of Bonn, Bonn, Working Paper, 1997, (available at: http://www.informatik.uni-bonn.de/~os/Publications/CAT.ps).

Stiemerling, O., “FlexiBeans Specification V 2.0,” Department of Computer Science, University of Bonn, Bonn, Working Paper, 1998, (available at: http://www.informatik.uni-bonn.de/~os/Publications/Flexibeansv20.ps).

Stiemerling, O., “Komponentenbasierte Anpaßbarkeit von Groupware,” in: Proceedings of DCSCW '98, T. Herrmann, Ed., Dortmund, Teubner, pp. 225-236, 1998.

Stiemerling, O. and Cremers, A. B., “Tailorable Component Architectures for CSCW-Systems,” in: Proceedings of 6th Euromicro Workshop on Parallel and Distributed Processing, A. M. Tyrrell, Ed., Madrid, IEEE-Press, pp. 302-308, 1998.

Stiemerling, O., Hinken, R., and Cremers, A. B., “Distributed Component-based Tailorability of CSCW Applications,” in: Proceedings of ISADS '99., Tokyo, Japan, IEEE Press, pp. (accepted for publication), 1999.

Stiemerling, O., Kahler, H., and Wulf, V., “How to Make Software Softer - Designing Tailorable Applications,” in: Proceedings of DIS '97, G. v. d. Veer, A. Henderson, and S. Coles, Eds., Amsterdam, ACM Press, pp. 365-376, 1997.

Syri, A., “Tailoring Cooperation Support through Mediators,” in: Proceedings of ECSCW '97, J. A. Hughes, W. Prinz, T. Rodden, and K. Schmidt, Eds., Lancaster, Kluwer, pp. 157-172, 1997.

Szyperski, C., *Component Software - Beyond Object-Oriented Programming*. Reading, Massachusetts: Addison-Wesley, 1998.

Szyperski, C. and Pfister, C., “Component-Oriented Programming: WCOP '96 Workshop Report,” *Special Issues in Object-Oriented Programming: Workshop Reader of the 10th European Conference on Object-Oriented Programming ECOOP ’96, Linz*, pp. 127-130, 1996.

Teege, G., “Users as Composers: Parts and Features as a Basis for Tailorability in CSCW Systems,” *International Journal of Computer Supported Cooperative Work*, Special Issue on Tailorability and Cooperative Systems, pp. (to appear), 1999.

ter Hofte, H., “Working Apart Together - Foundations for Component Groupware”, Ph.D. Thesis, Enschede, the Netherlands: Telematica Instituut, 1998, (available at: http://www.trc.nl/publicaties/1998/wat/wath.pdf).

Trevor, J., Rodden, T., and Mariani, J., “The Use of Adapters to Support Cooperative Sharing,” in: Proceedings of CSCW '94, R. Furuta and C. M. Neuwirth, Eds., Chapel Hill, NC, ACM Press, pp. 219-230, 1994.

Trigg, R. H., “Participatory Design meets the MOP: Accountability in the design of tailorable computer systems,” in: Proceedings of 15th IRIS, G. Bjerknes, T. Bratteig, and K. Kautz, Eds., Oslo, 1992.

Wulf, V., “Konfliktmanagement bei Groupware”, Ph.D. thesis, in *Fachbereich Informatik* Dortmund: University of Dortmund, 1995.

Wulf, V., Stiemerling, O., and Pfeifer, A., “Tailoring Groupware for Different Scopes of Validity,” *BIT - Behaviour and Information Technology*, pp. (to appear), 1999.