Computer Supported Cooperative Work **9:** 53–74, 2000.

Design of Extensible Component-Based Groupware

JAKOB HUMMES and BERNARD MERIALDO
Eurecom, 06904 Sophia Antipolis, France (E-mail: hummes@eurecom.fr and merialdo@eurecom.fr)

(Received 5 February 1999)

Abstract. Tailoring is identified as a key requirement for CSCW applications. One major tailoring mechanism is the extension of an application at run-time to change its behavior.

This article shows how synchronous CSCW component-based applications can be designed to be extensible at run-time. We propose to split the act of tailoring into two steps: the design-time customization of new components in visual builder tools and their insertion into the running application. Thus the customization tool is not required to be part of the application.

This article presents a new design pattern for extensibility and gives several examples based on that pattern. With the help of the pattern extensible application frameworks can be systematically created from a non-extensible application design. The different possibilities to place insertion points into the application design are discussed with respect to flexibility and ease of deployment. Finally, we present the advantages and limitations of this approach.

Key words: customization, design pattern, extensibility, Java Beans, tailoring

1. Introduction

Human beings interact in different situations and their cooperative actions depend on the context. Rather than following a predefined schedule of events, people tend to act spontaneously in creative phases. In general, it is not foreseeable how people work together; therefore, it is not always possible to define in advance which artifacts are adequate to support their cooperative process. A CSCW system that reflects these observations must allow the creation and insertion of new cooperative modules and artifacts.

Inserting new functionality into a running application is an act of tailoring. Tailoring is recognized in the CSCW literature as the key requirement for a system to adapt to different cooperative contexts (Trigg and Bødker, 1994; Malone et al., 1995). For Bentley and Dourish (1995), “support for customization is support for innovation”.

This article focuses on one important subset of tailoring: the ability to insert new functionality into an application and thus to change the behavior of the system. New functionality can be discovered by an extensible application at initialization time. It is harder to design applications that can be extended at run-time. Even harder is the design of extensibility at run-time in distributed interactive applica-

tions, such as synchronous groupware. This article presents a general design pattern to solve the latter problem.

Component-based frameworks are currently being investigated as a means to gain reusability on all layers and to adapt to change. Until recently, software engineering has focused on the development of code, which is reusable and extensible during the design phase. That focus has evolved towards the development of finished modules of code which can be reused and customized by the end-user. System modifications and extensions which were once strictly in the domain of the programmer are now being shifted into the domain of the end-user. This article applies the findings of framework research to the construction of tailorable CSCW systems, which allow the insertion of extensions on demand by the end-user. The code of the extensions is distributed on demand to every participant of the group; thus the extensions do not need to be pre-installed. Our approach provides the possibility to extend an application without terminating an ongoing cooperation.

The goal of this article is to demonstrate how component technology supports an efficient way of constructing extensible CSCW applications. Component models allow one to reason on different levels of abstraction depending on the composition level (Stiemerling and Cremers, 1998). We focus on the component model Java Beans and its supporting integrated development tools (IDE), which let the user create and customize components visually. The user may assemble components into larger composite components using the visual representation provided by an IDE rather than writing lines of code. Programmers can design the components and accompany them with special customizers to facilitate customization at design-time. In the extreme case, a new component can be assembled by only using drag and drop operations, so that even end-users can accomplish the task of creating new components.

By using only the standard and widely used Java and Java Beans technologies for their realization, our concepts become applicable and usable by other groupware developers.

If not otherwise stated, we will refer throughout the article to *tailoring* as the activities by the end-user to modify and extend the application at run-time. In contrast we use the term *design-time customization* to denote modifications at design-time.

The tailoring support for extending a running program is split into two different support-systems, the customization and the insertion support. Figure 1 illustrates our approach. In the first step, an end-user uses an off-the-shelf visual builder tool to customize a component at design-time. This component is then, in a second step, inserted into the running distributed CSCW application. Decoupling the customization tool for the components from the actual CSCW application has the following advantages for the developer:

– The product can be earlier delivered, because the tailoring functionality is not built into the product.

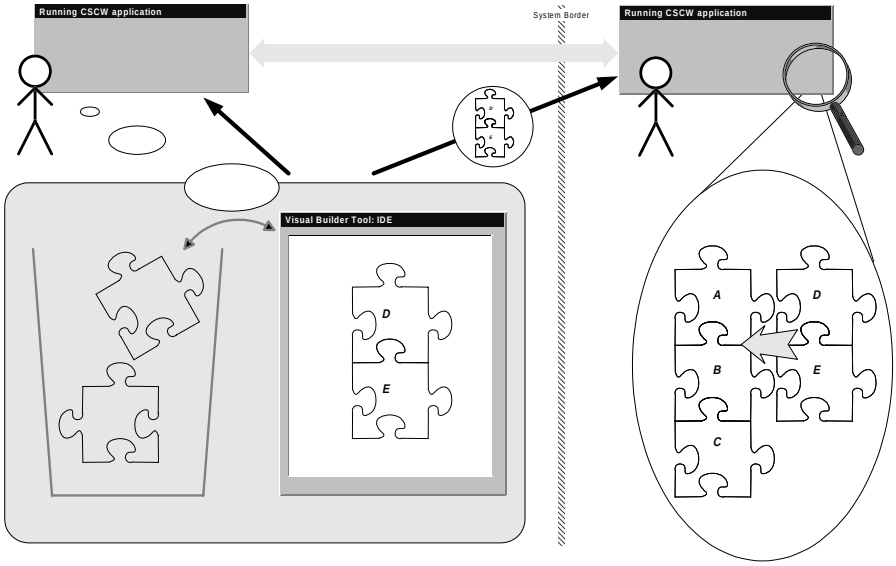

Figure 1. The two-step approach to tailor a CSCW application.

- The developer can save resources, because a proprietary tailoring tool needs not to be developed.
- General off-the-shelf IDEs are continuously improved by third party vendors.

The end-users profit from the decoupling as well. They can use their favorite builder tool for that component model and do not need to accustom to a new tool for every application. We will show that standard visual programming tools can be used efficiently by end-users to perform tailoring of CSCW applications.

2. Relevant previous work

The work described in this article is based and influenced by the research in different domains. Relevant for this work are the publications about tailorability in general and its significance for CSCW; about framework design, component technology and design pattern evolving from object-oriented software engineering; but also about new organizational forms by the business science and coordination theory under the umbrella of “virtual organizations”. All approaches have in common that they focus on evolving systems; they differ from their points of view. This paper tries to synthesize some of the findings.

2.1. TAILORING

Tailoring is defined as the activity of modifying the appearance and behavior of an application at run-time by the end-user (Trigg and Bødker, 1994; Malone et

al., 1995; Mørch, 1997). Tailoring support is normally built into the application, so that the user does not need a separate tool. In contrast to this definition, this article proposes a two-step approach, which uses customization at design-time to gain tailorability at run-time.

Before explaining our approach in greater detail, we give a brief overview of tailoring. Mørch (1997) distinguishes three levels of tailoring. The levels are classified by the design distance which is experienced by the end-user during tailoring. The first level, customization at run-time, allows to modify the appearance of presentation objects and to change their attributes. The second level allows the integration of new components or commands by composition of existing functionality within the application. The third level allows the extension of an application by adding new code. Generally speaking, with an increasing level the tailoring possibilities for a user increase, but also become more complex. To overcome the design distance, Mørch (1995) proposes to use so called "application units". Application units consist of three parts: a presentation-object, which is the user-interface, a rationale that provides meta-information about the intended use, and the actual implementation.

Tailoring beyond the first level involves end-user programming. End-user programming facilities can be offered by the application framework itself, such as in the "radically tailorable" tool for CSCW, Oval (Malone et al., 1995). The end-user can write small scripts, which are interpreted by the application, as in applications of the Microsoft Office suite (Solomon, 1995). But end-user programming can also be separated from the application and done in a general purpose language. Afterwards, the written functionality is inserted into the application at anticipated hooks. Since programming in a general purpose language is regarded as being hard for an end-user, she or he must be supported by high level tools for this task. Visual builder tools like IBM's Visual Age for Java offer an even higher abstraction than scripting languages and are thus usable by end-users (Weinreich, 1997). We will follow this approach.

On the level of programming languages, the possibility to reflect and introspect code is viewed as enabling technology to write tailorable software; reflection in component models is used to support self-representation (Stiemerling and Cremers, 1998). Reflection can also be viewed as a design pattern (Buschmann et al., 1996). Dourish proposes a reflective model for collaborative systems (Dourish, 1995) and implemented the toolkit Prospero for CSCW (Dourish, 1996) using this model to express meta-information and to allow the change of the behavior of the underlying toolkit. Component models, such as JavaBeans and DCOM, offer reflection capabilities and meta-data on components (Krieger and Adler, 1998); our approach uses these capabilities to automate interface negotiation and to offer the end-user an easy-to-handle user-interface to tailor components within a visual builder tool.

2.2. FRAMEWORKS: TOWARDS EXTENSIBLE APPLICATIONS

A framework is a skeleton of cooperating classes that forms a reusable implementation. An application framework defines the overall architecture of the applications that are created by adapting the framework. Framework-based applications are adapted by extending the framework at explicit hooks also known as “hot spots” (Pree, 1994).

Frameworks are currently successfully employed for general purpose software units, such as graphical user interfaces, system infrastructure, and middleware integration frameworks; also application domain specific frameworks are emerging.

Frameworks are distinguished into white-box and black-box frameworks (Fayad and Schmidt, 1997). Object-oriented white-box frameworks use inheritance to offer the developer extension facilities. To insert extensions into white-box frameworks the developer must understand the class hierarchy and derive new classes which have to be relinked with the framework. Black-box frameworks use object composition and delegation instead. Black-box frameworks anticipate extensions by defining interfaces and providing hooks to insert new objects.

Applications that can be extended at run-time need hooks like black-box frameworks. Unfortunately, designing frameworks – and especially black-box frameworks – is substantially harder than designing an application. However, the hot spots for a framework can be designed and implemented stepwise by a sequence of generalization transformations (Schmid, 1997). Since applications using a framework must conform to the framework’s design and model of collaboration, the framework encourages developers to follow specific design patterns (Johnson, 1997). In the other direction, developers can use design patterns to generalize an object-oriented application into a framework (Schmid, 1995).

2.3. COMPONENT TECHNOLOGY

In the field of software engineering, component based software development is seen as a major factor to facilitate reuse. Components can be purchased from third party vendors, customized and assembled within a component model. Examples for major component models are Microsoft’s Distributed Component Object Model (DCOM) and SUN’s component model for Java JavaBeans (JavaSoft, 1996). The component technology is predicted to acquire a significantly increasing importance (Kiely, 1998). Furthermore distributed component platforms are emerging, which allow interaction between components across system boundaries (Krieger and Adler, 1998).

A component is an independent “unit of software that encapsulates its design and implementation and offers interfaces to the outside, by which it may be composed with other components to form a larger whole” (D’Souza and Wills, 1998). Frameworks provide a reusable context for components (Johnson, 1997).

Components become most powerful within black-box frameworks, where they can be used to extend the hot spots.

2.3.1. *JavaBeans*

The examples in this article are implemented using JavaBeans, the component standard for Java.

The specification for JavaBeans outlines that "a Java Bean is a reusable software component that can be manipulated visually in a builder tool" (JavaSoft, 1996). Beans are self-descriptive Java classes that follow design patterns that let builder tools or applications introspect a bean. Properties reflect the accessible state of a bean. The Java Beans component model uses an event mechanism to interconnect the beans. A bean sends an event to all beans that have registered their interest in that event. The standard distinguishes two extraordinary states in the life-cycle of a bean: A bean can be manipulated in an IDE at design-time or behave like an ordinary object during run-time.

Properties and events can be manipulated within visual builder tools. The JavaBeans standard offers additional associated classes for each bean, which contain meta-information about the bean including special customizers and property editors to support a more intuitive interaction with the developer.

The component-based approach together with visual integrated development environments (IDEs) directly support our goal to be able to customize an existing application at design-time and to be able to build new similar applications by reusing the components. Beans with associated customizers allow even non-programmers to customize applications in an intuitive way. The easy grasp is achieved by the use of graphical and form-based editors within the IDEs.

2.4. DESIGN PATTERNS

Design patterns help one to reason about recurring design problems. Object-oriented design patterns describe "communicating objects and classes that are customized to solve a general design problem in a particular context" (Gamma et al., 1994). Patterns abstract from the used programming language and provide a basis for reusable design building blocks: "Design patterns are the micro-architectural elements of frameworks" (Johnson, 1997).

Design patterns are surprisingly useful to detect the hot-spots in an application design and to transform it into a domain-specific framework design (Schmid, 1995). Actually, the idea of hot spots was first introduced as a meta-pattern for framework design (Pree, 1994). In the domain of CSCW and user-interface design some patterns are well-known, such as the distributed versions of the Model-View-Controller and Presentation-Abstraction-Control patterns (Buschmann et al., 1996). Syri (1997) describes the use of the Mediator pattern to design tailorable cooperation support in CSCW systems.

To design CSCW applications that are tailorable by extension, the hot spots must be discovered in the design phase and then implemented. To ease the implementation we will introduce a design pattern which can be used to insert those hooks into the application. The pattern focuses on the ability to insert new code at run-time that conforms to an interface. By applying this pattern, one thus designs a black-box framework for a specific CSCW problem.

2.5. VIRTUAL ORGANIZATIONS

This work is also influenced by recent publications about virtual organization (Mowshowitz, 1997; Turoff, 1997). The idea of virtual organization stems from virtual constructs, such as virtual memory and circuit routing, and generalizes their concepts toward an integrating theory. One common concept in the virtual constructs is that the mapping between an abstract requirement and a possible concrete satisfier is dynamic. The mapping has to adapt as well to evolving requirements as to changing satisfiers.

The ability of virtual organized systems to dynamically adapt to environmental changes led us to think about how the dynamic exchange of software components could enhance CSCW systems.

In this context, this paper provides a technical basis to insert new satisfiers into a running groupware application when a new requirement arises. Changes to requirements for CSCW applications can be a result of evolving cooperative work patterns, for example when users become more familiar with a CSCW product or the context and goal of a work group changes (Mark et al., 1997).

3. Enabling technologies for extending CSCW applications

This section introduces a design pattern, which is used to insert hot spots in the design of applications. Since CSCW applications are inherently distributed, the pattern is accompanied with components that allow the distribution of arbitrary events to a group. By using the event mechanism and encapsulating code within an event, we place an event receiver in the pattern to allow the simultaneous extension of synchronous CSCW applications at run-time. Finally, we investigate the applicability of inserting code at run-time.

3.1. DESIGN PATTERN FOR EXTENSIBILITY

In a component model, applications are developed by interconnecting and customizing components. The components themselves are composed of other, smaller components. The design pattern for extensibility, which will be introduced here, can be encapsulated into one component.

The Extensibility pattern[1] is intended to be used to provide a default behavior, which can be changed at run-time. To change the behavior a new class can be

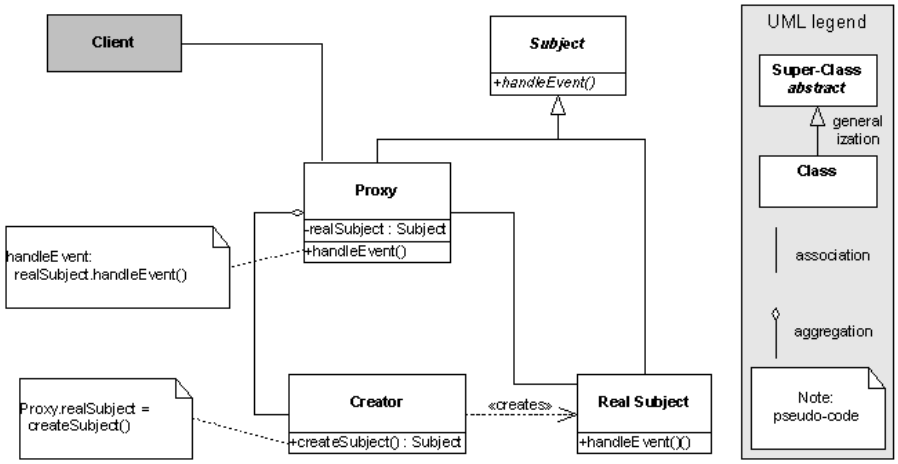

Figure 2. Structure of the Extensibility pattern.

inserted at a hook, which can either add new functionality or replace an existing class. The application sees only the specified behavior of a **Proxy** class.

The structural representation of a pattern is given by the relationship between the used classes. Figure 2 shows the structure of the Extensibility pattern in the UML notation.[2] This pattern consists of a **Proxy**, which extends the interface of a **Subject** that may be inserted at run-time.[3] Inside the **Proxy** exists a **Creator**, which is responsible to create a new object of an arbitrary class **Real Subject** conforming with the interface **Subject**. Actually this pattern is a combination of the Proxy and the Factory Method patterns from Gamma et al. (1994).

Figure 3 shows the interaction between the objects. At initialization time, the **Creator** object passes a reference to a default **Real Subject** to the **Proxy**. Any event that the **Proxy** receives is delegated to the default **Real Subject**. When the **Creator** receives an event (how that happens will be discussed soon) to create a new **Real Subject** it instantiates the respective class and sets the reference in the **Proxy** to the newly created object. The **Proxy** now forwards all subsequent events to this object, unless the **Creator** changes the reference to a **Real Subject** again.

A slight variation of the pattern allows to add instances of new classes instead of replacing the old objects. This can be easily accomplished by letting the **Proxy** store a set of all **Real Subjects**. All incoming events are then forwarded to all instantiated **Real Subjects**. This variation is useful if new functionality is added, which is independent in the application logic from the already existing objects.

3.2. REMOTE EVENTS

In an event based component model, events are the means to communicate state changes between components. The event mechanism follows the publisher-subscriber pattern (Buschmann et al., 1996). The Java Beans component model

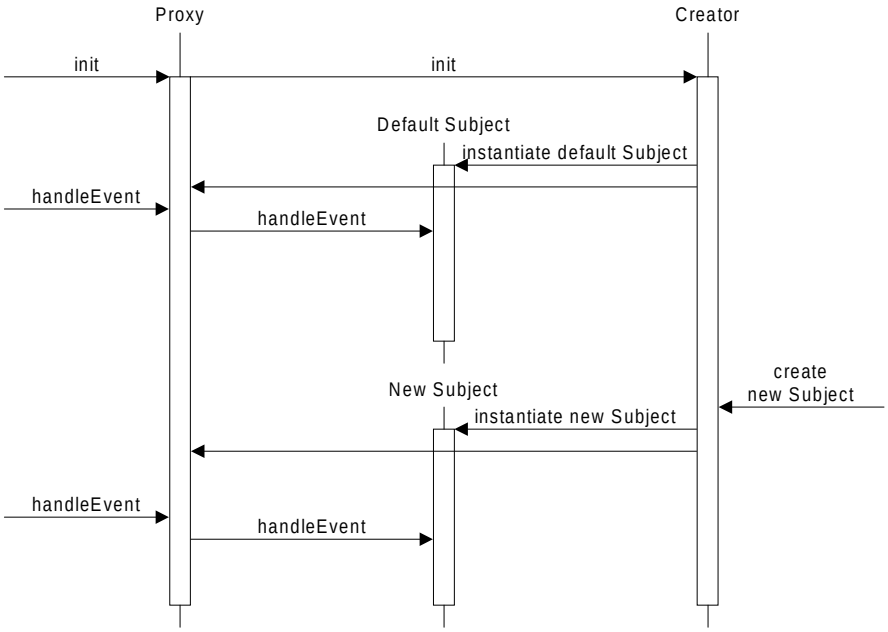

Figure 3. Interaction diagram for the Extensibility pattern.

uses such an event mechanism. We will concentrate here on Java Beans, because it is the component model we have chosen for our implementation. The design of group communication components does not rely on this particular component model, since event mechanisms are a common property of component models.

Since the Java Beans component model defines only the interaction between beans in the same virtual machine, we developed group communication beans, which act as access points for the distribution of events over process barriers (Figure 4). The group communication beans follow the publisher-subscriber pattern for a distributed platform. The group communication beans expose the event model to the developer for remote event communication. Two types of beans are necessary: The *GroupSender* forwards events to all *GroupReceivers*, which are configured with the same group name. The group name is a property of the beans and can be easily set within visual builder tools for beans, and the events can be visually connected to and from these beans. Both beans can be specialized for any event by simple object-oriented subclassing and implementing the register and handler methods for that event type. Thus the group communication beans form a white-box framework for distributed event communication. Although the definition of new events is considered as a programming activity, which goes beyond the usual capabilities of an end-user, the implementation is automated within most IDEs, thus nearly no source coding is required by the developer.

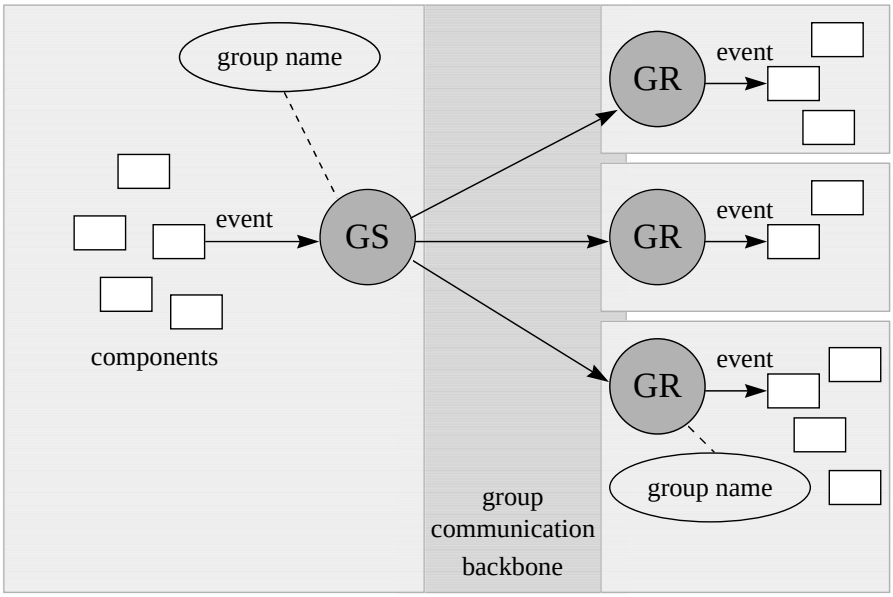

Figure 4. Group communication beans: The *GroupSender* (GS) distributes an event to all subscribed *GroupReceivers* (GR).

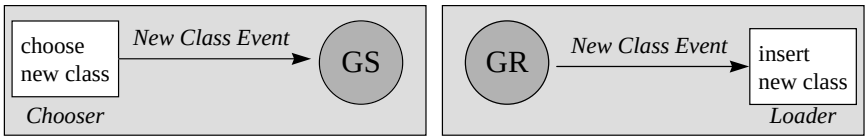

Figure 5. Distribution of a *NewClass* event.

3.3. DISTRIBUTING AND INSERTING COMPONENTS

Readers who ask themselves how the **Creator** in the Extensibility pattern is triggered, will get their answers here. The **Creator** encapsulates a *GroupReceiver* that subscribes to a group on which events may arrive that carry the classes to be instantiated. Since Java provides a platform-independent byte-code, we can directly associate the classes with the events. In other implementations, objects would have to be called remotely (by using CORBA for instance). We decided to actually distribute code, since it is a more general solution than calling remote objects. For example, a remote object would have difficulties to access system dependent resources to show a new graphical user interface.

The distribution of a new component is handled by our design as the distribution of a *NewClass* event by the beans for group communication (Figure 5). The bean, which acts as a *Chooser*, selects the class, which should be inserted in the distributed application. Often a *Chooser* is embedded in the user-interface to let the user decide, which class should be inserted. Eventually, the *Chooser* fires a *NewClass* event. The event is simply passed by the *Chooser* to a bean that is

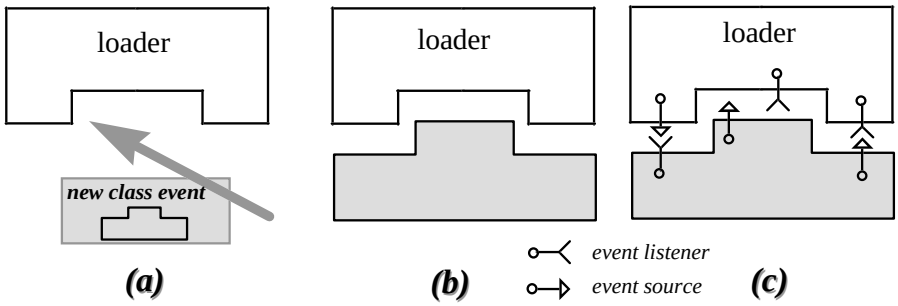

Figure 6. A loader receives a new class (a) and instantiates it (b); then the loader and the new object can register mutually (c).

derived from a *GroupSender*, which publishes it to the configured group. The event is then received by all beans that extend a *GroupReceiver* for this event type and are subscribed on that group. The *GroupReceiver* passes the event to a *Loader*, which instantiates the class. The resulting object can then be used by the **Creator** to replace or add a new **Real Subject**.

The combination of the Extensibility pattern with the group communication beans can be used to extend well specified hot spots in distributed applications; the specification is the interface **Subject**. If a hot spot defines a lot of methods, each component has to implement these methods, before it could be used to extend the hot spot. Sometimes, however, it is not feasible to be constrained by an interface. In the case of truly independent components, such as applications, it would be needed to write an adapter (Gamma et al., 1994) to insert them. On the other side, even such components may use some of the available information by the loading component. Instead of using the static information provided by the interface, a variation of our pattern uses the reflection mechanisms of Java and Java Beans to connect to the available hooks.

Figure 6 illustrates this concept. A *NewClass* event arrives at the *loader* (a). Upon arriving, the *loader* loads the class and instantiates it (b). Since the loader does not know at this time the features of the arrived bean, it uses introspection to discover the events, which can be fired by the new bean. For the events it is interested in, the loading bean adds its interest by calling the discovered registration methods (c). Now, the loading bean can receive events from the new bean. If the loading bean provides itself events and has discovered by introspection that the new bean implements the appropriate method to connect itself, it invokes that method. Then the loaded bean uses the same mechanism to subscribe itself to the events it is interested in.

The *NewClass* event may additionally carry the name of a start method. If the new class is not a bean, no events are connected, but the start method will still be called. Thus it is possible to pass arbitrary Java programs and start them remotely. The newly loaded code can interact with the loading application by means of two mechanisms: by mutual registration for the provided events that are discovered

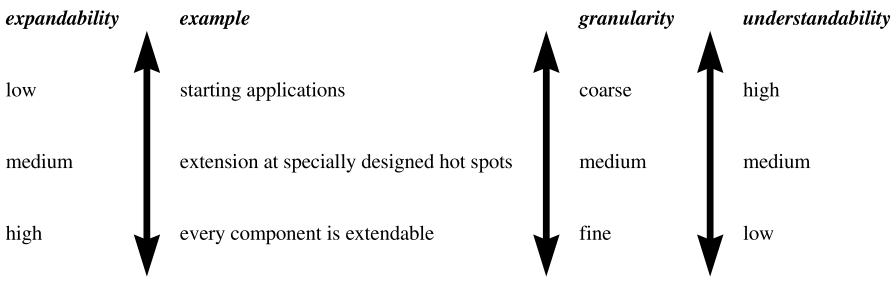

Figure 7. Trade-off between application extensibility, component granularity, and understandability for the end-user.

during initialization and by the presented group communication beans. The latter are also used to communicate with other remote applications.

3.4. APPLICABILITY

Extensibility of CSCW applications can be introduced on various levels of granularity, varying from the one extreme, where only new applications can be started, to the other extreme, where every component may be extended. The place and number of hot spots in the design determine the extensibility of the application framework. But, the number of hot spots does not only worsen the performance of the application, but it increases also the necessary effort of maintenance.

Figure 7 shows how the level of extensibility relates with the granularity of components that can be inserted and the understandability and maintainability for the end-user. The MBone tools (Eriksson, 1994) may serve as an example for very small but successful extensibility: the user can click in the session directory (sdr) on a session, which starts the needed tools to join the audio and video session. The tools are stand-alone applications, which are started in a different process. Medium extensibility is granted by domain specific frameworks with some hot spots; TeamWave (Roseman and Greenberg, 1997) is a groupware application, which uses a custom made component model on top of GroupKit (Roseman and Greenberg, 1996) to offer extensibility and tailoring support. The highest level of extensibility would be the usage of the Extensibility pattern for every component in a system.

If extensibility is only provided by means of starting applications in new processes, the original and the new application must use a protocol to exchange data, which is normally different from local interaction. Therefore, inserting components into a running application has the advantage that they can be integrated seamlessly; the new components become part of the application. The components can interact locally and use same the interaction protocol of the component model.

Our experiments with our pattern and component based CSCW applications suggest that most extensions of groupware applications happen at anticipated places. If the application uses design patterns, some hot spots can be found during

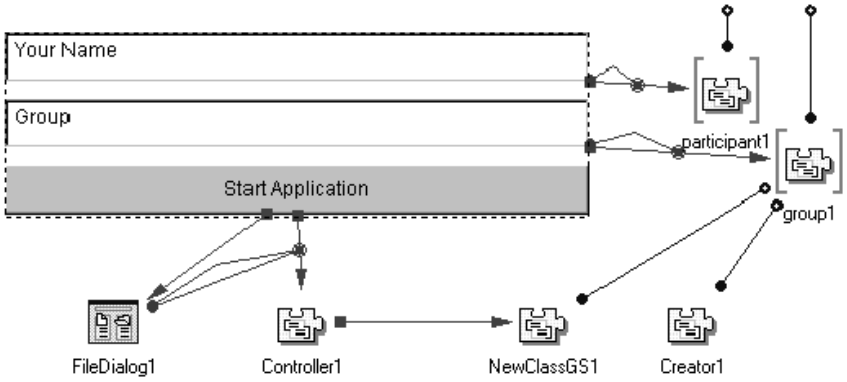

Figure 8. The loader application in a visual IDE.

the design phase (Schmid, 1997). However, it remains an art rather than pure engineering to design extensible applications. We will give some examples in the next section, how extensibility can be designed and implemented in CSCW applications.

4. Examples

This section gives some examples, how the Extensibility pattern is used to design extensible CSCW applications. The first example presents a minimal CSCW component, which is used to distribute and start other cooperative components. We use a chat component as example to demonstrate the application of the Extensibility pattern. The insertion of a voting component during a chat session highlights the use of the Extensibility pattern to support unforeseen cooperation modes. Finally, we summarize our experience of using the Extensibility pattern in tele-teaching components.

4.1. DESIGN OF A MINIMALLY EXTENSIBLE CSCW APPLICATION

An example for a minimally extensible CSCW application is a loader that offers the functionality to distribute and insert coarse grained components, which are actually CSCW applications themselves. When the user selects a new component for insertion, the code is distributed to all participants of the group and started within their instances of the loader.

Figure 8 shows the composition of the loader within a visual IDE.[4] The user interface consists of two beans to enter the participant and the group name and a button to insert a new component. When the user presses the button, a file dialog pops up, which lets the user select a component. After choosing the component an event is passed to a non-visual *Controller* bean, which generates a *NewClass* event and passes it to a *GroupSender*, which is configured with the group name. All

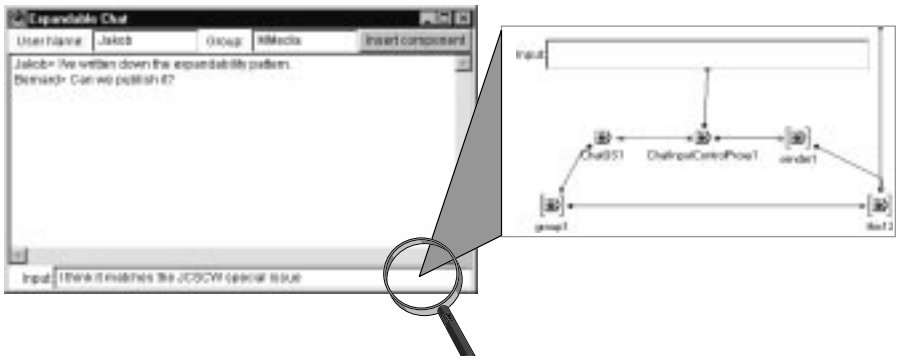

Figure 9. The design of an extensible input component (right) for a chat tool (left). The *this* variable gives access to the methods and variables of the defining bean (here: the input component).

loaders of the group members will eventually receive the *NewClass* event and start the associated component by the *Creator*. The *Creator* for the loader is configured to add every received component and to use the reflection capabilities to register for available events. The loaded component can query the properties of the loader via reflection – in this case it finds the participant and group name.

In the presented form, the loader supports the insertion of symmetric CSCW applications, i.e. applications that are executed at each participant. For example, the loader can be used to insert the components of the next examples: a chat and a voting component. We have also developed a loader component for asymmetric groupware, which supports the local insertion of a server component, and distributes clients for this component to all other participants.

4.2. DESIGN FOR FUNCTIONAL EXTENSIONS

A well-known example for a synchronous CSCW application is a chat. A chat allows the exchange of textual messages between all members of a group. This example will focus on the design of an extensible chat and present a component that can be inserted at run-time to support a simple floor control policy.

Figure 9 shows a running chat application, and the component composition at design-time for the input part of the chat. A new message is distributed by a *Chat* event to all participants; the output part of the chat component eventually receives the event and shows it to the user. The reaction on user input is performed within the bean *ChatInputControlProxy*, which has access to the input field and some environment properties. Whenever the proxy generates a new *Chat* event, it is distributed by a *GroupSender* for this event (*ChatGS*) to all participants. In this example, the user-interface additionally offers a button to insert a new component into the running application.

Figure 10 shows the internals of the proxy, which allows the replacement of the default strategy. The *BeanCreator* can receive new beans that implement the

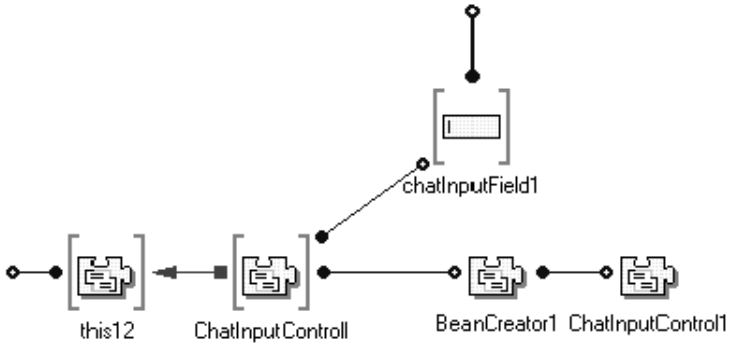

Figure 10. The design of the *ChatInputControlProxy*.

interface *ChatInputControlI*; it takes as default the component *ChatInputControl*. The input field and the current instance of the input control are associated with a variable of the type *ChatInputControlI*. Depending on the actual input control bean, a *Chat* event is fired to the proxy, which forwards it to the *GroupSender*.

This example implements a very simple mechanism to plug a new component into the running system (see Figure 11). When the user clicks on the "Insert component" button of the chat application, a dialog box pops up and the user selects the hot spot to extend. Then the user chooses from a list of available components. The actual design and implementation for the selection uses the same components as the simple loader, which was previously described. As will be discussed later, a more sophisticated mechanism should be used in real-world applications.

To add a floor control mechanism, the default implementation of *ChatInputControl* (Figure 12, left) can be replaced by *ChatInputFloorControl* (Figure 12, right) during run-time. The new component displays an additional simple user-interface to request the token for input; the input field of the chat bean is only enabled, if the user has the token. It also uses *GroupSenders* to request and release[5] a token. The newly inserted component interacts seamlessly with the existing components, since it implements the same interface *ChatInputControlI*.

The design of the chat components follows a simplified Model-View-Controller pattern (Buschmann et al., 1996). To insert components, which provides new behavior, we designed the controller of the chat input component to be exchangeable; the design uses the presented Extensibility pattern. The other chat components are designed in a similar way. Another hot spot is designed in the chat output component; a possible extension would be to add a component to write a log file of the discussion.

The chat example has shown the applicability of the Extensibility pattern to change the component's behavior at specially defined hot spots. It is thus classified in the medium level of granularity.

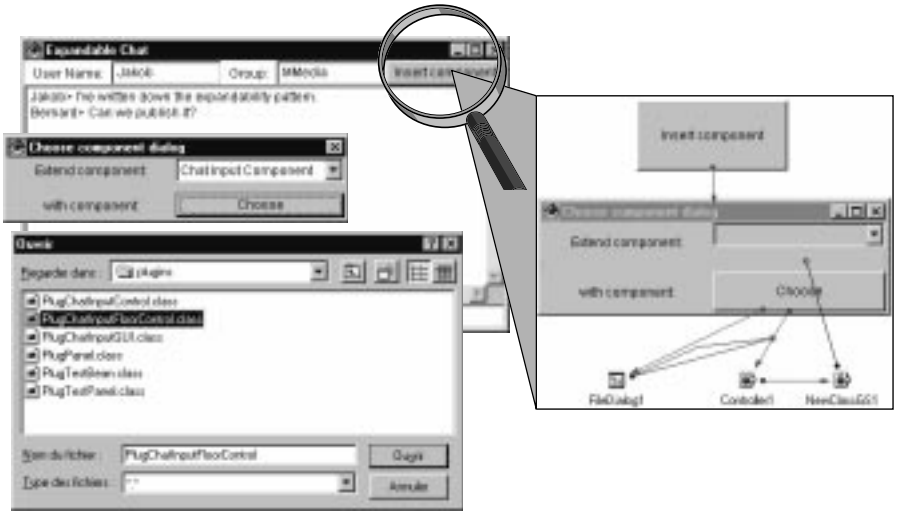

Figure 11. User-interface for insertion of a new component and its implementation.

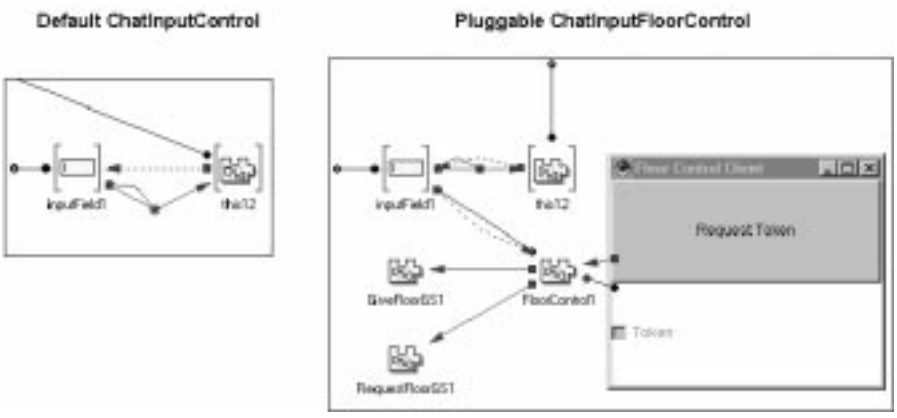

Figure 12. The *ChatInputControlProxy* is configured by default with the bean *ChatInputControl* (left); it can be replaced by *ChatInputFloorControl* (right) to support a token based floor control policy.

4.3. DESIGN OF A SECOND APPLICATION FOR INSERTION

The loader can be used to start more than one cooperative tool for all group members. For example, the chat tool is inserted for a discussion in a meeting with remote participants. After a while, a decision must be made about the discussed topics. The chair decides to create a list of the topics, and each participant has to vote for one item on the list. So, the chair uses an IDE to customize a voting component to be inserted and distributed using the loader. The voting component is shown to each participant; after a participant has submitted his vote, a separate frame shows all arriving votes from the others.

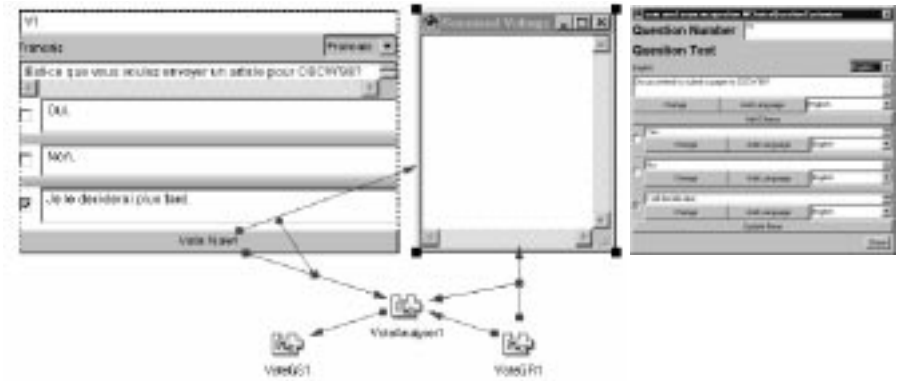

Figure 13. A voting component (left) and the associated customizer (right).

The design-time customization of a vote component is a very easy task: A question component is dropped on the vote panel within a visual IDE (Figure 13). The vote panel has an associated customizer to add new questions, to manipulate them, and to provide different language features. The customizer offers the user a graphical interface to hide all details of programming. The end-user only performs drag-and-drop operations and fills in text fields. The customizer constructs a new voting component with this information, which can then be inserted by the loader to be distributed to all participants.

This example has shown the applicability of the Extensibility pattern for coarse grained components to support the insertion of new cooperation forms. It also has validated the approach to use off-the-shelf visual builder tools to let the end user build a new component by design-time customizing existing beans, which are then distributed and inserted into the running CSCW system.

4.4. OTHER EXAMPLES

We have redesigned some of our earlier developed remote education components to offer extensibility. As an example, we placed the Extensibility pattern in remote tutoring components (Hummes et al., 1998b) to allow the insertion of arbitrary components supporting cooperation among the students and tutors. The tutoring components allow students to contact a tutor, if they want assistance in a remote laboratory course. The tutor gives peer-to-peer advice by using cooperation beans. In the original implementation the components for cooperation could be changed only at design-time; the new implementation can use several cooperation forms by inserting them at run-time. The tutor can now also distribute questionnaires (Hummes et al., 1998a) to all students at the end of a laboratory course to monitor their learning progress. The tutor has prepared the questionnaire during the laboratory course based on the issues that have been discussed with the students. The creation of such a questionnaire is highly supported by customizers within a visual

IDE. The presented customizer for the vote panel is actually a reused component for multiple choice questions from this tele-exam framework.

5. Discussion

The examples have shown the applicability of the Extensibility pattern within component based CSCW applications. By using the pattern one actually designs domain specific application frameworks. These application frameworks can be extended at run-time by inserting new components. The new components can be created by the end-user outside the application within visual IDEs.

The examples have used the Extensibility pattern to insert coarse and medium grained components. The placement of the hot spots with the pattern in the examples is based on the anticipation of possible extensions. This leads to the question whether a rule can be given where the hot spots should be located.

The main problem is that a conflict exists between the level of extensibility and the level of understandability (cfg. Section 3.4). If each component was made extensible, the design and implementation would become unnecessary complex. Even, if the performance affected by the added complexity could be improved (for example with the Flyweight pattern (Gamma et al., 1994)), the maintainability criteria still limits the amount of hot spots. On the other hand, too few hot spots limits the extensibility of the application. So, a compromise must be found depending on the domain of the application.

We found that the design of cooperation offers a good starting-point to insert hot spots in CSCW applications. Components that are triggered by user actions and perform operations depending on these actions are candidates to be extended. If the design of an application that uses design patterns, the location of potential hot spots can be derived from the design (Schmid, 1997). In the case of CSCW applications, patterns used for cooperation must be examined. In the often used Model-View-Controller pattern, a potential hot spot for extension in each application is located in the Controller, while the View would be a candidate for being extended only locally. Cooperations that can use different strategies, can change their strategies by placing the Extensibility pattern within the Strategy pattern (Gamma et al., 1994). A good example for adding a new strategy component would be a new algorithm for video encoding and decoding in conference systems. The Mediator pattern (Gamma et al., 1994) can be used to design tailorable CSCW systems by attaching cooperation enablers (Syri, 1997) to cooperative artifacts. By placing the Extensibility pattern within the Mediator new enablers could be introduced in the running system. While we have presented here a non exhaustive list of potential locations, where hot spots could be useful, it is still up to the groupware designer to decide, where hot spots will eventually be placed. Her or his analysis will be oriented on the domain of the application.

New components can be inserted on the demand of other components or on the demand of the end-user. In the latter case, the end-user must be supported by a

user-interface to select the appropriate hot-spot and component to obtain a certain behavior by his extension. We have used a simple file chooser in our examples. A more sophisticated approach would present the user the potential plug-points and a list of available components that are available to extend each one. By introspecting the selected component, such a list can be created automatically. Additionally, the user should also get a description of the intent, effects and possible side-effects for each component.

The presented implementation to insert components at run-time uses code distribution. To inform remote applications to insert a new component, we used group communication beans that can distribute arbitrary events. The needed information about the new components is encapsulated in an event. So, the implementation is coherent with the Java Beans event model. Thus it is supported by visual builder tools for Java Beans.

The distribution of code has the advantage that components have access to the local system properties. Thus user-interface components can also be distributed. Another advantage lays in increased performance compared with remote object communication if the inserted component is often used. The biggest advantage in a cooperative environment is that the component which should be inserted in the running application needs not be installed at the remote machines before the application is started.

The operation of loading and instantiating classes via the network opens severe security risks. Since Java is a network language, these risks are well-known and methods for protection exist. Java code can be signed. A signature authenticates the creator of the code. If code is manipulated after signing, this can be detected. Although signed code allows one to only accept code by trustworthy sources, the problem of who to trust remains. In a cooperative environment, this question is hard to answer. Even if all persons that are allowed to distribute new code are trustworthy, failures in the distributed code can cause damage (Zhang, 1997). The problem can be partly solved by giving explicit rights for customizing code (Stiemerling and Cremers, 1998). Another barrier can be inserted by granting new classes only the rights they need to function. If, however, a class claims to need full rights and is created by a person of full trust, the problem remains. This problem can not be generally solved.

6. Conclusion

This article has focused on the insertion of new components into running synchronous CSCW applications to tailor their behavior. We have proposed to split the act of tailoring into the steps of the design-time customization of new components within visual IDEs and their insertion into the running application. This decoupling leads to a shorter development cycle of applications. Furthermore, the end-user needs only to accustom to one IDE to tailor different applications.

When IDEs will be delivered as components, our approach can be taken to extend CSCW applications with those pluggable builder tools.

We have presented a design pattern which is focused on modeling insertion points at hot spots in a general way. Since CSCW applications are inherently distributed, we have developed components that are used to distribute arbitrary events across process boundaries to a configurable group of receivers. These group communication components are used by our examples for all remote event communication. Extensions are implemented as Java beans and distributed through remote events. They are then automatically inserted at the provisioned hot spots. Once inserted, the new components are seamlessly integrated within the running application. Independent coarse grained components that function also without information about their environment can be inserted without conforming to a predefined interface. Nevertheless they can query their environment via reflection to register for events or to read and write properties. Thus arbitrary applications can be distributed and started remotely. The presented examples have shown the use of the Extensibility pattern to create hot spots within component based CSCW applications.

This article has discussed the tension between extensibility and understandability in the design. Increasing the extensibility increases also the complexity of the design and thus decreases the understandability. This leads to the conclusion that a design is not reasonable where all components are extensible or exchangeable during run-time. So, we have revisited the design patterns, which were used in the examples, to find the locations where hot spots have been inserted. The located places have been compared with some design patterns in the literature. Thus we have shown, where the Extensibility pattern is most useful in the design of CSCW applications. However, it remains still a task for the application designers to identify the hot spots from their expertise. Once potential insertion points are recognized, the developer can uniformly design the hot spots using the introduced Extensibility pattern.

One problem, which should be addressed by further work, is how the extensibility can be presented to the end-user. The presentation should include the hot spots of an application and their possible extensions. Such a presentation must find means for an intuitive graphical user-interface to insert components at the right places.

Acknowledgments

We want to thank David Turner for proof-reading earlier versions of this article. Arnd Kohrs has provided valuable Java hints in various discussions and without his help most of our components for remote tutoring would never have been implemented. Last, but not least, we want to acknowledge the detailed comments by the anonymous reviewers.

The described work is part of the ACOST research project, which is funded by the research institute CNET Lannion of France Telecom.

Notes

1. We follow the convention to use an initial upper-case letter to name patterns, to use **bold** names for classes and to use *italic* names for beans and events.
2. Fowler (1997) provides a good overview on the UML notation.
3. In this pattern, the **Proxy** extends the **Subject** interface to be conform with the Proxy pattern. However, for the framework developer it is only important that the definition of the **Proxy** remains stable, while the developer of the pluggable components (the **Real Subjects**) relies on a stable definition of **Subject**. The **Proxy** class does not need to extend the same interface and may act as Adapter or Bridge. Proxy, Adapter, and Bridge are described in Gamma et al. (1994).
4. This and all subsequent examples are built with IBM's Visual Age for Java. A puzzle piece denotes a non-visual bean, a puzzle piece in brackets a variable, an arrow a connection between an event and a method, and a dotted line a connection between two properties.
5. This implementation implicitly releases the token after a user has sent a message. The server for the floor control is not shown here.

References

Bentley, R. and P. Dourish (1995): Medium versus Mechanism: Supporting Collaboration Through Customization. In H. Marmolin, Y. Sundblad and K. Schmidt (eds.): *Proceedings of the Fourth European Conference on Computer-Supported Cooperative Work*. Stockholm, Sweden, pp. 133–148.

Buschmann, R., R. Meunier, H. Rohnert, P. Sommerlad and M. Stal (1996): *Pattern-Oriented Software Architecture – A System of Patterns*. John Wiley & Sons, Inc.

Dourish, P. (1995): Developing a Reflective Model of Collaborative Systems. *ACM Transactions on Computer-Human Interaction*, vol. 2, no. 1, pp. 40–63.

Dourish, P. (1996): *Open Implementation and Flexibility in CSCW Toolkits*. PhD thesis, University College, London.

D'Souza, D.F. and A.C. Wills (1998): *Objects, Components and Frameworks with Uml: The Catalysis Approach*. Addision-Wesley: Object Technology Series.

Eriksson, H. (1994): MBONE: The Multicast Backbone. *Communications of the ACM*, vol. 37, no. 8, pp. 54–60.

Fayad, M.E. and D.C. Schmidt (1997): Object-Oriented Application Frameworks. *Communications of the ACM*, vol. 40, no. 10, pp. 32–38.

Fowler, M. (1997): *UML Distilled*. Addison-Wesley: Object Technology Series.

Gamma, E., R. Helm, R. Johnson and J. Vlissides (1994): *Design Patterns – Elements of Reusable Object – Oriented Software*. Addison-Wesley.

Hughes, J.A., W. Prinz, T. Rodden and K. Schmidt (eds.) (1997): *Proceedings of the Fifth European Conference on Computer-Supported Cooperative Work*. Lancaster, UK: Kluwer Academic Publishers.

Hummes, J., A. Kohrs and B. Merialdo (1988a): Questionnaires: A Framework Using Mobile Code for Component-Based Tele-Exams. In *Proceedings of IEEE Seventh International Workshops on Enabling Technologies: Infrastructure for Collaborating Enterprises (WET ICE)*. Stanford, CA, USA.

Hummes, J., A. Kohrs and B. Merialdo (1998b): Software Components for Cooperation: A Solution for the “Get Help” Problem. In *COOP ’98: Third International Conference on the Design of Cooperative Systems*. Cannes, France.

JavaSoft (1996): Java Beans 1.0 API Specification. http://java.sun.com/beans.

Johnson, R.E. (1997): Frameworks = (Components + Patterns). *Communications of the ACM*, vol. 40, no. 10, pp. 39–42.

Kiely, D. (1998): Are Components the Future of Software? *IEEE Computer*, pp. 10–11.

Krieger, D. and R.M. Adler (1998): The Emergence of Distributed Component Platforms. *IEEE Computer*, pp. 43–53.

Malone, T.W., K.-Y. Lai and C. Fry (1995): Experiments with Oval: A Radically Tailorable Tool for Cooperative Work. *ACM Transactions on Information Systems*, vol. 13, no. 2, pp. 175–205.

Mark, G., L. Fuchs and M. Sohlenkamp (1997): Supporting Groupware Conventions through Contextual Awareness. In Hughes et al. (eds.): pp. 253–268.

Mowshowitz, A. (1997): Virtual Organization. *Communications of the ACM*, vol. 40, no. 9, pp. 30–37.

Mørch, A. (1995): Application Units: Basic Building Blocks of Tailorable Applications. In *Proceedings of the Fifth International East-West Conference on Human-Computer Interaction*, Vol. 1015 of *Lecture Notes in Computer Science*, pp. 45–62.

Mørch, A. (1997): Three Levels of End-User Tailoring: Customization, Integration, and Extension. In M. Kyng and L. Mathiassen (eds.): *Computers and Design in Context*, Chapt. 3. Cambridge, MA: The MIT Press, pp. 51–76.

Pree, W. (1994): Meta-Patterns – A Means for Capturing the Essentials of Reusable Object-Oriented Design. In *Proceedings of ECOOP ’94*. Bologna, Italy.

Roseman, M. and S. Greenberg (1996): Building Real Time Groupware with GroupKit, a Groupware Toolkit. *ACM Transactions on Computer Human Interaction*, vol. 3, no. 1, pp. 66–106. http://www.cpsc.ucalgary.ca/projects/grouplab/papers/papers.html.

Roseman, M. and S. Greenberg (1997): Simplifying Component Development in an Integrated Groupware Environment. In *Proceedings of ACM UIST ’97 Symposium on User Interface Software and Technology*. Banff, Alberta, pp. 65–72.

Schmid, H.A. (1995): Creating the Architecture of a Manufactoring Framework by Design Patterns. In *Proceedings of OOPSLA ’95*. New York.

Schmid, H.A. (1997): Systematic Framework Design by Generalization. *Commiunications of the ACM*, vol. 40, no. 10, pp. 48–51.

Solomon, C. (1995): *Developing Applications with Microsoft Office: Strategies for Designing, Developing, and Delivering Custom Business Solutions Using Microsoft Office*. Redmond, WA: Microsoft Press.

Stiemerling, O. and A.B. Cremers (1998): Tailorable Component Architectures for CSCW-Systems. In *Proceedings of the Sixth Euromicro Workshop on Parallel and Distributed Programming*. Madrid, Spain, pp. 302–308. http://www.cs.unibonn.de/õs/.

Syri, A. (1997): Tailoring Cooperation Support through Mediators. In Hughes et al. (eds.): pp. 157–172.

Trigg, R.H. and S. Bødker (1994): From Implementation to Design: Tailoring and the Emergence of Systematization in CSCW. In R. Furuta and C. Neuwirth (eds.): *Proceedings of the Conference on Computer Supported Cooperative Work*. Chapel Hill, NC, USA, pp. 45–54.

Turoff, M. (1997): Virtuality. *Communications of the ACM*, vol. 40, no. 9, pp. 38–43.

Weinreich, R. (1997): A Component Framework for Direct-Manipulation Editors. In *Proceedings of TOOLS-25*. Melbourne, Australia.

Zhang, X.N. (1997): Secure Code Distribution. *IEEE Computer*, pp. 76–79.