Developing synchronous collaborative applications with TeamComponents

Jörg Roth, Claus Unger
University of Hagen, Department for Computer Science, 58084 Hagen, Germany

Abstract. Synchronous groupware applications are playing a major role in, e.g., distance education, video conferencing, joint program development, co-operative publishing, etc. There exists a variety of platforms which relieve the groupware developer from struggling with standard problems like network details, synchronisation algorithms, etc., and allow him or her to concentrate on application-specific details. Although these platforms support simple applications, groupware with a reasonable number of different kinds of artefacts (e.g. word processors with embedded documents) is still difficult to realise. Component based approaches simplify the development of such applications, but, although these approaches are quite common in single user environments, they have not yet been widely incorporated into groupware development platforms. In contrast to single-user component approaches, additional problems have to be solved: collaborative components have to communicate with other sites and multiple users, have to manage shared data, have to react on group events and have to offer group awareness services. The *TeamComponents* approach addresses these problems. It is based upon our groupware platform *DreamTeam* and covers a wide range of collaborative scenarios. A selection of sample applications with TeamComponents validates our design concept.
Keywords: Computer Science/CSCW, synchronous groupware, system design, object-oriented modeling

1 Introduction

To develop synchronous groupware is a difficult and time-consuming task. In addition to application-specific details, shared data have to be organised, communication between different sites has to be managed and user interfaces have to be developed which can handle events from multiple users.
To relive the application developer from re-designing standard modules, platforms are widely available with building blocks for standard solutions. They provide a communication infrastructure and a runtime system. Using a platform, the application developer can concentrate on application-specific details and can use collaborative services from a standard library.
With the help of platforms, a big class of applications like collaborative text editors or sketching tools can efficiently be developed, as long as each application supports a single, pre-defined kind of artefacts only, e.g. texts or images. To develop more complex applications which, e.g., support several different kinds of artefacts simultaneously, new concepts are needed. A good example for such a situation is a collaborative text processing program, which supports a variety of not necessarily pre-known types of artefacts, like embedded

diagrams freehand sketches, etc. Besides a text processor, the developer has to implement a diagram editor and a sketch editor, etc. Although separate programs may be available for these services, they cannot be used directly for embedded artefacts.
To address this problem, we realised a collaborative component concept *TeamComponents* on top of our collaborative platform *DreamTeam* ([19], [20]). TeamComponents are software components, each of them responsible for a special kind of artefact. TeamComponents offer collaborative services themselves. They can be viewed as small collaborative applications, which can be embedded into the overall application. Like applications, TeamComponents can be developed with help of the powerful DreamTeam development environment, i.e. the developer can use standard solutions from a class library.

2 Component concepts

2.1 The notion of component

Component-based approaches are currently in fashion. The notion of component or component based software development covers a wide area of concepts for single-user environments as well as for multi-user environments. Often, the term component is used in relation with graphical editors in which components can be configured and mounted to complete applications without any explicit programming. Our definition follows a different concept:

- a component is an independent software module which meets a certain set of functional requirements;
- each component supports its individual kind of artefacts;
- each components manages its own data and provides its own user interface;
- a component can either be used as a static part of an application or can dynamically be used inside a compound document.

Part 4 of the above definition describes the two major usage scenarios for components: they can either statically be integrated into the application at implementation time, or they can dynamically be called from within an application at runtime. In the latter case, the application, e.g. a sophisticated text processing program, handles the overall document structure, whereas arbitrary embedded documents, e.g. diagrams, images, spreadsheets, etc., are handled by separate components which are not necessarily known at implementation time but are dynamically integrated at runtime.

Benefits using components
The component concept provides a number of benefits to the developer as well as to the end-user of an application.
Benefits for developers:

- Once a component for a specific artefact has been developed, it can be re-used for other applications.
- The interface between an application and its component is well defined and documented.
- An application and its components can be developed independently. A component has not to know any details about its embedding application and vice versa; even the source code has not to be known.
- Components and embedding applications can be tested independently.

Benefits for users:
If a developer uses the same component for similar purposes in different applications, a user has not to switch between different user interfaces.

By just purchasing additional components, the user can easily extend the functionality of an application: a text processing application which is not able to handle spreadsheets, can be extended by adding a spreadsheet component without modifying the application itself.
Using embedded documents inside a compound document, a user can embed different artefacts and can conveniently call the corresponding editors from inside the compound document without struggling with intermediate formats (e.g. EPS).

Components in synchronous, collaborative environments
To apply the single-user component concept to synchronous, collaborative applications a variety of additional groupware-related problems has to be solved. In a collaborative environment, a component is embedded into a system of communicating sites participating in a joint session. This situation extends the set of requirements to be met by collaborative components:

- Components at different sites must be able to communicate with each other.
- Components manage their own data. In case of collaborative components, a component's data may be shared with other components and can be changed simultaneously. Thus, data distribution and concurrency control mechanisms have to be integrated.
- Components have to react to group events in an appropriate way. When, e.g., a newcomer joins a session, a component has to build the actual state with the help of already existing components.
- Components have to give their users a group feeling. For this purposes, so-called *awareness widgets* as described in [5] have to be integrated into a component.

2.2 Related work

Component concepts are often realised in single user environments in connection with compound documents, e.g. in Microsoft's OLE [12]. Text processing software such as MS Word can embed different kinds of components (called *objects*), without knowing details about the corresponding programs. Arbitrary editors can be used for diagrams, pictures etc.; painting and printing contents is done via well-documented calls. The ability to handle different kinds of sub-documents is one reason for the success of MS-Word. OpenDoc ([1], [14]) follows the same paradigms and is defined by Apple Computer, IBM, WordPerfect, Novell, SunSoft, XSoft and Taligent. In contrast to OLE, OpenDoc was designed for multiple operating systems. Neither OLE nor OpenDoc offer collaborative services. It is not possible to add multi-user services such as data replication, distributed mouse pointers etc. without re-engineering the involved applications and extending the corresponding component specifications. OLE and OpenDoc are primary designed for compound documents, there is no support for static component integration.
Sun's JavaBeans concept [6] takes up the above ideas and offers an open and flexible framework to develop components or to use third-party components in your own Java applications. It is easy to follow a Bean's specification and to realise any kind of objects with a Bean, including objects for compound documents, but also software components for completely different purposes. Beans without a graphical user interface nor the ability to handle artefacts can be defined. This flexibility with regard to the specification means a problem when using Beans inside a collaborative environment. For all components, the runtime system of a collaborative platform requires an appropriate set of standard services. Further, a component should use the communication infrastructure of the collaborative environment. Since the Beans concept does not define any rules how to handle collaborative services, it is too 'weak' for our purposes.
With CORBA remote objects can be accessed like local objects. A developer can simply use a service without the need of handling network access paths. Objects can be imple-

mented in various programming languages. They are linked to the runtime infrastructure via a language independent interface description in IDL. CORBA is not tailored for collaborative applications development, it does not offer higher-level constructs for data sharing, group communication, etc.

JViews [4] is a Java-based toolkit, built on top of the JavaBeans architecture. JView allows to develop an application by only using components. So-called repository components and view components are linked together. View components can be rendered graphically and provide editing functionality. JView components cannot be developed isolated and depend on other components. The JView approach assumes a fine granularity when splitting an application into smaller parts.

Many higher-level platforms have been developed to support the implementation of collaborative synchronous applications. Sharekit [2] and Groupkit [16] offer no support for component-based implementation. Habanero [13] is a Java toolkit for transforming applets into collaborative applets (called *Hablets*). More complex collaborative applications such as document editors are not supported.

TeamWave [17], is a groupware toolkit based on the room metaphor. The room metaphor allows to combine asynchronous and synchronous work in areas of collaboration, called *rooms*. A room can be viewed as a centrally stored compound document, to which all team members have access. The room metaphor requires a well-known server on which the room has to be stored. The usage of components is limited to rooms, i.e. applications themselves have to be developed conventionally.

A more document oriented approach is offered by DOLPHIN [23]. DOLPHIN is a collaborative hypermedia editor and is realised on the platform COAST [22]. A hypermedia document in DOLPHIN contains nodes, links and multimedia data such as sketches, texts and images. The approach focuses on gaining consistence in a collaborative hypermedia document. There is only limited support for developing and integrating new components into DOLPHIN.

Clock [3] allows component-oriented development and offers an MVC-style runtime architecture. A developer first divides the program with the help of a graphical editor into data objects (model) views and controllers. Each method can then be programmed in a functional language. The overall structure helps the runtime system to manage data distribution. Communication between sites can only be done via data objects, thus the implementation of a view or controller component requires the knowledge of the corresponding model. Since all shared models are located in a special public node of an application, components and applications cannot be realised separately. Clock components cannot appear inside a compound document.

CoCoDoc [7] uses the CORBA and OpenDoc infrastructure to offer component services to groupware developers. Since both, CORBA and OpenDoc, have not been designed for synchronous groupware, group oriented services are difficult to integrate.

The groupware project IRIS [9] follows a component based approach [8]; it focuses on asynchronous work. It provides only limited support for synchronous collaboration, e.g. no synchronous awareness widgets are offered, concurrency control is optimistic and needs user interaction in case of conflicts. IRIS components can be viewed as applets rather than components in our sense. IRIS components cannot be used inside compound documents.

3 The DreamTeam environment

DreamTeam is a platform for synchronous collaboration; it offers a variety of services for application developers as well as end-users. The DreamTeam environment allows the developer to develop co-operative applications like single user applications, without

struggling with network details, synchronisation algorithms, etc. DreamTeam has been realised in Java [25] without the need for any native code, i.e. it can be run on many platforms. It contains of a development environment, a runtime environment and a simulation environment. The development environment [18] mainly consists of a huge Java class library which contains groupware specific problem solutions as building blocks. The runtime environment provides an infrastructure with special groupware facilities. A front-end on top of the runtime environment allows end-users to control and configure the system. Finally, collaborative applications can be tested in the simulation environment, which allows to simulate network characteristics on a single computer.
DreamTeam is based upon a completely decentralised architecture, thus there is no central server for storing session states. Though the decentralised architecture leads to more complex algorithms, it avoids performance bottlenecks and ensures higher reliability.
Lauwers and Lantz distinguish two kinds of groupware applications [10]: *collaboration-transparent* applications are single-user applications made available in a collaborative environment. In contrast, *collaboration-aware* applications include additional code to support the group, e.g. they offer special awareness widgets and overview windows. We strongly believe that only collaboration-aware groupware applications will really gain acceptance by end-users. DreamTeam supports collaboration-aware applications. Unfortunately, developing such applications is very expensive. In addition to group-specific services, the actual application function has to be developed. If, e.g., we want to develop a collaborative text processing software, besides the group services such as telepointers and overview windows, the actual text editing function has to be developed as well, although programs such as MS-Words are already available which provide this function. Applications off-the-shelves offer neither source code nor documentation for modifications, and thus cannot be used as basis for groupware applications.
Having to develop both, group specific services as well as the program functionality, means a double burden. In the following, we present a concept, which counteracts this disadvantage and helps to develop collaboration-aware groupware applications more efficiently.

4 TeamComponents

4.1 The concept

A component-based approach can decrease the efforts for developing new applications drastically. We designed the component concept *TeamComponents* which both support dynamic usage (inside compound documents) as well as static component usage. Figure 1 shows the internal structure of a TeamComponent.

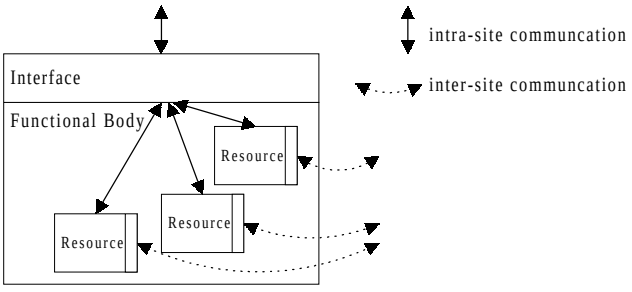

Figure 1: Internal structure of a TeamComponent

A standard component interface, not to be confused with a component's individual user interface, is defined within the component framework and ensures a component's proper

integration into its embedding application. Messages from or to a component can only be routed through its interface. A component's functionality is defined in the functional body. A component may be divided into *resources*, which themselves can be composed of other resources.
The TeamComponent architecture allows two kinds of communication between collaborative components:

- *intra-site communication* allows a resource to communication with its application via its interface.
- *inter-site communication* allows a resource to communicate with its peer resources at other sites.

Figure 2 shows, how TeamComponents are embedded into DreamTeam's communication infrastructure.

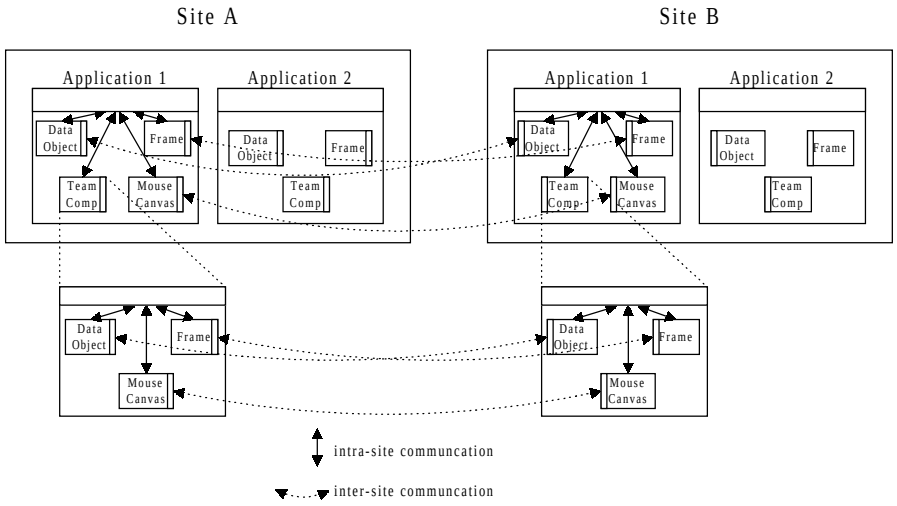

Figure 2: Communication infrastructure

According to a session's goal, a session may use one or more applications, which synchronously run at every participating site. E.g. for a brainstorming session, a brainstorming tool, a drawing tool and a Web browser may be helpful.
Like components, applications can be divided into resources. Resources are either related to the user interface such as frames or are data objects. In our example, the left application is divided into one data object and two user interface objects: Frame and MouseCanvas (see below). The forth resource is a TeamComponent. TeamComponents are special resources, which themselves can be composed of other resources. In contrast to resources, in the current version TeamComponents cannot be composed of other TeamComponents.
Application resources communicate in the same way as resources inside a component. They can communicate with their embedding runtime system via their (intra-site) application interface or can communicate with their peers at other sites. Both kinds of communication are realised by events or call-backs. Inside a site, standard call mechanisms are used, between sites event distribution is handled by a message-based communication system. The developer has no direct access to the underlying message system, i.e. the corresponding calls are available as local procedure calls, which automatically initiate communication. The runtime system ensures that at all sites identical resource structures are being used.
Resource of one application must not directly communication with resources of other application at the same site, nor can resources of different components communicate directly.

Because all implementation details of a component are hidden, a resource of component X must not directly communicate with any resource of component Y, even if both components are running within the same application.

4.2 Managing shared states

Because of to the fully replicated architecture of DreamTeam all components states have to be replicated as well. A component is responsible for its own data, data distribution and data replication. For this purpose, a component developer can use data object resources and high-level platform services for concurrency control and data distribution.
The basic means to achieve data replication is a form of multicast method invocation. A special method, called `distribute`, allows to invoke a method on all replica of the calling resource. If, e.g., a resource calls

```
distribute(TO_ALL,"anyMethod",anyParam1,anyParam2);
```

the method

```
anyMethod(anyParam1,anyParam2);
```

is invoked on the corresponding resources on all sites. The first parameter is called *modifier* and allows to configure a multicast call. It is possible to configure

- the group of recipients (only one, all in a session, all but the caller),
- whether the call is transmitted via a reliable connection or via a fast but unreliable connection (e.g. via multicast-IP),
- whether the call is protected by a lock.

The last point allows to implicitly use concurrency control mechanisms during a call. A call can be protected by a lock, i.e. as long as the specific method code is processed (on all sites), other calls are blocked. Three kinds of locks can be used:

- A *transaction lock* exists once per application. If one owns this lock, all other callers requesting the transaction lock are blocked, even when running on other sites. The transaction lock is the most restrictive lock.
- A *monitor lock* exists once per resource, thus parallel method invocations are possible on different resources. Only calls applied to one resource are blocked.
- *Explicit locks* can be allocated by the developer for any variable and any method call.

These kinds of locks supports different levels of concurrency control. The transaction lock is the most secure kind of locks, but prohibits possible parallel processing. The explicit lock on the other hand can be applied in a very fine-grained way, but needs careful planning by the application developer. The choice for a lock depends on the specific resource implementation. In our opinion, there is no generic way to decide, which lock has to be used for a certain problem.
The runtime system automatically handles latecomers: from the site of a participating user, an existing resource structure and the contents of all shared data objects are transferred to the latecomer's site. Since component and application states highly depend on the actual implementation, there is no generic way for the runtime system to build a replicated state automatically. Instead, the application developer has to implement two methods for each application and component; one to store a state to a data stream and one to build a new state from a stream. Calling these methods and creating the communication stream is done automatically by the runtime system. In most cases, streaming internal states can efficiently be developed by using Java's Object Serialization concept [24].
Direct inter-site communication between resources allows to develop a component in a more module-based manner: the developer of a TeamComponent has not to rely on data objects inside the application, but can locate component-related data objects inside the

component. The data objects themselves can be developed without knowing details of other parts of the component. This concept increases a component's efficiency. To provide, e.g., distributed mouse pointers, a mouse canvas is responsible for multicasting mouse positions to other sites. A canvas object can perform this task by directly communicating with remote canvases without using any data object. Our approach of direct communication is much more efficient than an update via a data object.

4.3 Using components

Components can be used in different ways, described by two properties *isolation* and *integration*. Isolation describes the level of collaboration, while integration describes the way how a component has been integrated into an application.

Isolation

There exist two isolation levels, *private* and *public*, where public is divided into *shared* and *exclusive*. Private components can reside in collaborative applications but their data are not shared with other applications. With private components, a user can, e.g., prepare his own graphics before publishing it to other users. Any TeamComponent can run in private mode, the runtime system blocks it from any inter-site communication, the component runs as if the session has only one member, the actual participant. Although this is a generic way to put a collaborative component into a private mode, often a component wants to react on its current mode and enable or disable some services itself: distributed mouse pointers are not helpful in private mode and may confuse the user, thus a component may wish to hide the corresponding menu entry. For this purpose, components can ask for their current isolation level.

Public exclusive components have shared data but at a certain point of time only one user may manipulate these data. In such a mode, floor control mechanisms can easily be realised. The runtime system controls the usage of an exclusive component. If a component edits its contents, other components are automatically prevented from editing.

Finally, public shared components control data which can be manipulated simultaneously. Public shared components offer the highest level of collaboration and thus are much more complex to realise than exclusive components.

Integration

The integration property defines how of a component is integrated into its embedding application. We distinguish between *seamless* and *anchored* integration. Figure 3 illustrates the different integration modes.

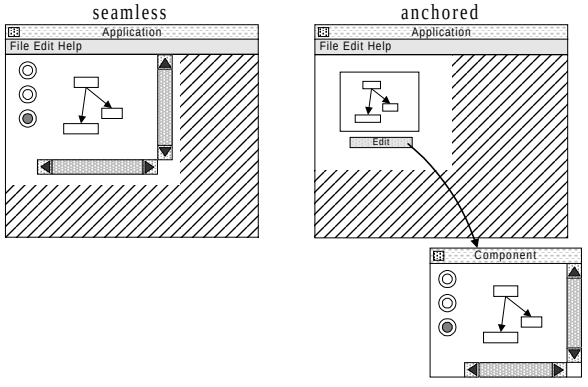

Figure 3: Integration modes

In seamless mode (often called *in-place editing*), a component's user interface becomes part of an existing window. When this kind of integration is difficult to realise, anchored integration can be used instead. The window, which displays the graphics, contains an anchor normally displays the artefact e.g. the graphics. For editing the artefact, the user has to activate (click) the anchor and thus open a separate editing window. This approach allows the runtime system to detect the editing and, e.g. in case of mutual exclusion, activate the corresponding access control. To detect such an interaction in a seamless component is much more difficult.
We finally divide anchored components into *dynamic* and *static* ones. Dynamic components can be instantiated during runtime on user's demand. Dynamic components usually reside inside compound documents, they are not available during compile-time, thus the corresponding binary code has to be loaded during runtime. In addition, compound documents have to provide special document services like layout and printing functions.
In contrast, static components are statically integrated into their the corresponding application and thus are known at compile-time. During a save dialogue, e.g., users can collaboratively define a comment which can be displayed later in an overview window. The comment component is a static part of the save dialogue, whose behaviour cannot be changed at runtime.
Figure 4 shows the different types of components and their characteristics.

			Integration		
			seamless	anchored	
				static	dynamic
Isolation	public	exclusive	not allowed	referrable at compile-time only one can edit separate edit window	instantiated on demand only one can edit separate edit window
		shared	no separate edit window collaborative editing	referrable at compile-time collaborative editing separate edit window	instantiated on demand collaborative editing separate edit window
	private		no separate edit window private editing only	referrable at compile-time private editing only separate edit window	instantiated on demand private editing only separate edit window

Figure 4: TeamComponent categories

A TeamComponent can be realised in such a way that it can be used in different integration or isolation modes, i.e. normally a component covers several fields of the eight potential fields in table 1. An application can "ask" a component in which mode it can run and then use it in the appropriate way.

4.4 Group awareness

A groupware user interface has to handle two contradictory tasks: on one hand it has to give the user the impression of working in a group (*group awareness*), on the other hand the interface has to limit the noise of group events. Sasse et al. call the latter effect *interference* [21]. In extreme situations, users can be overwhelmed by system messages or notifications and prevented from continuing their work in an orderly way. This problem becomes even worse, if components are involved: since components are small applications inside an application, the amount of interference increases if no noise reduction mechanism is integrated.

To address this problem, we distinguish two levels of information, a system gives to its users. First level information is given automatically, it is not very detailed, provides the user with a rough event classification and is displayed in the form of unobtrusive state icons. Figure 5 shows all possible state icons.

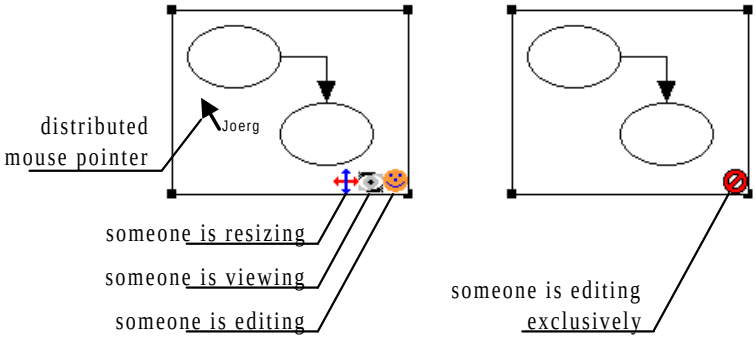

Figure 5: Group awareness widgets

The small icons on the lower right corner of the left component indicate that:

- someone is currently resizing the component;
- someone else has opened the component and views its contents;
- someone else is currently editing the component's contents.

The icon in the right window indicates, that another user is currently editing a component in exclusive isolation mode. For components, which can be opened collaboratively, an editing or viewing icon occurs, when at least one other user is currently editing or viewing the component.

Second level information has to be requested explicitly by the user. For every first level state information the user can ask for the exact list of users who are involved. The user, e.g., can ask for the exact list of users who are currently editing a component. For this purpose, a menu item "Who is editing this component?" is added to an application window.

This two level information system helps users to keep track about their work without getting overloaded by group events. Distributing state information and displaying the corresponding icons is done automatically by the runtime system. The application or component developer has not to implement this service in the application or component code.

4.5 The TeamComponent class hierarchy

An object oriented library supports the use of TeamComponents. Figure 6 shows the UML class diagram [15] of the TeamComponent library package.

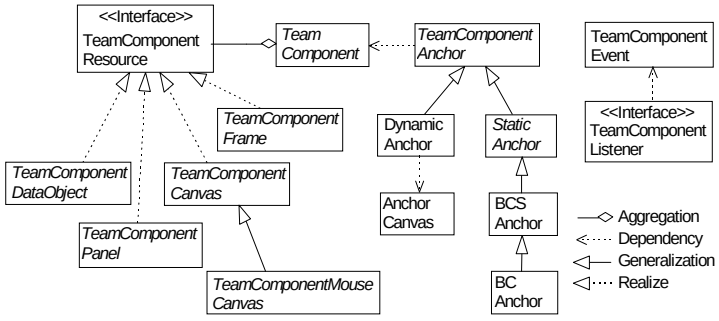

Figure 6: UML diagram of the TeamComponent package

A developer has to perform the following steps:

- Build a subclass of the library class *TeamComponent*, which defines a component's standard behaviour. In this subclass, only non-standard functions have to be coded.
- Write individual classes where necessary. For some Java classes, counterparts exist which define a standard group behaviour inside a component. The classes *TeamComponentFrame* and *TeamComponentCanvas*, e.g., are component variants of the Java classes *Frame* and *Canvas*; the class *TeamComponentMouseCanvas* provides distributed mouse pointers; the class *TeamComponentDataObject* supports replicated data objects.

The classes *TeamComponentListener* and *TeamComponentEvent* allow to register for component events. In case of an event, a registered object is called via a predefined method which allows to react to the event. Component events are opening or viewing a component via its anchor, closing a component, etc.
Anchors are represented by the class *TeamComponentAnchor*. Anchors of dynamic or static components are instances of class *DynamicAnchor* or *StaticAnchor*. Static anchors are available with different shapes: the *BCSAnchor*, e.g., consists of a button, a canvas and two scrollbars, whereas the *BCAnchor* consists of a button and a canvas only.

4.6 The interface

The class *TeamComponent* provides the main interface between application and component (see Figure 1). The interface offers method calls for a big variety of scenarios; most of them are predefined, some have to be individually coded by the component developer:
Component profiles: these calls return a component's profile data and help the runtime system to organise a specific component: to produce a list of all available TeamComponents, a component has to return its name and its icon; each component has to specify in which integration mode it can be used (anchored, seamless or both) and which isolation level it supports (shared, exclusive or both). To use the same component with different realisation levels in the same session, version numbers can be assigned to components. Version conflicts can then easily be detected by the runtime system, inconsistent data states are avoided.
Usage modes: the application can determine, in which mode an actual component has to run. Such a mode has to be chosen from the component's available modes, e.g. if a component can either be used shared or exclusively, the application can enforce an exclusive usage. Illegal settings will be rejected.
Anchored components: if a component offers anchored usage, it has to implement methods for attaching an anchor object, for opening (for editing or viewing) and for closing the component.
Seamless components: if a component can be used seamlessly, it must be able to embed itself into an existing window. For example. when a dialogue window with menu bar, buttons etc. is existing already, a seamless component has to embed itself without affecting existing dialogue elements. For such a purpose, a component can realise its own dialogue elements in a reserved, rectangular area of an overall dialogue frame.
Printing and painting: text processing applications usually allow to display and print a document. If a component is part of a document, the application cannot know, how the component has structured its data, nor how its contents can be presented. Thus displaying and printing has to be done be the component itself. Hereto, the application asks the component to display (or print) its contents in a predefined rectangle on the screen or the printer page. Since in Java printing and displaying is performed by the same mechanism, the component developer has only to code a single function to realise both services.

Events: in order to receive component events, an arbitrary object can be registered by the runtime system. Our event concept follows Java's awt event concept.
States: a component maintains its own data. To store and retrieve a component's state, *getState* and *setState* methods have to be provided. Since a state is a *serializable object*, it can easily stored to disk or transferred between sites using Java's Object Serialization mechanism [24] Beside for persistence purposes, the state methods are needed for managing newcomers: when a newcomer enters a running session, all required components are instantiated by the platform and the actual state is set automatically from existing component states.

5 Samples

5.1 Sample components

To test and validate our component design concept, we developed five sample components. To detect potential weak points in our specification, these components cover a wide range of artefacts:
Simple drawing component: with this component, simple freehand sketches can be created using a mouse or a sketch pad. If the component is used inside a collaborative session, all session members can collaboratively edit the sketch. Individual mouse pointers can be distributed among session sites.
Diagram component: this component supports diagrams such as UML, flow charts, entity relationship diagrams. Like the simple draw component, this component can be used in public and shared mode. Distributed mouse pointers are provided.
Text input component: this component supports simple text processing. It can only be used exclusively in public mode. Plain text can be entered and formatted with different font styles.
WWW component: this component allows to browse Web pages collaboratively, i.e. it can be used as a small Web browser. It stores the current page in its data structure in order to present it, even after the Web connection has been released. Distributed mouse pointers use semantic mapping, i.e. when someone points to a specific word or graphics, the corresponding pointers on other sites point to the same item, regardless of different line breaks.
Audio component: the audio component allows to store and reproduce sound, e.g. spoken text. Since Java does not allow to access audio hardware directly, we had to use native C code inside the component's package. This component is very useful for 'voice annotations' to documents.
We used these components for a set of sample collaborative applications, which will be sketched in the following chapter.

5.2 Applications using TeamComponents

Editing compound documents
The first example shows a collaborative document editor for compound multimedia documents. A canvas allows to place and edit components co-operatively. Figure 7 shows a sample document using different TeamComponents as part of the document contents: a diagram component and a sketch component.

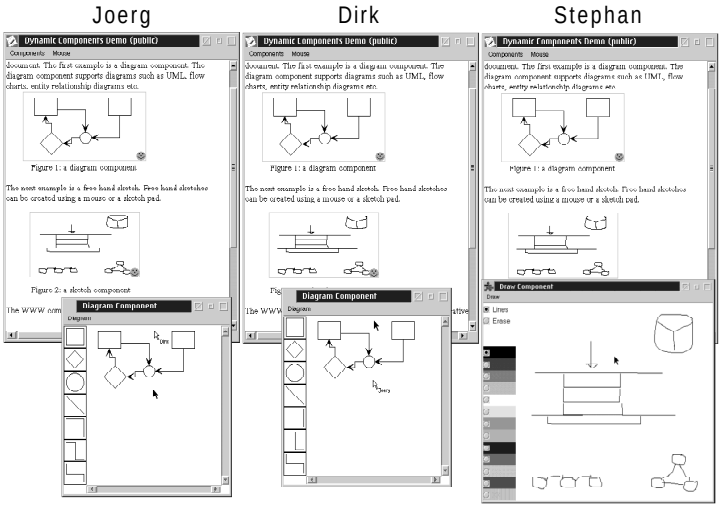

Figure 7: A document containing TeamComponents

Inside the application, only the mounting area has to be coded. Any kind of TeamComponent which supports the *anchored, dynamic* integration can be used inside a compound document. In this example, three members of a group are editing a document collaboratively. Joerg and Dirk edit the diagram; Stephan edits the freehand sketch. As Joerg and Dirk edit the same artefact simultaneously, they can see each other's mouse pointer. In this scenario, the users have the following possibilities to continue their work:

- Stephan can close the sketch component and can join editing the diagram.
- Joerg or Dirk can join editing the sketch.
- Each user can close the component and can edit the overall document structure, e.g. change the text or add new artefacts.

When a user opens a component which has already been opened by other users, his or her distributed mouse pointer is automatically displayed on other sites.

DreamView - a collaborative Web browser

DreamView is a collaborative Web browser developed in the DreamTeam environment. Beside providing a useful tool for e.g. distance education [11], DreamView was developed for testing and validating the DreamTeam as well as the TeamComponent design concept.

DreamView allows a group of users to co-operatively browse the World-Wide Web. The reasons for implementing a co-operative Web browser are manifold. In addition to browsing remote documents, Web browsers can be used to browse local HTML documents, e.g. manuals and learning materials that have been published as HTML pages and distributed on CD-ROMs.

Beyond common browser functions, DreamView allows to annotate items in Web pages. Annotations can be viewed and discussed by all session members simultaneously. Annotations can be chosen from a variety of artefacts like texts, freehand sketches, voice, etc. Figure 8 shows a DreamView window. The right part of the windows is used for annotations, which can be linked to the page contents on the left side.

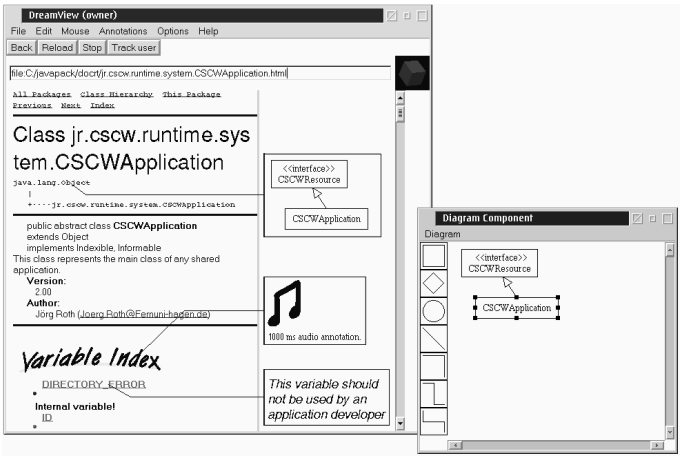

Figure 8: The DreamView browser

To attach an annotation to a Web page, one has to proceed as follows:

- Select the "Add annotation" menu from the Web browser. The system presents all available dynamic components in a select box. This list is computed on demand, e.g. it is not hard-coded in the browser's code.
- Choose the page item to which the annotation has to be attached. The browser adds an empty component of the appropriate type. Draw a line from the annotation to the page item and move and resize the component anchor.
- With the "Edit annotation" function, the user can open the corresponding component and edit the annotation (Figure 8).

A user can attach as many annotation as desired and delete unused annotations. Public annotations are automatically distributed among the session. To increase the space for the page, the annotations' area can be hidden.

A brainstorming tool

DreamTeam's brainstorming application is an example for the use of static components. It allows a group to collect and evaluate ideas. Ideas are represented as small texts (usually not more than 100 words) and are collected in a list box. To each idea textual comments can be attached and are displayed in a separate box. In many cases, graphical explanations are more expressive than texts, thus we integrated a sketch component. Once a user has selected an idea, he can open the drawing area via a button, i.e. the integration mode is *anchored*. Solely the drawing component is provided, the user has no choice to use another component for graphical annotation, i.e. the kind of component is specified at compile time (*static* integration). If two users select the same idea, they can draw collaboratively, thus the integration mode is *public* and *shared*.

These three examples demonstrate the benefits of TeamComponents concept for developing collaborative applications.

6 Conclusion and future work

The TeamComponent concept allows to efficiently develop synchronous groupware applications with the help of software components. Each component is responsible for a specific artefact, offers its own user interface and has to manage its internal data. Using components helps to divide a software project into well defined parts. Well documented interfaces help to reduce integration efforts and improve software quality. TeamComponents can be

developed separately from an application, thus the component developer and the application developer can be different people. A set of classes and interfaces guarantees the proper integration of components into an application, even if the corresponding component source codes are not available.

Compared to other approaches, the TeamComponent approach covers a wide range of collaborative scenarios. TeamComponents can either be used in compound documents or as a static parts of a collaborative application. TeamComponents can be tailored via the criteria of integration and isolation. Especially the dynamic anchored usage of components is a powerful tool for implementing complex applications. The strength is demonstrated by the annotation feature of our Web browser DreamView. Mixing Web page contents with arbitrary, embedded annotations linked to arbitrary page items would be very difficult to realise with other component concepts.

TeamComponents have built-in collaboration services, including communication infrastructure, floor control support, concurrency control and latecomer services. Group awareness widgets such as distributed mouse pointers can easily be used inside components. Additional group awareness functions are automatically offered by the runtime system.

Currently it is not possible to build a TeamComponent with the help of other TeamComponents. We think about lifting this restriction which would allow to create hierarchical components. Such an extension would have two major advantages: first, even very complex components could be developed efficiently, and secondly, the process for developing applications and components could be unified. Applications could be developed by developing a component and applying a generic wrapper.

Managing shared states in data objects can be done with the help of transaction, monitor and explicit locks. To develop shared data object much easier, our research group is currently working on a shared data layer. Self-replicating objects with adjustable concurrency control policies would dramatically decrease the development efforts for new collaborative components and applications.

The approach has been verified with the help of sample components and applications. Our next step will be to evaluate the concept with a group of students: as part of a software project course, two teams of students have to develop a collaborative presentation tool with the help of the DreamTeam and TeamComponent environment.

References

[1] Dykstra-Erikson E.; Curbov D., The role of user studies in the design of OpenDoc, in Proceedings of the conference on Designing interactive systems: processes, practices, methods, and techniques, (DIS '97), 1997, 111-120

[2] Edlich, S., Software Cooperation with the Share-Kit: Influences of Semantic Levels on the Working Efficiency, Vienna Conference on Human Computer Interaction VCHCI '93, Vienna, Austria, Sept. 20-22, 1993, 225-234

[3] Graham N.; Morton C. A.; Urnes T., ClockWorks: Visual Programming of Component-Based Software Architectures, Journal of Visual Languages and Computing, Academic Press, July 1996, 175-196

[4] Grundy J., Engineering component-based, user-configurable collaborative editing systems, in Proceedings of the 7th Conference on Engineering for Human-Computer Interaction, Crete, Kluwer Academic Publisher, 1998

[5] Gutwin C.; Roseman M.; Greenberg S., A Usability Study of Awareness Widgets in a Shared Workspace Groupware System, in Proceedings of the ACM'96 Conference on Computer Supported Co-operative Work (CSCW '96), ACM Press, Nov. 1996, 258-267

[6] Hamilton G, Java Beans, Sun Microsystems, 1997

[7] ter Hofte G. H.; van der Lugt H. J., CoCoDoc: a framework for collaborative compound document editing based on OpenDoc and CORBA, in J. Rolia, J. Slonim and J. Botsford (eds): Proceedings of the IFIP/IEEE international conference on open distributed processing and distributed platforms, Toronto, Canada, Chapman & Hall, London, May 26-30, 1997

[8] Koch J. H., Entwurf und Implementierung einer Komponentenarchitektur für den Mehrbenutzereditor IRIS, Master's Thesis, University of Munich, Germany, 1997

[9] Koch M., The collaborative multi-user editor project IRIS, Technical Report TUM-I9524, University of Munich, Aug. 1995

[10] Lauwers J. C.; Lantz K. A., Collaboration awareness in support of collaboration transparency: requirements for the next generation of shared window systems, CHI '90 Conference on Human factors in computing systems, special issue of the SIGCHI Bulletin, 1990, 303-311

[11] Lukosch S.; Roth J.; Unger C., Marrying on-campus teaching to distance teaching, in Proceedings of the 19th world conference on open learning and distance education, Vienna, June 20-24,1999

[12] Microsoft Corperation, OLE2 Programmer’s Reference, Vols 1 and 2, Microsoft Press, Redmond, Wash. 1994

[13] NCSA, NCSA Habanero Homepage, http://www.ncsa.uiuc.edu/SDG/Software/Habanero/HabaneroHome. html, 1999

[14] Piersol K., A close-up of OpenDoc, Byte, Mar. 1994, 183-188

[15] Rational Software, UML Notation Guide, http://www.rational.com/uml/html/notation/ Version 1.1, Sept. 1997

[16] Roseman M.; Greenberg S., Building Real-Time Groupware with GroupKit, A Groupware Toolkit, ACM Transactions on Computer-Human Interaction, Vol. 3, No. 1, Mar. 1996, 66-106

[17] Roseman M.; Greenberg S., Symplifying Component Development in an Integrated Groupware Environment, in Proceedings of the ACM Symposium on User Interface Software and Technology, Banff, Alberta, Canada, Oct. 1997, 65 -72,

[18] Roth J., How to write shared applications with DreamTeam, Technical Reference, Fernuniversität Hagen, Germany, Jul. 1999

[19] Roth J.; Unger C., DreamTeam - a Synchronous CSCW Environment for Distance Education, in Proceedings of the ED-MEDIA/ED-TELECOM’98, Freiburg, Germany, Jun. 98, 1185-1190

[20] Roth J.; Unger C., DreamTeam - a platform for synchronous collaborative applications, in Th. Herrmann, K. Just-Hahn (eds): Groupware und organisatorische Innovation (D-CSCW'98), B. G. Teubner Stuttgart, 1998, 153-165

[21] Sasse M. A.; Handley M. J.; Ismail N. M., Coping with Complexity and Interference: Design Issues in Multimedia Conferencing Systems, Design Issues in CSCW, Springer, 1994, 179-195

[22] Schuckmann C.; Kirchner L.; Schümmer J.; Haake J.M., Designing Object-Oriented Synchronous Groupware With COAST, in Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW ‘96), ACM Press, Nov. 1996, 30-38

[23] Streitz N.A.; Geißler J.; Haake J.M.; Hol J. (1994), DOLPHIN: Integrated Meeting Support across LiveBoards, Local and Remote Desktop Environments, in Proceedings of the ACM Conference on Computer Supported Co-operative Work (CSCW '94), Chapel Hill, North Carolina, Oct. 22-26, 1994, 345-358

[24] Sun Microsystems, Java Object Serialization Specification, 1997

[25] Sun Microsystems, JavaSoft Home Page, http://java.sun.com, 1999