Using Component Technology for Group Editors —
The Iris Group Editor Environment

Michael Koch, Jürgen Koch
Institut für Informatik, Technische Universität München, Germany
{kochm|kochj}@informatik.tu-muenchen.de

Abstract
Component technology and framework concepts are seen as the solution to typical engineering problems that emerge when building groupware. In this paper we introduce and discuss group editors as one example for the need for component based design in the groupware application domain. We first outline the requirements for group editor applications and show how these requirements could be solved by applying framework concepts. Then we present the architecture of our group editor environment IRIS and describe the basic architecture concepts and the user interface component infrastructure ICI.

1. Introduction
The collaborative editing of documents by teams is a very common task nowadays. Writing groups are often distributed over many locations because of the globalization of organizations and the increasing interdisciplinary nature of tasks. Since many writers already use computers for their jobs, providing computer support for the collaborative writing process has been identified as an important goal of CSCW research.

The design of a group editor also represents a very good example of the need for component based design in the groupware application domain. In this paper we first outline the requirements for group editor applications and show how these requirements could be solved through the application of framework concepts (Section 2). Then we present the architecture of our group editor environment IRIS and describe the basic architectural concepts and the user interface component infrastructure ICI.

2. Collaborative writing and group editors
In general, collaborative writing involves two or more people working together to produce a document. It involves phases of writing and phases of communication. It also involves periods of synchronous activity, where the group works together at the same time, and periods of asynchronous activity, where group members work at different times. Collaborative groups of writers can adopt varying writing strategies [11]. These strategies may change during a single writing project.

2.1. Requirements for group editors
A tool for supporting collaborative writing has to face a number of challenges. It has to support different writing strategies and therefore has to satisfy different needs in information and functionality which may change during the writing project.

Additionally, a group editor has to offer the co-authors themselves as much flexibility as possible. There must not be any constraints on the work of an author. More precisely one can say that any collaborative writing software must support the writers’ normal working practices (see [6] and [12]). If that requirement is ignored, the usage of the group editor will not achieve the critical mass needed for a successful introduction [4].

A group writing tool has to support all authors in their preferred working styles and working environments. That includes the fact that different authors should be able to use different user interfaces (including standard tools or standard-tool-like user interfaces) to access the document.

To summarize, the authors need to:

- be able to use their favorite user interface or at least to be able to largely customize the standard interface to their needs,

- be able to integrate the different tools needed to accomplish the different tasks included in collaboratively writing documents, and

1 It is possible that authors have to agree on some common procedures for enabling coordination. But the software itself should not force a group member to change his environment or procedures.
be able to change the tool composition for other group setups or while the project is in progress.

Other aspects of the request for not constraining the authors in their normal work practices involve distributed system issues such as distribution, replication, consistency, concurrency and communication protocols.
The authors need to:

- be able to read and update (write) any displayed document content (no technical access restrictions, use of social protocols instead; possibility to have private areas, adjustable granularity and chooseable time for making updates public), and
- get immediate feedback on their actions (low response time).

Finally, both, the document storage component of the group editor and the user interface components should be extensible to new content types. It should be possible to handle concurrency control and replication for different content types in different ways (e.g. restricted replication for video content; also see [13]).

In summary, to design a group editor we must face different requirements addressing the document storage mechanisms and the user interface of the group editor. Main requirements in both areas (document storage and user interfaces) are extensibility and configurability.

2.2. General group editor architecture

To fulfill the requirements listed in the previous subsection a group editor cannot be designed as one single program but has to be designed as an integrated environment. A group editor has to be highly modular and dynamically changeable. The state-of-the-art solutions for the corresponding engineering problems are object-oriented concepts and component technology.

There are two layers that should be regarded separately: the storage of document objects and the user interfaces for manipulating documents.

Document storage service

The document storage service has to provide a reliable storage mechanism and mechanisms for generating and distributing information for group awareness.

It has to provide high availability and a low response time. Therefore, the document data should be replicated. This implies a storage service that is built of several storage service instances that hold the replicated data. Since we want to support disconnected operation (it should be possible to edit documents when the computer is not connected to the network) we have to use optimistic concurrency control for maintaining replication consistency.2

To provide extensibility, a storage service instance (local replica handler) has to consist of a basic component that can be uniformly accessed by the user interface components and that handles the communication among the different storage service instances, and of a number of content management components that implement the storage and replication policy for the different content types. This architectural principle (see Figure 1) is well known as the object framework approach ([8, p221f]).

In summary, to design a group editor we must face different requirements addressing the document storage mechanisms and the user interface of the group editor. Main requirements in both areas (document storage and user interfaces) are extensibility and configurability.


3 For more information on the need for awareness (information) in computer support for group work see for example [2, 3, 7, 9, 10].

Figure 1: Storage service framework

Finally, the storage service of a group editor has to produce and distribute awareness information. This information is needed for the implicit coordination among the co-authors and can be partly generated from the interaction of the co-authors with the document3.

Building storage services that provide these features is a big challenge for the database community. In contrast to existing distribut ed database systems and RPC-based transaction services the group editor storage service should use replication and lazy consistency protocols (see [1] and [15] for systems that tackle these requirements).

Group editor user interface

The user interface of a group editor has to offer different features:

- editing of the document content,
- navigation within the document,
- display of awareness information,
- communication among co-authors, and
- import and export of data from external applications.
Different users should be able to choose any of these features in any of several implementations and combine them in one application that presents itself in one window. Therefore, the different features should be encapsulated in components. In our model these components are called 'tools'. The tools should be replaceable and configurable according to the group and user needs. It should even be possible to integrate standard editor tools into the framework. Users should be able to start with an editor environment where existing single-user applications are combined with collaborative platform features. Then, the environment can evolve, component-by-component, towards a groupware application environment that consists of groupware components that support more media types, enhanced awareness support and more collaboration features.

This once again can be solved by a framework approach. The framework provides a window and integrates different tools in this window. Examples for such user interface frameworks are OpenDoc and OLE.

3. Modular design in the group editor environment IRIS

In the project IRIS we are working on providing support for group writing for several years. After tackling different aspects from concurrency control to awareness information in wide area networks we began in 1996 focusing on migrating IRIS to an extensible configurable environment according to the ideas presented in the previous section. In this section we will briefly present the new IRIS architecture. We will especially focus on the ICI (IRIS Component Infrastructure) that resulted from our efforts in building a component framework for the IRIS user interface layer. More details can be found in [7], [5] and on the IRIS-web site4.

Our current prototype implementation is pure Java (JDK 1.1). The new version including the framework concepts will be available to the public in September 1997 (an older version is already available via the IRIS-web site).

3.1. Storage and awareness service

According to the general architecture described in the previous section we divided IRIS into a storage and awareness service and different applications presenting parts of the document to the user for editing and providing support for communication and awareness (user interface layer).

The storage and awareness service is responsible for providing the document data and information on the session, on users and on hosts which are involved in the session (awareness information). The storage service is replicated on all workstations running IRIS applications. For concurrency control an optimistic approach is used (object versioning and dynamic update propagation).

In IRIS a document is a set of objects (text, graphic, video, bibliographic references, ...). The objects can be grouped in different types of document structures. Examples for document structures are hierarchical document structures or hypertext graph structures. The granularity of objects can be determined by the user (applications): It is possible to store each paragraph of a text document in a separate object, group these paragraphs to subsections, the subsections to sections and the sections to a document (hierarchical document structure). It is also possible to store whole subsections in one content object. The dynamic document granularity can be utilized for conflict avoidance.

To support a user interface defined coupling policy the storage service only distributes changes explicitly committed by the user interfaces. The communication among user interfaces and storage service is mainly handled by get and put methods: The user interface loads a document object using a get method. It then caches the object content and performs any number of updates before committing the new content by using a put method. This commit action can be performed whenever a character has been changed or after the user has been working several hours on the content. Because the storage service does not know of the change until the put method is called, the change is not previously distributed to other storage service instances and to the corresponding user interfaces.5

The storage service architecture follows the model presented in Figure 1: We have a generic storage framework and a basic storage and multicast service that handle the communication with the clients and provide basic features for content handling. For different content types specialized modules can be installed which provide content type specific access methods and which can define new storage and replication policies.

3.2. User interface layer

To access document data the user interface applications connect to the local access layer. The basic features of the IRIS user interface layer are:

- A user interface application consists of different 'tools'6 that present themselves to the user in one or

4http://www11.informatik.tu-muenchen.de/proj/iris/

5To further restrict the distribution the storage service supports 'private versions' (see [6, 7] for more details).

6A tool is an application that fulfills a specific task in the context of collaborative document editing. A tool instance is related to exactly one
more windows or in an applet area in a Web browser.

- The tool composition and the relation of tools and windows is changeable during run-time.
- The tools are highly configurable.
- The tools combined in one application are able to communicate with each other (inter-tool communication).
- Communication among applications is handled by the storage layer (notifications and awareness attributes).

Before we discuss some of these issues in more detail we want to present an example (see Figure 2):

![Iris user interface application](image)

The application consists of a navigation area that displays the document structure and additional awareness information (color coded tree nodes for visualizing information on working areas and change history). Beneath the tree widget, a document part is presented in an editing area and at the bottom of the main window a list of the authors is displayed in a user list tool. Additional windows are available on demand for displaying the history of document sub-trees and viewing or editing attributes of document objects.

For building IRIS user interface applications we developed a component infrastructure framework named ICI (IRIS Component Infrastructure). ICI provides features for grouping tools in one or more windows and even in applet panels, for inter-tool communication and more (see next section).

Using the ICI interfaces we implemented many tools that can be dynamically combined to form a group editor environment:

- **edit document content**: We have tools for viewing and editing single content objects of different media types and a tool for editing a linearized sub-tree of an hierarchically structured document. The latter tool implements a compound document framework and supports different content types. Additionally, we implemented a wrapper tool that supports the editing of text objects or sub-trees of text objects with external editor applications. Finally, we have tools for version control and for conflict notification and conflict resolution (merge tools).

- **navigation in the document**: Here we have some tools for displaying document structures enriched with awareness information. One example is the tree display shown in Figure 2. Other tools are a folder display for hierarchical document structures and a graph view for hypertext structures.

- **display of awareness information**: Awareness information is displayed in the editor and navigation tools. Additionally, there are two tools for displaying user and host information (one possible appearance of the user information tool is shown in Figure 2).

4. Iris Component Infrastructure

4.1. Overview

The ICI consists of two different kinds of components:

- **Tools** which implement the functionality needed by the users, and
- **services** which supply numerous methods to be used by tools and other services.

All these components are linked by the ToolConnector object (see Figure 3).

Tools do not have to know anything about the environment they are running in: They do not have references to other tools currently connected to the ICI nor do they know where their user interfaces are located (window or applet). They only know the ToolConnector object which they may use to call ICI service methods.

All tools must implement the following methods:
Figure 3: Basic structure of the ICI

- initialize() which contains all initialization code including the building of the tool’s user interfaces,

- toolMain() which contains the tool’s main program (if necessary) and

- terminate() which does all cleanup work.

These methods contain all code responsible for the tool’s behavior and are called by the ToolConnector object while tool connection (initialize and toolMain) respectively tool disconnection (terminate). Tool connection and disconnection are invoked by user actions, e.g. selecting a menu item, and are exclusively handled by the ToolConnector object.

While tools can be exchanged dynamically, services are coupled much more tightly to the ToolConnector object: The ToolConnector object instantiates and initializes its service objects only at startup time and removes them when shutting down. In this sense ICI services are very similar to shared libraries.

Currently implemented services cover for example user interface management (UI Service), property management (Properties Service), an API to the storage layer mentioned in the previous section (Storage Service), and so on.

4.2. UI Model

The tools’ user interfaces are organized in so-called Workspaces. Users can place workspaces in windows or applets and change their positions dynamically during run-time.

The ICI supplies special windows and applets in which to store all workspaces. These are called ToolShells. Each ToolShell contains one or more workspaces and some default menus that are handled internally by the ICI. Default menus cover opening and closing documents, adding and removing tools, invoking communication applications, setting user preferences (colors, positions of workspaces), and so on. Of course, tools can add their own menus to the default menus supplied by each ToolShell.

For an example see Figure 2: There we have four tools, each tool managing one workspace. Three of them (a simple text editor, a tool presenting some document’s structure and some user list tool) share the same ToolShell (a window) while the fourth (a history viewer) has its own ToolShell (again a window).

Tools paint their user interfaces in their workspaces, but the user decides where to put the tools’ workspaces. The tools do not even know where their workspaces are currently located. This is handled by the UI Service which also cares for the user preferences related to colors, fonts, etc. and for the persistent storage of all user preferences related to the Workspace composition.

4.3. Communication and Cooperation

All communication among tools linked to the same ICI is based on events. To send an event, the tool builds an event object and transmits it to the ToolConnector object which delivers the event. The ToolConnector object can handle unicast and multicast events.

Notifications Notifications are multicast events which can be sent by any object connected to the ICI. We distinguish between ICIEvents e.g. SystemEvents, UIEvents, etc., which are sent by services or the ToolConnector object and ToolEvents which are used by tools to notify other tools.

For example, imagine a user who selects a content node in the structure editor tool. Now, the structure editor sends a notification to all content editor tools connected which in turn display the content of the currently selected node.

Commands Commands are sent as unicast events. This makes it necessary for a tool to address some “target” tool to send the command to.

Each tool has its own globally unique identifier. Since no tool knows references to the other tools connected, tools use these ids to address other tools. Again the ToolConnector object can be used to get the ids of all tools currently connected to the ICI. The ToolConnector object also supplies some methods which return only ids of those tools capable of handling the given event and command types. This can be used as a simple filter mechanism.

Now a tool can set the command’s target to one of these ids. The ToolConnector object delivers an event (containing the command) to the given target tool which in turn executes the command. Obviously ICI commands are very similar to OpenDoc’s Semantic Events.
5. Summary and Future Work

In this paper we presented group editors as one example of the CSCW application domain. We highlighted the fact that there are different user needs that can be summarized in the need for a modular, extensible and configurable system.

For building group editors we proposed two layers, each layer built as a component framework. The storage layer provides document access. It has a common interface for the user interface applications and is extensible for new content media types. The user interface layer consists of a set of tools that can be dynamically arranged in an application framework. Some of the tools can be wrappers for external applications.

We think that these concepts are applicable for many groupware application domains. Work on object oriented groupware platforms should focus on these two areas: extensible configurable storage services that can be used by different application areas and application frameworks that can be used to build different sorts of groupware environments.

Future work on object oriented groupware in our group will be the further development of IRIS and its user interface component architecture (ICI) and the application of ICI in other groupware application areas.

Another area of future work will be the integration of CORBA technology into IRIS. The ORB technology might easily be used in the user interface - storage layer interface (we use a self-made socket-based RPC protocol now) and in the frameworks in the storage layer and the user interface layer. The problem with using CORBA concepts for building the replicated storage service is that CORBA does not currently provide multipoint communication. Here we might make use of CORBA extensions as the multicast ORB introduced by Hofte and Lugt [14].

References


ECSCW’97 OOGP workshop