A Framework for Developing Component-based Co-operative Applications

Dem Fachbereich Informatik
der Technischen Universität Darmstadt
zur Erlangung des akademischen Grades eines
Doktor-Ingenieurs (Dr.-Ing.)
vorgelegte

Dissertation

von
Diplom-Informatiker

Daniel A. Tietze

aus Tübingen

Referent: Prof. Dr. Ralf Steinmetz
Koreferent: Prof. Atul Prakash, Ph.D.

Tag der Einreichung: 18. Dezember 2000
Tag der mündlichen Prüfung: 09. Februar 2001

Darmstadt
D17

Contents

List of Figures

List of Tables

Chapter 1

Introduction

An increasing portion of people's tasks within today's modern work and office environments is no longer performed individually, by a single employee working alone on his or her task until completion, but performed collaboratively: A group of people needs to co-operate closely in order to successfully complete the task at hand. This trend is driven partly by the increased complexity of the individual tasks and partly by recent trends towards new management structures and paradigms of work (work in distributed teams, more team-oriented work practices e.g. in software development and other creative activities). Another influencing factor for this need to co-operate is the advent of "virtual organizations" or "extended enterprises" (c.f. [Gor99]): task-driven structures spanning several organizational units within a company or even between companies, which are potentially distributed on a global scale, formed with the specific goal of producing a certain product or performing a specific project.

CSCW (Computer-Supported Co-operative Work) applications enable such groups of users (usually distributed over time and/or space) to co-operatively solve a common task by co-operating on a set of shared artifacts, such as a common document, a graphical draft, construction model, etc., on which the co-operation takes place. The research area of CSCW is concerned with the use of computers in group settings. The computer-based tools used to support such collaborative work are often referred to as *groupware*. Coleman's definition of the use of groupware is that "groupware supports the efforts of teams and other paradigms that require people to work together, even though they may not actually be together, either in time or space." ([Col97], p.1). Greenberg, in [Gre91], defines the term "groupware" as follows:

> Groupware is software that supports and augments group work. It is a technically-oriented label meant to differentiate "group-oriented" products, explicitly designed to assist groups of people working together, from "single-user" products that help people pursue only their isolated tasks.

The aim of CSCW systems is to aid not only the co-operative tackling of the task at hand but also the communication between the group members as well as the co-ordination of the tasks performed individually or in a group. It does this by providing the users with *collaborative applications*. [DCS94] broadly defines a collaborative application in this way: "A collaborative application is

a software application that (a) interacts with multiple users, that is, receives input from multiple users and displays output to multiple users, and (b) couples these users, that is, allows one user's input to influence the output displayed to another user." This general definition also holds for the collaborative systems presented in this thesis.

It is widely accepted that successful groupware applications need to provide support for what has been termed the "three Cs of workgroup computing" ([The01]):

- communication: enable exchange of e.g. ideas and notes in the course of the work;
- coordination: allow structured flow of artifacts between tasks, as well as a means to schedule and control complex cooperative activities;
- co-operation: provide tools for joint work on a common artifact.

Before proceeding, it is necessary to define a number of terms used in the course of this thesis.

> The **setting of collaborative work** (or collaborative setting) is understood to be comprised of a group of users at one or more locations using a set of tools, jointly manipulating shared objects at the same time or at different times, to work together towards a common goal. The users' work environments and the tools used to perform the work need not necessarily be the same for all users. Variation between different settings of collaborative work can be along any of the aforementioned dimensions, i.e. the number of users involved, the time bounds placed on interaction, the locations of the participants, the set of tools used by the users and the set of shared objects on which collaboration is taking place.
>
> The set of all tools used in a collaborative setting, plus the architectural elements such as client and server applications will often be referred to as the **groupware system**.
>
> The shared objects on which the collaboration is based are said to form the **artifact of work** of the group.

Increasingly, the users' work processes are becoming more flexible but also more complex. As the content of their work evolves over time, the users require new and different tools to be used in the changing work settings. Also, users are becoming increasingly mobile, performing their collaborative activities in changing environments, with changing groups and on various platforms, ranging from stationary office PCs over computer-supported meeting rooms to hand-held and mobile devices. In order to support such changing settings, groupware needs to be able to evolve along with the work processes and be adaptable (or be able to adapt) to new collaborative settings.

In the light of this growing demand for supporting flexible and evolving work processes, Henri ter Hofte (in [tH98]) cites two main requirements for modern groupware environments: *extensibility* and *composability*. These, he defines as follows ([tH98], p.54):

Extensibility is the property of a system that denotes how easy new functions can be added to the system, without interference with existing functions.

Composability is the property of a system that denotes how easy the function of a system can be composed by selecting and combining more basic component functions.

Regardless of the co-operative application's semantics (e.g. supporting a document-based metaphor which gives the users the idea of co-operating on a common document or providing a shared communication medium such as a bulletin board or discussion database), general provisions within the collaborative tools have to be made to enable sharing of data items and management of the concurrent access which takes place when several users modify a certain shared object concurrently. This is especially important in *synchronous* groupware tools, which enable a group of users to simultaneously work on a common document, where each user is able to perceive the modifications performed by the other users as they occur in what is perceived to be real-time. This is in contrast to *asynchronous* groupware, where one user's actions do not directly affect another user's actions taking place at the same time[1].

The development of groupware is supported by groupware development tools and frameworks, such as the COAST toolkit [SKSH96], Disciple [Mar99], GroupKit ([RG96]), DistView ([PS94]), etc., which relieve the groupware developer of a number of such technical issues. Using such construction frameworks, groupware is built like most applications: The framework is reused in multiple groupware applications, but these applications remain static and closed and thus not easily extensible or adaptable. Typically, a dedicated (application-specific) server is installed somewhere and specific applications are installed on the users' machines. When the needs of the group change, because they need to work in a different collaborative setting or the format of the artifacts of work change, new applications need to be developed and installed on the users' machines (sometimes even necessitating a change of the server). Such groupware applications are not well suited to address the challenges of flexible and evolving collaborative settings, as will be shown in this thesis in the overview over the state of the art in groupware development support.

In current software engineering literature and practice, there is a trend towards software (development) environments based on the notion of reusable, composable building blocks: Components [MN98]. In [Szy97], a component is defined as follows: "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties." Component architectures allow the flexible assembly (either by developers or by end-users) of components into suitable applications. Component-based development holds the promise of easier construction of complex applications and of improving software quality by fostering wider reuse of existing components. This trend has not yet fully carried over into the development of distributed groupware applications, though. These are still mostly custom-built using CSCW toolkits or frameworks which support such basic technologies as data sharing, object replication, etc. Only recently have approaches been made to support groupware development by providing reusable compo-

[1] A more detailed overview over the dimensions of different groupware situations can be found in section 3.1.

nents (see, e.g., [HKM98],[HM98], [Ise98]). As will be shown in the chapter on the state-of-the-art, such systems allow the component-based construction of collaborative applications but only partly carry the component-based approach through from the development time into the runtime. A small number of systems (see e.g. [Sti00]) allow end-users to tailor and extend their collaboration environment by accessing and composing different components for their collaboration. Current approaches do not fulfill a number of end-user requirements related to flexibility and extensibility of collaboration environments.

Developing components for use in reusable component-based environments is often done using component frameworks, or component development frameworks. A framework can be defined (as in [Joh97]) as follows: "A framework is a reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact. Another common definition is 'a framework is the skeleton of an application that can be customized by an application developer.' " Typically, a framework provides a prototypical implementation of a solution for a set of problems, which can be adapted to a specific problem by extending the framework at specific extension points, also called "hot-spots". Using development frameworks, new problem solutions can be quickly constructed, since a large part of the framework itself can be re-used.

1.1 Goal

The goal of the work presented in this thesis is to aid the use and development of flexible component-based groupware systems. To this end, a framework needs to be developed, which supports development and deployment of Groupware Components - a term which will be defined in the thesis to describe visually interactive application elements which can be fetched over the network on demand and which allow users to collaborate on shared objects. Groupware Components form the basis of groupware environments which fulfill the two guiding requirements of extensibility and composability (see above).

The development of the framework relies on the tackling of a number of sub-goals. A reusable architecture and implementation needs to be developed, on which the Groupware Components can be developed. Additionally, a development methodology is required, which will guide the development of new Groupware Components.

Synchronous and asynchronous collaboration relies on a means for communicating data and changes between the users' systems. As will be shown in chapter 3, the requirements of both synchronous as well as asynchronous collaboration can be fulfilled by providing support for object sharing. For this, a shared object model and supporting architectural modules need to be developed and made available to the component developers. A major requirement for the object sharing support is the performance, which will be defined in terms of response time to user interaction.

As a means for supporting composition, extensibility and reuse of Groupware Components, a common shared programming abstraction for component-based environments needs to be developed, which facilitates the flexible combination and reuse of components and thus the development of extensible collaboration environments. The thesis will present this programming model and demonstrate how it eases the construction of reusable Groupware Components.

In order to support end-users in interactively adapting the system to their needs by deploying new components and by combining existing components in different ways according to their current work situation, application support is required. The Groupware Components and the models underlying them need to be made accessible and manageable by end-users.

Means for integrating the resulting groupware environment into modern information and application architectures such as Intranets but also complex architectures composed of external tools will be presented.

1.2 Approach

A flexible collaboration environment can be modeled as a set of Groupware Components which can be composed, which can invoke each other and can be added to the groupware system at runtime, in order to extend it. These components support collaborative use in distributed collaborative settings and they can be fetched over the network when required. Constructing the collaboration environment from such components can increase extensibility and composability of the resulting system. Also, decomposing the application into distinct exchangeable modules aids the use of varying component configurations, tailored for various settings: Users can access different components in the course of their work and can use these components to co-operate within different team settings.

In a world of "collaboration components" - groupware components giving the users collaborative access to shared objects - dynamic distribution of tools (components) and the shared artifacts becomes of even greater importance.

The approach taken in the development of the target system is characterized by the combination of three basic building blocks, which will be presented in this thesis:

- an **object-oriented framework** for the development of Groupware Components which can be dynamically invoked over the network,
- an underlying **architecture** with supporting tools and services such as a component invocation and execution environment, a component and object repository, etc.,
- a **central programming abstraction** underlying the components, which aids combination and interaction of components and supports the flexibility of the collaboration support environment.

These three building blocks, in combination with current state-of-the-art technologies in collaboration support, help to fulfill the requirements of today's dynamic and changing collaborative work processes. A groupware development framework called DyCE (Dynamic Collaboration Environment) has been implemented and will be presented in the thesis. The potential of this approach will be demonstrated with a number of sample applications for different purposes built using DyCE.

1.3 Organization of the thesis

The remainder of this thesis is structured as follows.

A usage scenario, presented in chapter 2, will lay the foundations for the requirements which are to be fulfilled by the system to be developed. From this scenario, a list of requirements will be derived.

An analysis of the state of the art will be presented in chapter 3, in which a number of approaches and systems for the development of collaborative systems as well as component-based environments will be described in more detail. For each system, a comparison to the system requirements will be performed.

The central concept of the thesis, Groupware Components, will be presented in detail in chapter 4. In order to support the composition and reuse of these components, a central programming model will be presented. This chapter will also introduce the system architecture developed for supporting the use of Groupware Components.

Specific implementation details of the DyCE system will then be presented in chapter 5, together with a discussion of a number of implementation-specific features of the system.

The chapter on usage experiences (chapter 6) will present a number of collaborative systems realized using the DyCE system, e.g. for support of Collaborative Hypermedia as well as for support for Computer-Supported Collaborative Learning (CSCL).

In chapter 7, a discussion of the system presented in this thesis will provide a summary comparison of the system to the requirements presented in chapter 2 and with the state-of-the-art presented in chapter 3. The thesis concludes with an overview of future research work which can be based on the DyCE system.

Chapter 2

System Requirements

This chapter will present a set of requirements which are to be met by the resulting system. The rationale for the requirements will be presented based on a scenario which is presented in section 2.2.

Requirements for a system - be it a development framework, class library or a graphical development tool - which supports the development and use of interactive groupware applications can be categorized according to the two distinct target audiences: The developers of a groupware application and the end-users of the application. In order to aid comprehension, the requirements will therefore be subdivided into these two categories (Developer Requirements and End-User Requirements) and labeled accordingly.

2.1 Layers of Requirements

In order to present the requirements which need to be addressed by the system presented in this thesis, we will make use of a usage scenario consisting of a set of situations of collaborative work, i.e. scenario snippets illustrating a certain requirement.

When discussing the requirements to be fulfilled by the resulting system, we are faced with varying levels of requirements. A usage scenario can, obviously, present only the users' view of the system. When discussing Groupware, this is arguably the most important driving force behind the requirements. The remainder of this thesis will mainly focus on presenting a development support environment (development framework), though. Therefore, we are faced with the issue of deriving and later addressing requirements for the development system from requirements stated rather more indirectly, through the use of a system developed on top of this system. If we assume that use of a system leads to a set of requirements for that system (as is most certainly the case), we are faced with a more complex requirements view (see figure 2.1).

The two main constituents of the requirements definition for a development support system are (1) *the developer*, who uses the system to develop an application, and (2) *the end-user*, who uses this application. The developer's requirements towards the development support system (labeled [a] in the figure) stem, quite logically, from his use of the system to develop the application system. Several of the end-user's requirements (labeled [b]) towards the application sys-

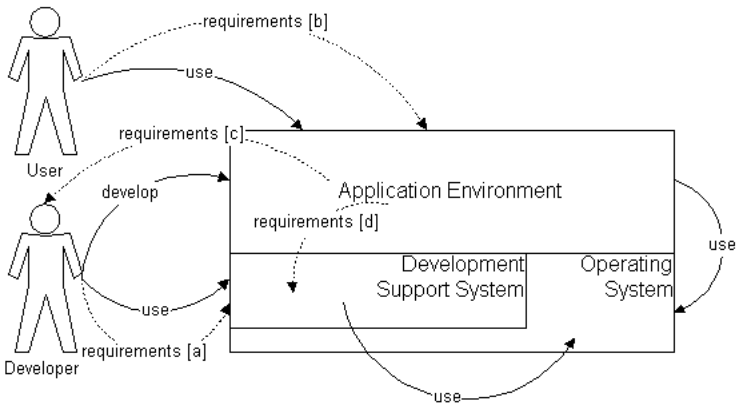

Figure 2.1: Levels of requirements

tem are to be addressed by the developer, so they result in requirements to the application developer (labeled [c]). In addition, several of these requirements result in requirements towards the development support system (labeled [d]). These are requirements which the developer can do little or nothing to address (apart from using a different development support system, perhaps).

The following usage scenario will present a number of usage situations of collaborative applications. These collaborative applications are developed on top of the development support system presented in this thesis. We are therefore mostly interested in those user requirements which can be mapped onto requirements towards the development support system.

Following the scenario situations which illustrate end-user requirements, situations which focus on the developers will be presented in order to motivate the developer requirements.

2.2 Scenario: Usage of Groupware Components

A company with branches in several distributed locations uses the corporate network to deploy a collaboration environment providing the users with the ability to form distributed teams. Using the environment, the employees can conduct their activities, closely collaborating on common document bases [1] with their team partners (including synchronous and asynchronous collaboration as well as multi-party multimedia conferencing), exchanging documents and document drafts and leaving notes and annotations for others. The collaboration environment accesses common databases which are used to persistently store the data objects on which the users are working. We will observe a number of different collaboration situations encountered by a distributed team of office workers, named Andrew, Barbara, Charles and David for easier reference.

[1] While in the major part of this thesis, the term "document" is not used to refer only to typical linear documents such as the text of this thesis, the document bases in the repositories presented in the scenario are indeed taken to contain such documents.

2.2.1 End-User Requirements

RU1: Access to shared artifacts and collaborative tools

After coming into the office in the morning, Andrew logs into the system at his workstation and is presented with his desktop environment (see Figure 2.2), including:

- a set of icons depicting shared document folders (1) to which he has access and which he uses to collaborate with his various project partners,
- a calendar tool depicting his appointments (2),
- a mail in/out-box (3) used for sending and receiving email messages,
- a set of tools (4) which he can use to perform his tasks,
- a user list: a list of collaboration partners (5) who are currently available.

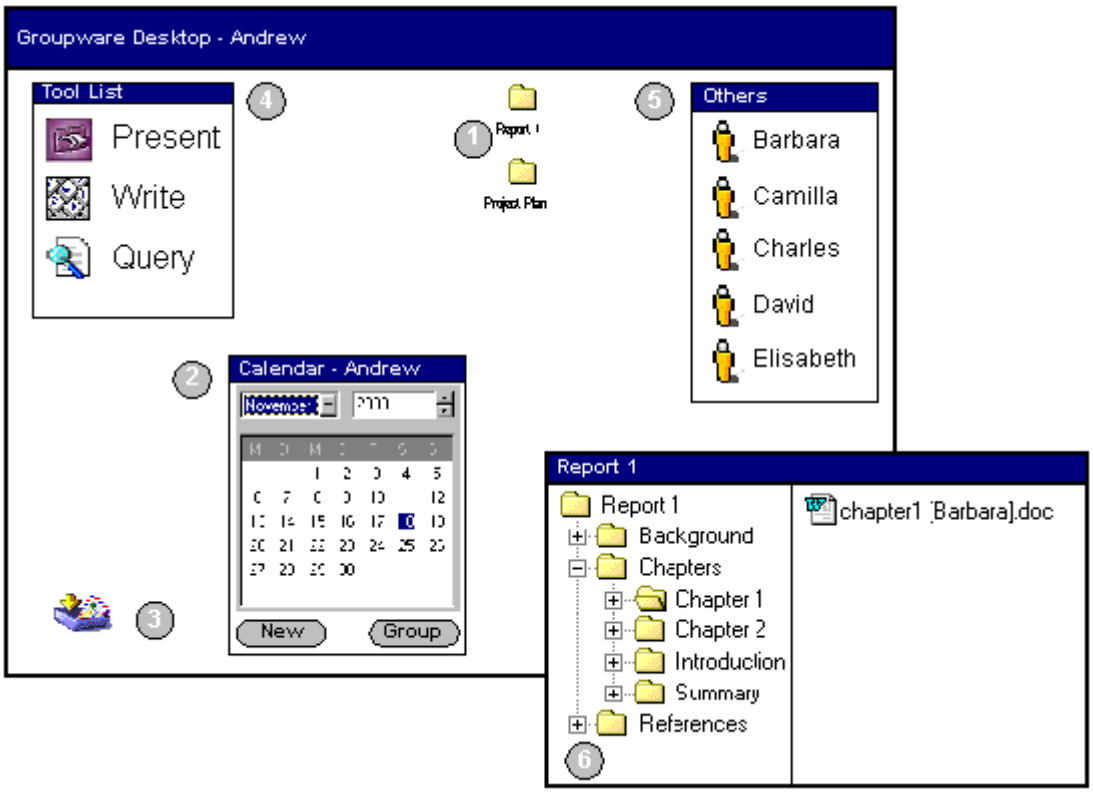

Figure 2.2: Collaboration Desktop Application

Andrew now begins working on his tasks. From the set of icons depicting shared document folders, he selects the folder for "Report 1" opens it. He is presented with a "Folder Browser" ((6) in figure 2.2) component showing the

contents of this repository: A hierarchy of folders and sub-folders containing documents, annotations, etc. Using the component palette (or menu), he accesses a group calendar component and checks which appointments are entered for the groups of which he is a member.

RU2: Computer guidance in selecting appropriate tools

The goal of the work group of which Andrew is a part is to collaboratively write a complex technical report. Each member of the group is responsible for one chapter of the report and also contributes his specific expertise in certain technical areas to other group members when required. Andrew continues working on his chapter of the document. He has already written an initial version of the chapter which is available as an icon on his desktop. To do so, he uses a function of the desktop to indicate his desire to open the document. The system knows which is the correct application for this kind of document and opens the document with the correct tool. Andrew now proceeds to work on the document.

RU3: Provision of Group Awareness

Barbara is not in her office, but instead uses her laptop computer and wireless network connection (through cellular phone) to access the system. She logs into the system and is also presented with a view of her collaboration desktop, similar to the one Andrew got after logging in. An icon representing Barbara now automatically appears on the user list on Andrew's desktop, indicating to him that Barbara is now available for collaboration. Likewise, an icon representing Andrew is displayed in the user list on Barbara's desktop.

On her desktop, Barbara has a folder representing the group's technical report, containing icons for each chapter. Upon opening this folder, she sees a small name tag next to the chapter currently being edited by Andrew, indicating to her that Andrew is in fact currently working on his chapter.

RU4: Support for synchronous and asynchronous collaboration

Looking at the contents of one of the shared folders on his desktop, Andrew discovers that a chapter of the shared report has recently been edited. This chapter has been added to by Barbara while he was out of the office. Since it is relevant to the chapter he is writing, Andrew accesses the new chapter version and looks up a certain technical definition. Again, the system provides support by invoking the appropriate tool. Andrew is not satisfied with the explanation given in the chapter and can now do a number of things:

- He can attach a note to the chapter and leave it for Barbara to see,
- he can directly contact Barbara to discuss the chapter's contents,
- he can directly rewrite the definition (potentially leaving a note about the changes he made), or
- he can contact Barbara using e-mail or the telephone to discuss more directly with her.

Since the user list on his desktop shows the availability of Barbara, Andrew chooses to contact her in order to make the changes together. In addition to opening the appropriate communication channel(s), e.g. using a voice conference or simply making a telephone call, Andrew invites Barbara into the editing

session in which he is currently viewing her chapter. Upon accepting the invitation, Barbara receives an editor tool which is shared with Andrew, the two can now co-operatively work on the document, making changes, discussing the contents, etc. Andrew also invites Barbara into the editing session on his own chapter in order to explain to her where the need for referring to her chapter arises. The group awareness information attached to the two chapters is updated to reflect that both Andrew and Barbara are now working on this chapter.

RU5: Ubiquitous access to collaboration environment

After working with Barbara on the common problem for some time, Andrew needs more in-depth information which is not accessible in his office, but for which he needs to go to the lab, which is also equipped with PCs. He therefore logs off from the office system, goes to the lab and logs in to the lab PC. After starting the general collaboration desktop application and identifying himself to the system, he is presented with his desktop contents as he left them in his office environment and can directly resume the collaboration sessions with Barbara. The two can now continue discussing and collaborating on the technical reports.

RU6: Multiple simultaneous collaboration modes and transitions between them

While working together with Andrew, Barbara uses an editing tool in a separate session, currently visible only to herself, to take additional notes. She intends to incorporate these notes into her chapter at a later point in time, but does not yet wish to share the notes taken with Andrew.

She opens an editing tool in a separate session and begins individual work in this tool, allowing her to concentrate on the technical details of a certain paragraph. Having completed the paragraph, she invites Andrew into her editing session in order to show him the result. Together, they now integrate the new paragraph into their common document.

RU7: Dynamic extensions of the collaboration environment

In the course of collaboratively working on the shared technical report, Andrew and Barbara need to incorporate more elaborate figures into the document. There is currently no suitable diagram editing tool available in the collaboration environment, but Andrew has recently downloaded one onto the lab PC. In order to use this tool in the collaborative work, Andrew loads the external tool into the collaboration environment. It automatically appears on all users' tool palettes and is directly available for collaborative use. It is not necessary for Andrew or Barbara to exit or restart the system in order for the new tool to be available to them.

Barbara now creates a diagram as provided by the new diagram editing toolkit and introduces it into the common document. Using the new shared diagramming tool, the two can now jointly create the new figures in their report.

RU8: Coupling of different tools

In order to get an overview over their current document structure, Barbara opens a hierarchical outline viewing and editing tool on their report. She activates the tool in a separate, non-shared mode but uses it to access the report she is writing together with Andrew. The outline viewer therefore displays to her a graphical overview of the current structure of their report. While the two continue their work on the report, the overview display is continuously updated and synchronized.

Using the hierarchy viewer, Barbara detects a problem with the chapter sequence of the report. She uses the hierarchy editing capability to fix this problem by rearranging the chapter sequence. Since she is working directly on the structure of their shared report, these changes are also immediately reflected in the tools she is sharing with Andrew. After discussing the changes she made, their joint work can continue.

RU9: Support for mobile work

Charles, a project manager and the third member of the distributed team, currently spends a lot of time on one of the companies campusses, walking between meetings, labs and different users' offices. In order to check on the progress of the report, he uses his hand-held wireless PDA (Personal Digital Assistant) to log into the collaborative system.

He is presented with a "stripped-down" version of the collaboration desktop, adapted to suit the restricted screen estate of the mobile device. Still, he has access to the same shared objects as Andrew and Barbara and can also perceive the group-awareness information about what is currently going on. He now uses an outline viewing tool on the common report in order to get an overview over the structure of the report and see what has changed since he last checked. He receives an indented table-of-contents view showing the current structure of the report. This structure display is a "live" display of report's table of contents. As the work on the report progresses (e.g. as additional subsections are added by Andrew and Barbara), Charles's view of the structure is always kept up-to-date.

RU10: Combination of existing tools (End-User customizability)

At several points throughout her collaboration sessions, Barbara observes the need for a number of users to combine textual and graphical data in their discussions. They often use a shared drawing tool (shared whiteboard) and a multi-user chat tool together in order to allow textual interaction (chat) to be enhanced by quick scribbles, notes and rough illustrations. The collaboration system provides individual tools for chatting and for drawing. The recurring act of setting up these two individual tools for a group of users is perceived to be arduous and repetitive.

Using facilities provided to all end-users, Barbara sets up a new tool, incorporating the Chat and Whiteboard tools and deploys this new combination of components via the server. Now, whenever the need to do such communication arises, she only needs to open the one new tool and invite her co-workers into the collaboration session. All other users will automatically receive the new component configuration, which appears to them to be a single tool providing complex compound functionality.

RU11: High system performance

While Barbara and Andrew work together in the shared whiteboard and on their shared document, they are discussing the changes they are making to the document contents. Both expect the system to behave in a way reflecting the interactive and direct nature of their collaboration. They expect changes to the document to be reflected at their partner's site instantaneously and they expect the ability for their own work to proceed as naturally as possible throughout the collaboration.

Technically speaking, the users expect low latency, fast turn-around and expect the performance loss due to synchronization of their shared work to be

as low as possible. ter Hofte (in [tH98]) defines the terms feedback (changes observable due to my own actions) and feedthrough (observable changes due to others' actions). Different collaboration setting have different requirements in terms of feedback and feedthrough. The highest demands are imposed in real time distributed collaboration settings (see [Gre98]). Here, the direct nature of the interaction between the participants requires fast system response. The better the feedback and feedthrough performance of the CSCW system is, the more direct and synchronous situations the tools built on the system can support.

2.2.2 Developer Requirements

David is a member of the in-house development staff of the company and is responsible for developing new collaboration tools as well as maintaining those tools already available. From David's work situations we can derive a number of general developer requirements for a development support system.

RD1: Reuse of existing programming knowledge and experience
In order to gain acceptance and wide-spread adoption, a system for the development of collaborative applications should introduce as many new concepts as required but as few as possible. Put differently, the developer's experience in developing single-user applications must also be applicable in the development of collaborative systems.

RD2: Reusable shared data models
The shared data models created by the developer describe the structure of the shared data items, which are viewed and edited using collaborative tools. When developing new components (or extending existing ones), it has to be possible to reuse and extend existing data models (and not be forced to develop them from scratch). This is also important if new components need to interact and coexist with existing components from a similar application domain. Support for creating and maintaining such data models needs to be provided. The developers need a way to specify the shared data models used by their components. This specification should be in a way as close to the target programming language as possible (this ties in with the previous requirement).

RD3: Transparent sharing of data objects
Unless necessary or desirable for the development task at hand, the developers should not have to be concerned with the actual distribution of the shared data objects throughout the system. The development system can be expected to provide support for the sharing and distribution.

RD4: Support for shared as well as local data objects
Not all data items, even in a collaborative application, need to be shared. Some might be local to the current machine, to the current user or just be temporary data structures related to the local graphical user interface. Simply sharing all data elements of the application would potentially incur high performance costs. The developer must be provided with the ability to make the design decision whether a data item is to be shared or purely local. This distinction needs to be supported *within* the shared data model, i.e. it must be possible to implement a data object of which parts are shared between all users while other parts are local to a machine and thus not shared.

RD5: Access to co-operation information when required

Should it be necessary for a specific component (e.g. when wanting to display a list of current users), information about the collaboration setting and the collaborative interaction needs to be available to the developer. If David wants to enhance the collaborative editing tool to provide additional mechanisms for interacting with the users currently sharing a text, he needs to have access to all the information about the current collaborative setting.

This cooperation information must also provide means to provide group awareness information in the components. Group awareness relates to providing users with an awareness of the activities and tasks of other users. Typically, group awareness includes information about the past (who has done what with which shared artifacts?), the present (who is currently logged in an what are they doing with which artifacts?), as well as - if possible - future actions of the other users.

RD6: Deployment of newly developed components

Once a new component has been developed, it has to be relatively easy to distribute it to the target users. Also, it follows that all users need to have access to newly developed components simultaneously, so that the collaboration can make use of these components at all connected sites at the same time and that the work environments of the users do not diverge. The same applies obviously to new versions or releases of existing components[2].

RD7: Technical scalability of the solution

The components developed using the toolkit have to scale in terms of the number of users, i.e. there needs to be a graceful degradation of performance as the number of users increases. This technical requirement does not apply, though, to issues of scalability of group awareness displays, which remain within the responsibility of the component developer. E.g., when developing a component including a user list, the developer is faced with the issue of how many users this list can actually display at the same time in order to still be usable. Discussions of such problems are beyond the scope of this thesis.

RD8: Support for integration into external architectures

Modern work environments often use complex distributed architectures in order to integrate the groupware environment with various other tools, potentially off-the-shelf products, or developed on other platforms. Developers require a sufficiently open interface to the collaboration, which enables them to query the current collaboration state, invoke collaborative tools, etc.

RD9: Support for server-side components

Some operations in a collaborative system are best done in a single central location, either because they require access to a centralized resource, because

[2]This requirement ties in with the recent discussion about computing environments' TCO (Total Cost of Ownership). Making new tools easy to deploy and maintain throughout the company reduces all systems' TCO. Inversely, collaboration systems not fulfilling this requirement (e.g. systems requiring new tools to be manually installed on each client system) can dramatically increase maintenance cost, since similar work environments need to be maintained for all possible collaboration groups. One user's use of a new (version of a) tool or new document format can force all users to upgrade - while at the same time still maintaining the previous versions of the tools since these could still be used in other teams or groups. A dramatic illustration of exponentially increasing TCO is the large variety of (enormous) office suites that are required on each PC in order to be compatible with all potential collaboration partners inside and outside of the organization. While an interesting sidebar to bear in mind, a solution to this problem in its entirety is obviously beyond the scope of this thesis.

they need to run continuously even without a user logged into the system or because they would otherwise be very difficult to schedule and coordinate. Examples for such components would be data imports from external sources or a workflow enactment engine which needs to keep track of time and tasks even when no user is logged into the system.

For such components, the system needs to support server-side components, i.e. components which, when invoked, run on the server machine, with direct access to all services and shared data objects. Regardless of the fact that they are executed on the server, the components need to be invoked in the same way as all other components, providing a uniform access model for components (and no specific treatment of server-side components by the users, e.g. through the use of specific use-interface features).

2.3 Overview over Requirements

An overview over the requirements presented in the scenario segments above is shown in table 2.1.

ID	Name
RU1	Access to shared artifacts and collaborative tools
RU2	Computer guidance in selecting appropriate tools
RU3	Provision of Group Awareness
RU4	Support for synchronous and asynchronous collaboration
RU5	Ubiquitous access to collaboration environment
RU6	Multiple simultaneous collaboration modes and transitions between them
RU7	Dynamic extensions of the collaboration environment
RU8	Coupling of different tools
RU9	Support for mobile work
RU10	Combination of existing tools (End-User customizability)
RU11	High system performance
RD1	Reuse of existing programming knowledge and experience
RD2	Reusable shared data models
RD3	Transparent sharing of data objects
RD4	Support for shared as well as local data objects
RD5	Access to co-operation information when required
RD6	Deployment of newly developed components
RD7	Technical scalability of the solution
RD8	Support for integration into external architectures
RD9	Support for server-side components

Table 2.1: Overview over System Requirements

Chapter 3

State of the Art

After having presented the system requirements in the previous chapter, the current state of the art is presented. The discussion of the state of the art will focus on two key areas of relevance to this thesis:

- general issues for groupware development and groupware development environments, and
- approaches focusing on support for component-based groupware environments (component-based development environments as well as component-based application environments).

The identification of deficits of existing approaches will be based on the set of requirements presented in the previous chapter.

Some other, more specific issues regarding individual technologies will be presented later on in the system design and implementation chapters.

3.1 Situations of Collaborative Work

The following sections will often refer to the specific collaboration situation(s) in which certain systems can be used. Collaboration situations, and the capabilities of systems built to support these situations, can be distinguished according to the two dimensions of distribution in time and distribution in space, as proposed by Johansen in [Joh91] (see table 3.1).

In this model, collaboration situations are distinguished regarding whether the participants are co-located at the same place or are distributed over several locations, as well as whether collaboration takes place at the same time (also referred to as synchronous collaboration) or at different points in time (also referred to as asynchronous collaboration). The examples for situations in the quadrants of the matrix give an indication of the type of collaboration taking place in this combination of parameters. A number of examples for support technologies for these types of situations are given in square brackets.

	Same Time	Different Time
Same Place	*face-to-face interaction*: co-located meeting-room or classroom situations [meeting support, group moderation, shared brainstorming]	*asynchronous interaction*: shift work, job sharing [note-passing, handover protocols]
Different Place	*synchronous distributed interaction*: distributed shared editing [application sharing, audio/video-conferencing, synchronous groupware tools]	*asynchronous distributed interaction*: workflow, draft passing [shared repositories, workflow management systems]

Table 3.1: Johansen's Same/Different Time/Place matrix

3.2 Developing Co-operative Applications

A number of technologies are available for providing collaboration support through the use of computer-based applications. These technologies can be distinguished as keeping the applications collaboration-unaware (i.e. the applications do not explicitly support the collaboration and internally appear to be single-user applications) or making the applications collaboration-aware (i.e. the applications in some way explicitly model and support the collaboration of several users).

A comprehensive overview over technologies and architectural alternatives for developing synchronous groupware is available in [Phi99]. The following sections are intended to give a brief overview in order to support the discussions in the subsequent sections.

3.2.1 Collaboration-Unaware: Application Sharing

A technique for making applications co-operative that is commercially available in a number of products is termed application sharing or window sharing. In this technique, which is most often employed to make single-user applications available to a number of users, a single instance of the application runs on one user's machine - the application server - under the control of the application-sharing package. This package is responsible for broadcasting the application's output (in modern GUI environments, its windows' contents) to all connected users, gathering all users' inputs, serializing these into one single input stream which is then passed to the application which operates as if a single user was controlling it. Examples of such a system are JVTOS [DGO+94] (available for the X-Windows environment and a number of other platforms), or Microsoft NetMeeting [Sum98] for the Microsoft Windows environments.

The advantage of this approach is that it allows available single-user applications to be used by more than one user at a time, even those applications which were not designed to be used co-operatively. In order to support this, no change to the applications is necessary. Also, only one user needs to have access

to the application software in order to run it as the application server. Apart from some form of network connection to the application server, the other users only require the application sharing client software.

One of the disadvantages of this approach is that it leads to a high network load, since all application output is transmitted in bitmap form - no knowledge of the application semantics is available which would allow the system to make optimizations. The users' input is transferred at a semantically low level, such as low-level GUI events like mouse movement, mouse click, etc. It is left to the application server to translate these events into semantically meaningful events such as selecting an option from a pull-down menu or pressing a button in a dialog window.

Also, since the application remains a single-user application, no concept of concurrency control is available other than floor passing: blocking the input from all but one of the users' systems and allowing only one user to control the application while the others are merely watching. This model of co-operative activity can only implement strict WYSIWIS (What You See Is What I See) functionality: All users are viewing the exact same window contents, all the way down to the current cursor position. No notion of individual or decoupled work is supported (RU4, RU6).

Additionally, no concept exists of a common shared artifact: The users are viewing and editing a document which is local to the application server machine. Since the application runs only on the application server and has no access to resources local to the user, there is no means to save a local copy of the edited document or to integrate other document parts resident on a client machine, e.g. in the form of an import of a previously prepared document section. Modern application-sharing packages try to overcome this deficit by accessing the local system's clipboard and utilizing it for data transfer into the application.

Additional issues and more detailed discussions regarding collaboration-transparent sharing of tools in collaborative settings can be found in [LL90], [LJLR90] and [BRS99].

3.2.2 Collaboration-Unaware: GUI Event Multiplexing

A system which takes a similar approach to application sharing but operates on a semantically higher level is GUI event multiplexing, as described in [AW96]. Here, all users are running an instance of the co-operative application and a special interface layer between the graphical user interface and the application's event handling layer takes all GUI events and broadcasts them to all connected sites which interpret the events as if they had been generated by the local user. Collaboration and co-ordination is performed by executing the same set of operations in the same order at all sites.

The communication bandwidth required for the systems' communication is nowhere near as high as that required by application sharing systems: The application is available locally and no output bitmaps need to be broadcast. Also, input multiplexing is performed on a semantic level close to the application's GUI semantics, such as "button clicked", "command event", etc.

Again, this method is only applicable for strict WYSIWIS situations, since all communicating applications need to be in the exact same GUI state in order for the incoming events to be interpreted correctly.

Since this model also has no notion of a shared document - all applications are modifying a local instance of the document according to the input events they receive - and there is often no means of transferring a sensible "event history" there is no provision for late-comers. Users who join the co-operation session after some editing tasks have already been performed have no way of "catching up" with the group by receiving the current state of the collaboratively edited document. The seemingly obvious approach of capturing the event sequence and replaying it in the late-comer's application is insufficient, since this would require the user to sequentially watch the entire groups' past activities (including those which might have been undone at a later point in time). Apart from the fact that this could take an indeterminable amount of time (thus violating the requirement for high system performance, RU11), it would also seem incredibly tedious.

Additionally, the replication model of collaboration-unaware systems using GUI event multiplexing requires that the identical applications are available at all connected sites. Even having only a slightly different version of an application at one or more of the connected sites could make collaboration impossible, since potentially not all applications understand and generate the same set of GUI events (they might even have different interaction elements). Therefore, the requirement of being able to couple multiple different tools in a shared editing session (RU8) cannot be fulfilled by these approaches. From the developers' perspective, these approaches allow only an "all-or-nothing" coupling of applications. It is not possible, e.g. to have certain areas of the application local to the user and others coupled (c.f. req. RD4).

3.2.3 Collaboration-Aware Distributed Systems

Co-operative systems typically form a distributed system, where each user has access to a locally executed application instance. All running applications are connected to a server process or interconnected and exchange information over designated communication channels.

Data Sharing in a centralized architecture

In a centralized architecture, there is a central server holding all application data. Client applications generate input events, access data values or perform data modification operations and send these to the server which executes them locally and sends the execution results back to the clients. An example of such a system is an application based on CORBA (Common Object Request Broker Architecture) [omg00], where the server provides a number of object services which are publicly accessible through an invocation interface. Instead of holding local copies of the application data, the clients generate local proxies (so-called stubs) which are responsible for invoking the server's operations. The operations required for data passing - marshaling and unmarshaling of invocation parameters and results - are performed by the underlying invocation infrastructure. In the case of CORBA, these operations are the responsibilities of the CORBA runtime system and the ORB (Object Request Broker). Method invocations, object references, method parameters and invocation results are all passed through a standardized data protocol, the Internet Inter-ORB Protocol (IIOP). All processing and serialization is performed by the server objects.

One drawback with such a strictly centralized approach is that all object accesses, including read accesses, are marshaled, passed across the network, unmarshaled and invoked on the target object by the server process. In large object-oriented systems, this can lead to a high network load and slow down the co-operating applications considerably. Especially in the case of synchronous groupware providing seemingly real-time interaction based on direct manipulation, this overhead can greatly influence the overall system performance (RU11).

A clear advantage of using a standardized distributed object system such as CORBA is that it allows the developer to specify and implement the services and data model of his application in an object-oriented manner. Different client applications, written in different programming languages, can access these services using the specified invocation interfaces and can potentially implement different sets of functionality on a common object pool.

Replicated Data Sharing Architecture

In a replicating system based on shared data, each node receives and stores the data objects which it accesses or modifies. Unlike the centralized approach, there is no single instance of each object. Instead, copies (replicas) of the required data items are made and distributed in the system. The benefit of this replication is that read accesses to objects need not be propagated through the network and invoked on only one object, thus creating a potential bottleneck when several nodes simultaneously access the same object. Instead, they can be performed on the local replicas, which is much faster because neither do the requested and transmitted data elements have to be marshaled and unmarshaled nor do the invocation parameters and results need to be passed over the network. Replication is a well known and widely researched technique, especially in the field of distributed databases, to improve performance as well as availability of data.

The special application domain of this thesis, interactive groupware, involves to a large extent displaying document contents on the users' screens and allowing the users to manipulate these data elements. The speed of displaying the data, and thus the perceived performance of the co-operative application, can be greatly improved if the data to be displayed is available locally and can be accessed on the local node. This is especially true for complex document and data structures, which typically require a large number of data accesses in order to display them completely on the user's screen.

An extra overhead incurred by replicating data objects throughout the distributed system is due to the fact that concurrent accesses to data objects need to be co-ordinated in order to keep the document in a consistent state. This is especially true for interactive collaborative applications, in which a number of users can modify a common document co-operatively and simultaneously. Not only do the users have to be prevented from making conflicting modifications to the document (thereby potentially destroying each others' work), but also changes and additions to the document need to be propagated to all sites currently displaying the document. As a reaction to these changes, the users' displays need to be updated to reflect the changed document content. In the replicated case, ideally only the document modifications need to be broadcast to all connected systems in an appropriate manner and the local replicas need to be updated. Again, the resulting re-display required for a truly interactive

co-operative application can benefit from the local availability of the data items. Replication of data elements is also a promising approach to improving system scalability (RD7), since a single-site shared data model can quickly become a performance bottleneck.

An example of a groupware development framework using this approach are the COAST system ([SKSH96], [SSS99]), DistView ([PS94]) or Habanero ([CGJP98]). COAST separates the application behaviour implementation and the shared data model. The collaborative application needs to be present at all sites. The shared data is replicated between the systems. A central server is used for persistence, object access, and thus for support for asynchronous collaboration. DistView, on the other hand, separates the collaborative application into application and interface objects, using a replication approach (exporting an application window at one site and importing it at another site) to add participants to a collaboration session. The applications built using COAST or DistView are not component-based in the sense that this thesis uses the term. These are development framework and as such is extended into specific applications by programmers using the hot-spots provided by the framework. The resulting applications are not reusable or combinable, neither COAST nor DistView provide any support for runtime extension of the collaboration system, end-user customizability or tailorability. The approaches to data manipulation and concurrency control in the replicated data sharing approaches differ. While COAST uses a transaction-based model, in combination with optimistic concurrency control allowing a user's application to proceed even if not all changes to data objects have yet been validated and broadcast, DistView employs locks to prohibit concurrent and conflicting changes to shared objects. The Habanero framework provides support for sharing Java objects between multiple applications. Shared objects are kept up-to-date through events, which are distributed between the connected systems. Habanero users interact with Habanero tools, which are made collaborative through data and event sharing. The Habanero environment is component-based in the sense that new Habanero tools (called "Hablets", for "Habanero Applets") can be added to the system and can then be used collaboratively. Since Habanero is a framework concerned only with data and event sharing, it makes no provisions for coupling different tools, for combining tools in a component-oriented way or for tool discovery by end-users.

3.2.4 Summary

Approaches using collaboration-unaware applications are very helpful when wanting to make existing applications usable in a co-operative setting. It is often useful to take this approach, since one does not have to re-implement important functionality which is already available in single-user applications. Also, users are often very familiar with their everyday single-user tools and it seems desirable to carry this know-how over into the collaborative setting. Additionally, more often than not we do not have the source code of the applications we want to use, so we have to take an approach such as application sharing to use these applications collaboratively.

Collaboration-unaware approaches have the drawback, though, of being not very well applicable to asynchronous work or work consisting of shifts of asynchronous and synchronous collaboration. This is due to the fact that, for example, the documents used in the collaboration reside on the one machine from

which the application was shared. There is often no notion of a common shared workspace from which editing and collaboration sessions can be launched. Access to shared artifacts and collaborative tools (RU1) is difficult in such settings.

Additionally, the network overhead incurred by collaboration-unaware systems is often very high, since coupling is on a very low semantic level (such as screen outputs and mouse inputs). The lack of collaboration-awareness in the applications means that interaction conflicts need to be resolved by the sharing system, often in the form of turn-taking floor control, leading to various modes of collaboration, such as "sharing" (the user's local application is being shared with other users who are allowed to take control of the application), "not sharing" (the other users are only able to observe the local changes in the application but are not allowed to perform changes), "in control" (the user currently has control over the shared application), etc. These collaboration modes and the transitions between them need to be managed by the users taking part in the collaborative session; floor passing, sharing and unsharing of applications needs to be coordinated by the users. Especially in collaboration sessions including more than two or three users, this approach can be very hard to handle. Usage experiences in the POLIWORK project ([STBT99]) have shown that especially users who are not very experienced computer users have problems controlling the collaboration, taking the floor, and generally figuring out in which collaboration mode they currently are. This problem is most certainly also increased by the fact that it *appears* to the users that they are working with their familiar office application, but due to the fact that it is being used collaboratively over application sharing, some concessions and changes to the familiar way of interacting have to be made.

While collaboration-aware approaches do not have the benefit of allowing any single-user application to be used collaboratively, they often have the advantage of addressing the demands of the collaborating group in a much better way. Also, since sharing and coupling can be done on a different granularity (especially in the data-sharing or replicating approaches), the network overhead is significantly lower than with the other approaches. Collaboration-aware approaches can better allow fulfilling the user requirements RU3 (group awareness), RU4 (synchronous and asynchronous collaboration), RU6 (multiple simultaneous collaborative modes), RU11 (high system performance) as well as the developer requirements RD4 (shared as well as local data items), RD5 (access to co-operation information).

3.3 Component-Based Architectures

Even in today's multi-window graphical environments, many collaborative applications are developed and presented to the users as monolithic, all-encompassing applications providing a large amount of functionality. Such applications can be hard to maintain and are largely inflexible, due to the fact that they contain many hard-coded interrelations and interdependencies. Additionally, the technologies used for developing such applications are not sufficient for fulfilling the requirements of flexibility and system extensibility. Component architectures such as OLE2, DCOM, OpenDOC or, more recently, JavaBeans aim at composing applications of interoperable, reusable components. Different component approaches differ in the level to which they expose the users to the fact that the

system they are using is built in a component-oriented fashion.

The collaboration environment developed and described in this thesis gives the users access to various individual (but potentially interdependent) Groupware Components, each providing a distinct functionality for a certain task in a certain way.

In order to understand the major difference between the variety of approaches to providing functionality spanning multiple applications and the system proposed and described in this thesis, it is important to discuss the distinction between (and specific features of) applications, application suites, component frameworks and the system of Groupware Components presented in this thesis. The major dimensions of distinction are the different degrees of openness, integration and extensibility provided by the various approaches.

Applications are perceived to be closed environments providing a certain functionality (or set of features)to the user. In order to extend the functionality, an application might provide the possibility of invoking (executing) another application, such as the editor being used to write this text, which allows direct invocation of the LaTeXcompiler. The unit of interaction, though, remains the single application. Calling another application (and optionally passing information in the form of files) does not constitute a tight integration. Also, this form of integration is not really extensible. Granted, instead of the LaTeXcompiler, the editor could also be configured to call, say, the Minesweeper game, but it cannot be configured to do both (much less to exchange information between the two applications). Additionally, this form of integration would not appear to the user as a single application which provides a wide range of features. Instead, the result always appears as a set of different applications (each with their own look-and-feel, etc.).

Application suites consist of a number of applications (typically from a single vendor), which are designed to be integratable in a way which makes sense to the user and provides more functionality than a single application would. One example for such an application suite is the Netscape Communicator suite, comprising (depending on the configuration, e.g. private use vs. Enterprise edition) the *Navigator* Web browser, the *Composer* HTML editor, the email tool *Messenger* and (optionally) a Calendar tool. The Navigator suite is developed in such a way that switching between the applications of the suite is done in a standard manner (via the "Communicator" menu, which is present in all the suite's tools, see figure 3.1).

Typically, application suites also enforce a standard look-and-feel over all the suite's tools and also define a standard means for exchanging information between them. To the user, the suite appears more like a single large application. Most application suites are not extensible, though. They might be restrictable, i.e. the user can decide which of the suite's tools he wishes to install, but an integration of other applications into the suite's tight-knit structure is not possible in most cases.

Component Frameworks provide a dynamically extensible basis for the integration of different tools (also from different vendors), according to the user's demands and feature requirements. One of the most widely used component frameworks is currently without doubt OLE2 (see [Cha96]). This component framework is used to integrate documents from one application into documents from another application (e.g. in order to present advanced mathematical formulas within a document produced with a word processing tool) - typically,

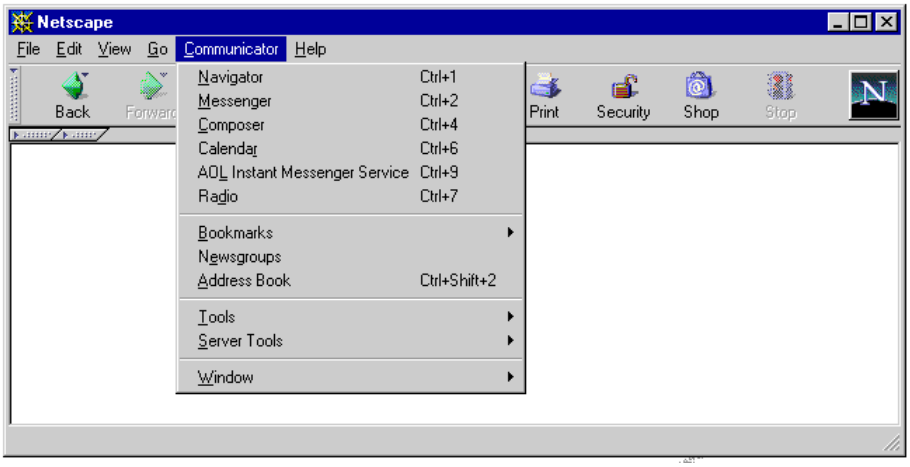

Figure 3.1: Access to applications of the Netscape Communicator suite

component inclusion is in the form of an OLE2 *container* (e.g. the word processor) which incorporates the *component* (e.g. the mathematical formula editor). Access to the available components and integration thereof is via a standard mechanism, controlled by the user (see figure 3.2).

Provided that it is developed according to the development guidelines of the OLE2 component framework, any application can serve as a container of as component. Components can even be nested hierarchically. The user can extend the system, simply by installing new applications which serve as OLE2 containers or components. These new applications can then also be used within the other OLE2-compliant applications (which need to have no additional knowledge about the new applications). The combination of OLE2 applications in a document often appears to the user to be a single powerful application. The document contents are directly bound to the application which is needed to edit the specific document content, though. If the required application is not present on the system, the included content can not be edited (in most cases, provisions are made that it can at least be viewed and printed even without the application being present on the system, e.g. by including a static preview bitmap). Also, the mapping between the included document data and the application required to edit it cannot be changed by the users. If, for example, user A receives a document containing a mathematical formula created using formula layout tool XYZ, but prefers to use the more intuitive application ABC, he is out of luck. The drawbacks of this type of component framework become even more apparent in collaborative settings, when one user's choice of components influences the work of other users in the same collaboration setting. Since there is no central place from which components can be fetched and instantiated, one user could introduce a component which the other users do not have available. This would greatly disrupt the collaboration between the users.

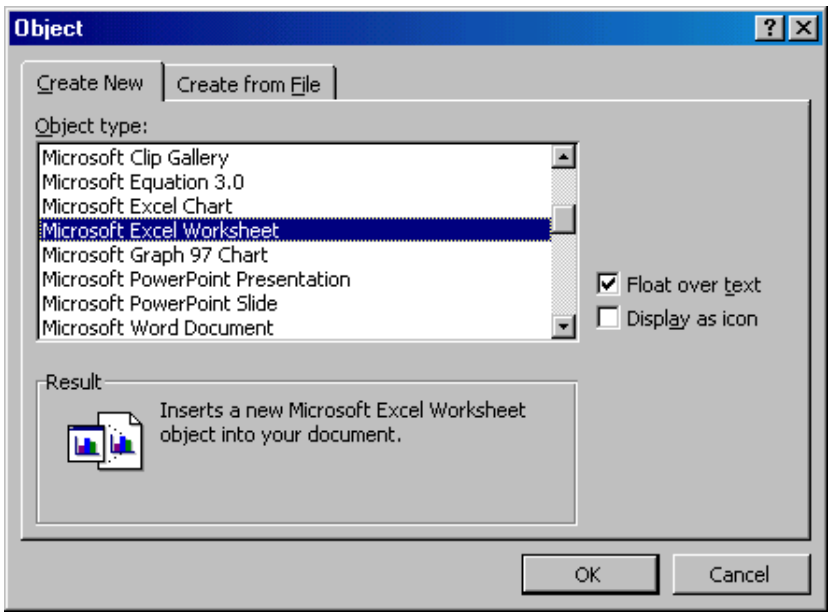

Figure 3.2: Integrating an OLE2 object in a word processor

3.3.1 COM/DCOM and CORBA

COM, the Component Object Model, is Microsoft Corporation's platform for component interconnection (see [Sai98]). COM defines the way in which client and server components interact, either within a single application process or in the form of inter-process communication. COM defines the client/service calling interface, which allows components written in different programming languages to call each other. COM is available as a standard part in all Microsoft Windows operating system platforms.

DCOM, the Distributed Component Object Model (see [HK97]) is the extension of the COM model to a distributed setting. DCOM defines the methods by which architecture components distributed over multiple machines can communicate. Interaction is done via object-oriented RPC (Remote Procedure Call). The DCOM protocol layers hide the actual on-the-wire implementation from the client components. Interaction between client and server is similar to the COM model, but the DCOM implementation ensures that the calls to the server component are adequately distributed. As [HK97] puts it, neither "the client nor the component is aware that the wire that connects them has just become a little longer." The DCOM model ensures that a component which can be used as a COM server can be transformed into a DCOM server for use in a distributed application setting.

With regard to their utilization within software architectures, COM/DCOM can be more adequately be compared to CORBA (Common Object Request Broker Architecture, see [omg00]). CORBA defines a cross-platform standard for communication between client objects and servers (services). CORBA uses an Interface Definition Language (IDL) to describe the public interfaces of a CORBA service. These public interfaces can be called through the CORBA

infrastructure from a CORBA-compliant client. By using standard CORBA services such as a naming or brokering service, client objects can discover which servers on the network offer specific services. Interface methods of these service objects can then be called in order to use the services offered by the server.

By defining the CORBA IIOP (Internet Inter-ORB Protocol), CORBA achieves true language-independence - applications whose on-the-wire communication conforms to CORBA IIOP can be developed in different languages. There are CORBA language bindings (rules which define how elements of the CORBA IDL are mapped to language-specific types and constructs) for many (mainly object-oriented) programming languages.

Infrastructures such a DCOM or CORBA can serve as the basis for distributed component-based groupware systems. Both systems define object-oriented inter-process communication in a distributed system which can be used to implement the coordination and distribution mechanisms required in a distributed groupware system.

3.3.2 JavaBeans

The JavaBeans framework [Eng97] provides a component model for reusable software components developed in the Java programming language. The JavaBeans specification specifies how a JavaBeans component can give information about itself, its properties and the events it understands (introspection). It also specifies a way in which JavaBeans components can be used in visual builder environments for developing component-based applications in an interactive manner. The JavaBeans framework is first and foremost a development framework. Users of applications built using JavaBeans are not necessarily exposed to the component-based nature of the application they are using.

The basic ability of a JavaBeans component is to expose a number of *properties* using syntactically specified "setter" and "getter" methods. By accessing the information about the available properties and the related methods, a visual builder environment can be used to wire data or events to these properties, e.g. upon clicking a button a certain input value can be transferred to a JavaBeans property (e.g. *width*) by way of the associated set method (`public void setWidth(); `).

JavaBeans components interact with other components through a publish-subscribe event model. JavaBeans components send event objects to event listener objects which have registered as event listener on the component. Inversely, JavaBeans components can attach themselves as listeners to other Java objects (such as GUI elements, etc.). A component which needs to send a certain event, `SomethingHappened`, to other components, defines an event subclass, derived from a common event base class, named `SomethingHappenedEvent`. Additionally, the component needs to provide two methods for other components to attach or detach themselves (in order to signal their interest in receiving the related events). According to the JavaBeans naming scheme, any interested components need to implement an interface `SomethingHappenedListener`, which should contain a method called when the related event occured, e.g. `public void somethingHappened (SomethingHappenedEvent ev);`. The original component needs to provide the methods
`public void addSomethingHappenedListener (SomethingHappenedListener l);` and

`public void removeSomethingHappenedListener (SomethingHappenedListener l);` . All of these interfaces can be "discovered" by the JavaBeans framework and by graphical application builders supporting JavaBeans, by a process called introspection, since the naming conventions for JavaBeans event handling are defined in the JavaBeans specification ([jav97]).

Additionally, the JavaBeans framework defines a number of methods which can cause a Bean to create instances of itself, to display an icon or display itself inside a display area as well as to accept and generate events. Using specific property sheets, JavaBeans-conformant components can by interactively manipulated.

By using these features, application developers can put together their applications from JavaBeans components. The strict JavaBeans API specification makes sure that JavaBeans components developed according to this API can coexist and interoperate within an application. JavaBeans components can be composed visually in an interactive builder tool called BeanBox (c.f. [DeS97]). In the BeanBox, JavaBeans components can be combined into applications using "visual programming": events and properties can be linked graphically in the BeanBox, events can be generated and sent to beans and a bean's properties can be set. Additionally, component-specific customization dialogs can be used to change appearance and other parameters of a JavaBeans component.

3.4 Component-Based Groupware systems

Recently, there has been some work (of which this thesis is a part) towards making component-based architectures available for supporting collaborative work.

3.4.1 DACIA

The DACIA framework for building adaptive distributed applications (see [LP00b], [LP00a]) uses a component-based approach to constructing flexible and changeable collaboration environments. By composing the distributed collaborative system of combinable PROCs (Processing and ROuting Components), complex application environments can be composed where data is gathered, manipulated and displayed at various sites. By employing component mobility, based on code mobility, the configuration of PROCs can be modified at runtime, making the system highly adaptable to e.g. changing network topologies or changing collaboration group compositions (a management interface for interactively modifying the component configurations is also provided). In this way, the resulting collaboration support system is highly configurable and adaptable, as well as extensible, at runtime.

DACIA provides a flexible development infrastructure for component-based collaborative distributed systems. In the state presented in the literature, there is no end-user support for identifying and accessing required components (RU2) or end-user combination of existing interactive components (RU10). The coupling of PROCs is based on sharing and distributing events (called *messages* in the DACIA environment). Hence, there is no support for reusable data models (RD2), or for synchronous and asynchronous collaboration (RU4).

3.4.2 Visual Component Suite

The approach described by Banavar et al in [BDMM99][1] provides a component-based development environment for synchronous groupware. Developers can use visually interactive development tools (Visual Builder Environments, such as IBM's VisualAge) to construct collaborative applications from reusable components. Application logic can be graphically specified by "wiring" the component's event sources and input methods. The available components conform to the JavaBeans component standard and are based on an event-broadcasting client/server synchronous collaboration infrastructure, Live, which handles group communication, locking, serialization, etc.

The approach of allowing groupware developers to use a graphically interactive, component-based development environment and create their groupware application in a "point-and-click" fashion seems very helpful. With the system as presented in the publication, though, it is to be anticipated that this form of interactive development will most likely quickly become very complex, especially for groupware applications which are not forms-based (e.g. a graphical editor).

In this approach, the components are only available and modifiable at development time. The components are not exposed to the users, nor can the users extend or modify the resulting seamless groupware applications (RU7). Dynamic coupling of different tools at runtime (RU8) and combining existing tools (RU10) are also not possible, since the resulting application, while built in a component-based fashion, is at runtime "closed". The approach is limited to synchronous (same-time) collaboration. Since the communication between the coupled components is done using event sharing, there is no common shared document or object model. Hence, there is no support for late-comers or for asynchronous collaboration (RU4).

3.4.3 Disciple and similar approaches

The Disciple (DIstributed System for Collaborative Information Processing and LEarning) groupware system ([Mar99], [LWM99], [WDM99]) provides a component-based collaboration system for Java, using the JavaBeans [Eng97] Framework. The goal of the Disciple system is to enable easy sharing of single-user applications. A collaboration-aware JavaBeans container allows management of collaborative sessions and allows users to extend the collaboration setting by importing additional components into the shared workspace. The Disciple shared workspace allows the invocation and collaborative use of collaboration-unaware JavaBeans components (CUAB - Collaboration-Unaware Beans). By implementing a specific Disciple API, developers can also make their JavaBeans collaboration-aware (CAB - Collaboration-Aware Beans) and provide additional functionality. Disciple is therefore a hybrid between collaboration-aware and collaboration-unaware systems.

In order to make collaboration-unaware Beans usable within collaborative sessions, Disciple employs a variant of the GUI-Event-Multiplexing approach: When fetching the Bean implementation from the server, the implementation

[1] Unfortunately, the article [BDMM99] does not give a name for the presented system. The descriptive term "Visual Component Suite" is used in this thesis merely to have a handle to reference the cited work.

bytecode is modified in order to allow intercepting of GUI events. These intercepted GUI events are later broadcast through the Disciple communication infrastructure to all users' systems in the current collaboration session, where the events are delivered to the appropriate JavaBeans. Coupling of components is strictly on the level of GUI events, conflicts are detected and resolved by a Conflict Resolver component in the system. Accesses to resources such as files are also intercepted and modified to access resource servers which are available to all users (similar to shared network drives). These resource servers also serve the components which are introduced into the shared workspace.

Specific collaboration components provide features such as concurrency control, group awareness, control of the degree of coupling, as well as telepointers and radar views. These collaboration components are always available and can be used in combination with any other Bean components being used cooperatively.

The Disciple system restricts itself to use in the Same Time / Different Place situation. The concept of persistent spaces allows suspending collaboration sessions and resuming them at a later point in time. The use of resource servers local to a specific collaborating user, as well as the lack of a common data repository inhibits the possibilities of use in asynchronous settings (RU4).

The Disciple system demonstrates a significant improvement over pure Application Sharing or GUI-Event-Multiplexing approaches: It allows flexible access to JavaBeans components within the collaborative session and allows simultaneous use of collaborating as well as single-user JavaBeans within the desktop. Still, the approach suffers from the drawbacks similar to those of other GUI-Event-Multicasting approaches, e.g. in Jasmine (see [SKFS99], [ESSGS00]), or JAMM (see [BSSS97]):

- There is no common document model which could be exchanged between the different components (RD2). Hence, all components provide collaboration on their own proprietary data structures.

- Also, as has been previously discussed, GUI-Event-Multiplexing is an approach which is only applicable to tightly-coupled, synchronous collaborations utilizing the identical components at all sites (c.f. req. RU8). All users' components necessarily need to be in exactly the same state in order for the collaboration to work correctly. The applications cannot provide, e.g., some windows which work on shared data objects and others which work on private portions of a document (RU6).

While Disciple does allow users to extend the collaboration system at runtime (fulfilling req. RU7) by importing JavaBeans components into the shared workspace, it makes no provisions for supporting the users in actually *finding* these components (RU2). The components to be used collaboratively can be fetched from any URL (Uniform Resource Locator) and it remains up to the user to find and import the components which he or she intends to use.

The work by Hummes et al ([HKM98], [HM98]) also aims at providing an extensible groupware environment by using collaborative components based on JavaBeans and focuses on the insertion of new components into running synchronous CSCW applications. Similar to Disciple, the approach to coupling is that of event sharing: GUI components at the user interface share certain events,

which can be wired in a graphical application builder in order to tailor the system or develop new components. The broadcast events are at a semantically higher level than the Disciple events, though, hence also different components can be coupled (illustrated by a teacher's and students' environment). This approach has mainly been applied to simple forms-based interaction elements. In the system described, there is no notion of a common shared persistent object space, hence the collaborative components only work in strictly synchronous settings and have limited provision for late-comers. Also, the work does not address how the components are identified and accessed at run-time.

3.4.4 Sieve

The Sieve system ([Ise98], [IBHS97]) is a collaborative workspace in which users can collaboratively create scientific visualizations, using a number of data source, processing and visualization components.

Components used collaboratively need to be developed according to the JavaBeans component standard. Similar to the Disciple system, Sieve offers a collaborative extension to the JavaBeans framework, making collaboration-unaware JavaBeans components usable collaboratively. In order to be usable collaboratively within Sieve, the JavaBeans components need to be developed using a JavaBeans feature called "bound properties". Bound properties are properties of a component which notify the component whenever they are changed. Additionally, changes to a bound property can be constrained and validated, e.g. in order to detect conflicting changes, etc. These bound properties do not fulfill the requirement of having reusable shared data models (RD2), since the properties are embedded in the components used within the Sieve visualization framework.

The Sieve component model offers a specific linking model which governs the way in which components are wired in order to produce information "flows", e.g. from a data source (signal generator), through a data processor, to a table collecting the results. The network of wired components can be dynamically extended, also at runtime. The visualizations are created any manipulated co-operatively. Sieve was initially aimed at the use in education, allowing the collaborative construction of electronic diagrams, physical simulations, etc. The Sieve workspace, which is similar to the JavaBeans BeanBox, provides a few features such as workspace awareness information (radar views) in order to support the users' coordination.

The coupling model of the collaboratively used components, based on bound properties and the propagation of change events, does not allow coupling different components (c.f. req. RU8), neither in synchronous nor in asynchronous mode. Also, there is no notion of how users can find the components which they need for their work (RU 2). It is assumed that the users are familiar with the available tool set, elements of which can be used in the collaborative visualization.

3.4.5 TeamComponents

The TeamComponents approach ([RU00]) provides collaborative components (the TeamComponents), which are based on the DreamTeam groupware development toolkit ([RU98]). The TeamComponents are specifically developed

components which can be used either as a static part of an application or dynamically inside a compound collaboratively edited document. TeamComponents support different levels of isolation (private, shared public and exclusive public) and integration (seamless and anchored). These isolation levels control in which ways a component can be used collaboratively and how it behaves while being edited. In this way, a component can be defined to be editable by a single person at a time (private), but be edited in-place, updating the other users' displays while being edited (seamless).

The DreamTeam system architecture is a replicated architecture without a central architectural component. Communication between the components occurs peer-to-peer using *multicast method invocation*, where method calls can be distributed to all connected components. Concurrency control is achieved by using different kinds of locks for the multicast method invocations.

Data is encapsulated in the components, with each component being responsible for handling its own data. Late-comers are supported by serializing a component's data at one site and sending it to the new site, so that collaboration can proceed.

The goal of the TeamComponents system is not to allow the sharing of collaboration-unaware third-party components. TeamComponents are specially developed components which are collaboration-aware to the extent that the developer even needs to care about broadcasting the correct method invocations in order to create collaborative behaviour. TeamComponents can make full use of their knowledge about the collaboration situations (RD5).

With the TeamComponents concept, users cannot collaborate using different components (RU8). The approach of multicasting methods and encapsulating the component's data in the component allows only coupling identical components. The lack of a central server and document persistence makes the TeamComponents not usable in an asynchronous setting (RU4).

3.4.6 The EVOLVE platform

Tailorability is a system property which allows end-users to adapt (tailor) their working environment so that it better suits their individual or group needs. The EVOLVE project (see [SC00]) aims at supporting end-user tailoring of running groupware systems. An empirical experiment (described in [Sti99]) showed the users' requirements for tailoring (or customizing) the tools they use in their everyday work, including those used in collaborative settings. Tailorability was shown to be an inherently co-operative activity. Skilled end-users can help less skilled end-users to perform the tailoring they require, groups of users can collaboratively tailor the system they are using together to met their group needs (preferences or special group processes).

The EVOLVE platform provides the basis for the development of component-based groupware systems which can be modified by users at runtime, making newly tailored components directly available to other users and even allowing several users to tailor their application environment collaboratively. The system maintains component compositions in CAT files (CAT stands for *Component Architecture for Tailorability*). These files contain textual representations of complex component hierarchies, which are interpreted at system startup and whenever users tailor the component compositions. The CAT files are stored on a server and are thus available to all users of the EVOLVE platform, allowing

them to share and exchange their customized components. An extension to the JavaBeans model is proposed, termed the FlexiBeans Component Model. Components developed according to the FlexiBeans Component Model are specialized JavaBeans components (and can be used as such), which provide extensions for groupware-specific functionality. In order to support their use in collaborative settings, FlexiBeans components communicate through shared variables as well as event passing. Both mechanisms work within the same machine as well as over the network, to support distributed use. [Sti00] gives a full overview over the design of the EVOLVE platform. Components provide "ports" which accept or send specific events; e.g. a GUI button might send an ActionEvent whenever it has been pressed and a component listening for such events would accept ActionEvents on a specific port. The ports of components can be linked in order to let events "flow" from one component to the next. Shared objects are provided by the components based on the Java RMI (Remote Method Invocation) distribution architecture: A component publishes a shared object as an RMI service. A shared object is seen to "belong" to that component which instantiated it. The object remains on the instantiating machine and all clients access the shared object through RMI access to the public interface of the object. The problem of concurrency on object access is addressed through the use of the `synchronized` construct in Java, which only allows one active thread at a time to pass a critical section.

End-users are provided with a graphically interactive tailoring tool, similar to the BeanBox used in the JavaBeans environment. Here, the ports of components can be connected in order to construct the application wiring logic. Changes made to the component composition are immediately reflected at all clients where an instance of the changed composition is currently used. In this way, tailoring of the collaborative system components can be done "live", while the components are actually being used.

The EVOLVE platform's support for online tailoring of a collaborative system composed of hierarchically structured components provides a very flexible groupware environment which the users can quickly adapt to their needs (RU10). The combination of shared objects and shared events as communication primitives for components of a CSCW system (an approach also taken in this thesis) supports a very logical design of a collaborative component's implementation. The choice of implementation of shared object access (publication of the object as an RMI service on one machine and access from the clients through the RPC-like RMI infrastructure) can quickly become a performance bottleneck, especially in highly interactive collaborative applications. Use of the *synchronized* Java construct to synchronize concurrent object access makes the client's thread of activity block in the remote invocation until all other accesses to the shared object have completed. To the users of data-intensive interactive applications such as shared whiteboards or other collaborative graphical applications, this synchronization will appear as very long response times, which will be perceived to be very low system performance, leading to potentially high turnaround times and slow system response in interactive settings (RU11).

The design choice of encapsulating the shared data objects in a certain component restricts the possibility of coupling different components on the same artifact (RU8) and of using shared artifacts with a variety of tools in synchronous and asynchronous settings (RU4).

3.4.7 GROOVE

A very recent development is the Groove system ([Gro00]). Groove is a peer-to-peer collaboration platform, where users can create workspaces, containing collaborative tools, into which they can invite other users with whom they wish to collaborate. Groove's extensibility and flexibility can be compared to the DISCIPLE system: Users can add collaborative tools to a shared workspace. These tools can be fetched from a central repository of tools (run, typically, by the inventors of Groove, Groove Networks, at http://www.groove.net). Users joining a shared workspace which contains tools that their system currently does not have, automatically receive copies of the tools in order to take part in the collaboration in the shared workspace.

What sets Groove apart from other collaboration systems is the approach to interconnecting the systems. The Groove approach is that of peer-to-peer computing, which has become very popular recently, with a number of peer-to-peer systems becoming widely used on the Internet. When multiple users share a workspace, the changes made to the workspace's contents are exchanged between their systems without the use of e central server. Each user's system holds a replica of the workspace contents and these workspace contents are kept consistent through command objects which are propagated between the connected peers. Queuing and caching strategies allow workspace synchronization to be performed even if the users' systems are only connected sporadically, e.g. in the case of offline use or asynchronous collaboration.

The tools which are to be used in Groove workspaces need to be specifically built using the Groove Software Development Kit (SDK). Groove provides no common shared data model for the various tools used in the shared workspaces. The medium for communication and the means for synchronization are command objects, which encapsulate changes to the data state of the tools. These command objects are created by the tools, intercepted by the Groove platform and distributed to the other replicas of the tools at the other sites where the shared workspace is being accessed. In effect, this places the restriction on the setup of the shared workspace that all users must have (the exact same versions of) all workspace tools at all times. The approach supports neither the coupling of different tools on common shared data (RU8), nor does it allow "mixed" collaborations, where different users may or may not use some or all of the same tools (e.g. depending on their roles or tasks, or which would be required in mobile situations (RU9). Also, the data model as presented in the Groove user interface appears to preclude using elements from one workspace in another workspace. Since no common shared data model is available for the tools, there is also no support for reusable shared data models (RD2). Also, the peer-to-peer approach automatically leads to lack of support for server-side components (RD9).

3.5 Summary and identification of deficits

With regard to the requirements identified in the previous section, it can be seen from the presentation that the approaches for component-based groupware systems do not fulfill the requirements as presented.

While most of the systems presented in this section (excluding the Visual

Component Suite approach by Banavar et al.) do allow users to extend the collaboration sessions with additional tools (RU7), the systems do not provide support for selecting which tools actually to use in which situations or for which documents (RU2).

Most approaches do not fulfill requirement RU4, for supporting synchronous *as well as* asynchronous work - most approaches do not adequately support asynchronous collaboration. A notable exception is GROOVE with its replication and replica synchronization concepts. The resumable collaboration sessions in Disciple appear to be an approach towards supporting this requirement. Most specifically the lack of a common server makes asynchronous collaboration difficult to coordinate between the participants. The same applies to the TeamComponents approach.

Due to the lack of a common shared data model, none of the approaches presented above support the coupling of different components (RU8). Especially in a heterogeneous collaboration infrastructure and different group compositions, fulfilling this requirement becomes very important if flexible collaboration is to be performed.

The combination of different tools (RU10) can be said to be supported by the TeamComponents system, in the form of the compound documents into which the user can embed components. These compound documents cannot be deployed and reused or standardized, though, and a new one needs to be put together each time it is needed. The easy deployment of such component configurations is perceived to be an important element of end-user tailorability of the collaboration system, though.

The developers requirements are fulfilled to varying degrees in the systems described above. Since collaboration-unaware systems such as Disciple, Sieve, etc. make collaborative use of tools not initially developed for collaboration, no developer access to collaboration information (RD5) can be supported. Non-extensible groupware development environments, such as the Visual Component Suite approach, COAST, etc. make no provisions for deployment of newly developed collaboration tools (RD6).

None of examined systems support the use of server-side components, which can be invoked in conjunction with client-side collaborative components in order to provide singular functionality (RD9).

Chapter 4

Groupware Components

The central concept introduced in this thesis is that of *Groupware Components*. Based on a schematic description of the system architecture, this chapter will present the Groupware Components and their realization in the groupware development framework DyCE (Dynamic Collaboration Environment). In order to develop component-based groupware which fulfills the requirements presented earlier, the Groupware Components need to be based on supporting technologies, including a consistent shared data model, object replication and means for providing view consistency. These technologies will be introduced after the Groupware Components.

Interconnection and combination of Groupware Components necessitates a common programming model, according to which the components need to be developed. This thesis presents a task-based programming model for the Groupware Components, providing a loose binding between related components as well as support for system extensibility. The presentation of this programming model will follow the presentation of the Groupware Components and the supporting technologies.

One of the major requirements presented in a previous chapter is that for an extensible and adaptable collaboration support system (RU7). The two phases in which changes and extensions to the collaboration support system are made is at *development time*(when qualified developers create new Groupware Components based on the development framework) and at *runtime*, when the end-users adapt the system to their needs. This chapter will therefore present the development and tailoring support provided by DyCE.

The chapter concludes with a detailed presentation of the system architecture, which refines the initial schematic view and lays the foundation for the presentation of implementation-related aspects in the next chapter.

A note about typography: In the following discussions, frequent reference will be made to elements of the DyCE framework, the development class library as well as more general concepts. Wherever necessary, concepts will by highlighted using *emphasized* font. In contrast, programming elements, classes from the DyCE framework and other programming language items which come directly from the implementation side will be presented in `typewriter` font. This typographical convention is necessary in order to distinguish between general concepts and their actual implementation.

4.1 Groupware Components - Overview

Section 3.3 has presented an overview over component-based systems which make the components accessible to the end-users in order to allow them to use their work environments in a more flexible way. The **Groupware Component** system presented in this thesis aims at providing the extensibility of component Frameworks, but with added flexibility and functionality: Leaving aside the major difference that Groupware Components are *per se* collaborative (which OLE2 and other component frameworks aren't), one major goal of the design and development of the framework was to make the resulting system extensible and to make components exchangeable (even to the point of using different components on the same document contents). This inherent extensibility is the response to the end-user requirements RU2 (computer guidance for selecting appropriate tools), RU7 (dynamic extension of the collaboration environment) and RU8 (coupling of different tools).

In such an environment, the distinctions between the components and the level to which a user can identify an application become more and more blurred. A user is no longer working *within* a single application, but rather he is applying his choice of tools to the documents or other artifacts on which he is currently working (together with others). To put it another way, instead of presenting the user with the "put-nail-in-wall-application", he is handed a tool box containing, among other things, a hammer, pliers, screwdrivers, etc. and from which he can pick the tool most appropriate for the task at hand. Additionally, existing tools can be combined to form new, more complex tools for specialized problem areas.

4.1.1 Schematic system architecture

An overview over a collaboration system based on Groupware Components is shown in figure 4.1. This schema will serve to guide the in-depth presentation of the Groupware Components in the following sections, which will systematically refine different areas of the schematic overview.

In the application schema shown in Figure 4.1, each user is provided with a client application within which Groupware Components are used to access and modify elements of a common domain data model.

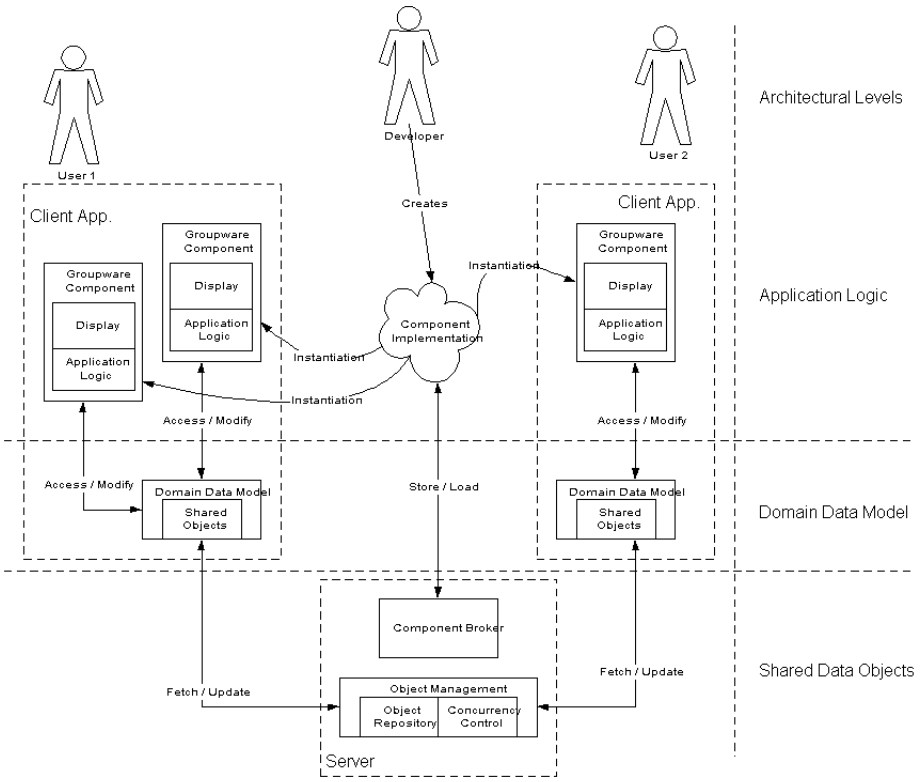

Figure 4.1: Schema of application system

Groupware Components

A *Component* is a complex element of the application system, providing access to objects through a user interface. Conceptually, a component combines a display (which presents elements of the shared domain model in a meaningful way and allows interaction) with an implementation of application logic, which is responsible for controlling the access to the shared domain data model. Components can be nested: Components can be contained in other components and can in turn contain components (this nesting relationship is not shown in figure 4.1 for clarity reasons). Additionally, several components can be coupled on one shared data item.

Components are implemented by developers, who create (program) component implementations and deploy these by storing them in a *Component Broker* subsystem accessible to all users. Upon access by the end-users, the implementations of components are fetched from the Component Broker located on the server. They are instantiated and set into relation to the data objects which they can display and modify.

User1, on the left-hand system in the diagram, is using two components to access elements of the shared domain model, whereas User2, on the right-hand system, is using only a single component. In this way, several components are sharing objects, thus providing the users with the ability to co-operatively view

and edit shared documents.

Other component architecture approaches, such as DACIA ([LP00b]), treat all elements of the system architecture as components (i.e. the object management system, the database back-end, data processing units, etc. would all be treated as components with component-specific interfaces). While this view provides a homogeneous view of component architectures, it is not important for the work presented in this thesis. While elements of the architecture and framework presented in this thesis *could* be seen as components, the subsequent discussions of the interfaces and programming models for components apply only to the *Groupware Components* as visually interactive and dynamically linkable elements of the groupware support environment.

Groupware Components are accessed through a *Client Application*, which provides users with a means to access the collaborative environment and invoke the desired Groupware Components.

Shared Objects

The basis for collaborative use of Groupware Components is realized using the data sharing approach (see section 3.2.3 for a discussion of the advantages of this approach). As shown in the diagram, components access shared objects, which form the basis of the collaboration, the shared artifacts. Components visualize the contents of these objects, allow the users to manipulate the shared objects and keep the display consistent with the state of the shared artifacts.

The shared objects represent the shared application state common to all users accessing these objects. Additional (non-shared) objects can be used by components, e.g. to represent data elements of services which are only available locally or only used by a single user (c.f. RD4).

Component Broker

Component implementations are loaded from a Component Broker, which manages a repository of component implementations where they are persistently stored and available to all connected users. The component implementations are requested from the Component Broker and loaded from the repository. Instances are then created at the clients' sites, ready for use. New components can be added to the Component Broker and thereby become available to all users (RD6, RU7).

The Component Broker can be local to the clients (e.g. as in the case of a locally installed application) or - as shown in the diagram - can be remote and accessed via the network. In the DyCE system presented in this thesis, the Component Broker is centrally available to all users. This decision eases the deployment of new components (RD6) as well as maintenance of the same versions of components in all client systems. Additionally, providing a central Component Broker supports ubiquitous access to the collaboration support environment (RU5).

Object Management

In order to support synchronous as well as asynchronous collaboration (RU4), shared objects are persistently stored in an Object Repository, which is accessed

through an Object Management subsystem. The application environment ensures that the objects are loaded from persistent storage and instantiated locally to be available to the related components when required. Between uses, the object instances - and not just the classes they are instantiated from - are stored in the Object Repository.

While objects are being used by Groupware Components, the users are potentially changing the objects. In order to enable collaboration, it is important that all shared objects remain in a globally consistent state, i.e. that changes made to a shared object at one site are reflected at all other sites sharing this object. It is the responsibility of the Object Management system to maintain this consistency. It is able to do this because all clients propagate changes performed on the shared objects to the central Object Management system. This can then employ concurrency control mechanisms in order to resolve concurrent conflicting object accesses. Changes to the shared objects which are relevant for other clients are propagated by the object management system to the affected clients. These can then update the user interface displays accordingly. More details about these mechanisms will be presented in section 4.3.

4.1.2 Characteristics of Groupware Components

The Groupware Components used as parts of the collaboration environment can be characterised as:

- Collaborative - by accessing a shared data model, which is maintained in a consistent state throughout the system, the different components provide synchronous as well as asynchronous collaborative functionality;
- Interactive - the components give the user immediate and co-operative access to elements of a shared object space and give instant feedback about the results of these operations;
- Loaded on demand - instead of being directly included in the applications accessed by the users, the components are fetched on-demand from a component repository;
- Interdependent - different components (instances of different classes) potentially access common data objects; there is potentially an intersection between the data objects accessed and manipulated by the various components;
- Dynamic - the exact point in time at which a user invokes a certain component cannot be predetermined; the same holds for the inverse: discarding of the component;
- Composable - it is possible for component developers and end-users to create new Groupware Components by composing them of other (sub-) components.

Groupware Components can be used in various ways, depending on the users' requirements. They could, among others, be used as

- components of a "groupware desktop" (as in the scenario), providing access to different features, based on the different elements of the desktop (e.g. time planner, documents, project planner, etc.)
- collaborative elements embedded in Web pages (e.g. small awareness markers indicating who is currently viewing the page, including the ability to initiate communication, interactive collaborative workspaces delivered via Web pages, etc.)
- collaborative tools on mobile and hand-held devices, providing mobile collaboration support.

4.2 Shared Objects

As has been pointed out, a Groupware Component allows a group of users to interact with (i.e. view and modify) a set of - potentially - shared objects. Groupware Components need to provide shared access to the objects which form the basis of the collaboration. In order to support collaboration, the components need to reflect the current state of the shared object.

4.2.1 Separation of application and data model

When designing a development framework for distributed co-operative systems, we are faced with varying levels of development which need to be distinguished and for which development support and framework support need to be provided: General support for *shared data objects*, support for putting together *domain data models* and support for developing the *application logic*. The separation of concerns into these levels was guided by the end-user and developer requirements presented earlier. The actual design of the different aspects outlined in this section as well as the means by which they are supported in the development framework will be presented in detail in the subsequent sections.

The levels (which can also be found in the schema in figure 4.1) relate to the varying levels of abstraction in a specific application built on a general framework; the levels also relate to the development support for Groupware Components:

Shared data objects

Characteristics: Since the framework is to be applicable to all Groupware Components and needs to provide basic support for collaborative components, we require a general model for shared objects. This general model governs how all shared data objects in the distributed system are set up and behave. Instances of the shared data object model are used in all domain-specific data model implementations (see below).

Framework support: The development framework provides the implementations of the elements of the shared data model. The set of shared data objects with which the framework deals (both at development time and at runtime) is a homogeneous set of objects conforming to this general shared object model. No application or domain-specific logic to verify external data consistency is available at this level. This level is concerned with internal data consistency:

The framework ensures that if one replica of an object is modified at one site, these modifications are propagated to all replicas.

Developer Access: Developers of Groupware Components do not extend shared objects at this level. The implementations of the general shared object model are used when putting together implementations of the next higher level.

Domain data model

Characteristics: At the domain data model level, domain-specific functionality related to the shared data objects is provided. The objects at this level contain all domain-specific functionality required to maintain externally (semantically) consistent data items. An instance of the domain-specific data model can be used in a variety of different components (e.g. instances of a domain-specific Bank-Account object could be used in a collaborative accounting system as well as a shared personal checkbook application and a collaborative Monopoly game). The consistency logic included in the domain data objects is the same across all instances of such a model class (e.g. a Bank Account class has the same consistency constraints for withdrawals and deposits across all components using instances thereof).

Framework support: The framework provides a number of base classes for classes of the domain data model. These classes need to be subclassed by the groupware developer. The framework ensures that instances of the domain-specific classes are created when shared data objects are replicated.

Developer Access: The developer subclasses the framework's base classes for the domain-specific model. In these domain-specific subclasses, elements of the general shared data model are used to put together the classes of the domain-specific data model (e.g. a domain-specific `Person` class could use a number of objects from the shared data model to model the person's name, his address and his phone number). The developer-created subclasses form the semantically meaningful interface to the shared data model. The developers also need to implement the domain-specific consistency checks in order to prevent inconsistent data states.

Application logic

Characteristics: The application logic is the actual implementation of a Groupware Component's functionality. Using the domain-specific interface provided by the domain data model classes, the Groupware Components access and modify elements of the shared data model and are notified about any changes to the shared data occurring at other sites.

Framework support: The framework provides common superclasses for the Groupware Components which provide the general logic common to all Groupware Components: tying components to objects of the domain data model (how this is done exactly will be presented in section 4.6) and notifying the components about changes in the underlying data objects.

Developer Access: The developer of a Groupware Component subclasses the framework's component base class and adds the specific functionality. By creating instances of the domain-specific data model (or having such instances assigned by the runtime system), the component becomes a collaborative Groupware Component.

4.2.2 Class and Object definitions

For the remainder of this chapter, the following definitions are relevant (the definitions are general enough to apply in all object-oriented languages).

- Class: A Class is a static programming abstraction. A class provides behaviour shared by all its instances.
- Subclass: A subclass extends a class by providing additional behaviour or overriding existing behaviour.
- Object: An object is the instantiation of a class on a specific system. If an object O1 is an instance of a single class C1, it provides the behaviour implemented in Class C1. In [Mey89], B.Meyer puts it this way: "A class is a type, an object is an instance of a type" (p.94). An object and a type are elements of the programming language.
- Shared Object: A shared object is a conceptual element of the collaboration support system. It denotes a group of objects on different systems (on different host machines or used in different application processes), which all *share* the same state. The state of a shared object on different systems is allowed to diverge due to operations performed on the shared object. It is aa task of the runtime system to ensure that the state of the shared objects converges again (see also [Dou95b]).
- Polymorphism: If an operation is defined to be valid on (instances of) Class C1, then it is assumed to be valid for all (instances of) subclasses of C1.

4.2.3 Modeling shared data objects

The lowest layer in the above decomposition of shared object models is that of shared objects. These data elements form the basic data model of any Groupware Component. The instances of this data model - the data objects - are co-operatively created, modified and deleted. Each participating system needs access to the data model specification, therefore the implementation of the basic shared object model is provided in the DyCE framework.

The aim of a generic data modeling component is to allow the developer to implement any data model required for the solution of the problem at hand, be it a single shared document, a set of documents or document fragments or any other compound object-oriented data model. For this, a set of predefined data types can be used to construct complex data structures and object relations.

In object-oriented programming languages such as Smalltalk and Java, a number of basic (or core) data types are provided by the programming language itself or the programming language's supporting class hierarchy [1]. These

[1] Strictly speaking, Java is not an object-oriented programming language but rather an object-based hybrid programming language, since the language defines a number of primitive data types such as `int`, `char`, `boolean` which are not provided as classes which can be instantiated, subclassed and extended. In contrast, Smalltalk is a pure object-oriented language since all data types are provided by classes in the language's standardized class hierarchy. These classes can be subclassed and extended to provide additional, application-specific characteristics.

basic data types can be refined using the mechanisms provided by the object-oriented programming paradigm and complex data structures can be created by combining a number of class instances.

Similar support is to be provided by the development framework: predefined data types can be combined and extended to form new elements of the data model. These new elements can then be used to construct complex data models, which can be instantiated and manipulated at run-time. Since the data objects form the basis of the collaboration activity, the data modeling facilities need to be accompanied by the appropriate support mechanisms as described in the following sections: collaboration needs to be supported by providing object replication through the distributed collaborative system; simultaneous and conflicting changes to data objects need to be detected and corrected by appropriate concurrency control mechanisms.

4.2.4 Object Representation

Developer requirement RD2, presented in chapter 2 discussed developer support for reusable data models. An important part in defining and setting up such common data models are played by the syntax and semantics expressed by an object representation. Generally speaking, the object representation mechanism describes a means to use the expressive powers offered by the system to create and manipulate new data types and to combine instances of these data types into more complex data structures. When referring to programming languages, there are two levels of object representation to consider: language representation, governed by the syntax and semantics of the programming language used, and runtime representation, created by the compiler or interpreter and maintained by the language's runtime system.

The language representation governs the way in which the programming language elements are used to create new classes, instantiate classes as new objects and so forth. For the Java programming language, a valid language representation of a class containing a number of attributes as well as methods accessing and manipulating these attributes would be:

```
public class BankAccount extends Object {
    protected float balance;
    public float giveBalance()
    {
        return balance;
    }
    public void deposit (float amount)
    {
        balance = balance + amount;
    }
}
```

The runtime representation is specified by the way in which the Java language compiler transforms this class definition into bytecode and allocates heap space for newly created instances of bank accounts.

In order to allow the application developers to benefit from the system presented in this thesis, it is important to provide an object representation facility that

- matches the general paradigm of the target programming language (in the case of this thesis this is the object-oriented paradigm),
- is sufficiently flexible and powerful to express and utilize the required concepts,
- and which maps into a runtime representation which the system can use to provide adequate replication and concurrency control mechanisms.

It is the third requirement which introduces the need to extend existing object representation schemes, since the general object representation schemes have no provision for expressing and maintaining the information required for the flexible object replication mechanisms introduced in this thesis.

Several systems created in order to aid development of distributed applications have taken the approach of extending an existing language (e.g. C++) by a number of additional syntactical elements which enable the programmer to express concepts not previously present in the programming language; others even define an entirely new programming language, e.g. in the GUIDE system [BBD+91]. Clearly, this approach has its drawbacks: The application developers need to become familiar with the new programming language concepts and applications which would previously have been portable between a number of standardized development environments suddenly, through the use of non-standard language enhancements, lose this important characteristic.

Alternately, the runtime system (and possibly the compiler) could be modified to interpret the runtime representation differently and to gather the necessary information from the representation. Again, this would break portability and generality.

Instead, a representation scheme is presented which makes use of the concepts present in the object-oriented programming language - inheritance, polymorphism, etc. - to introduce a layer of system components which can be used within the application development to express and control the additionally required functionality.

4.2.5 Specification of Object Representation

Def.: Slot
A slot is a data container consisting of its name, its type and its value [2]. A slot's data type can be a programming language type (e.g. a String) - referred to as a base type in the following sections - or a reference to an RObject type (see below), thus creating complex structures consisting of RObject instances. Slots are added at runtime, through the use of slot creation operations in the class constructors. A Slot provides a simple accessing interface for retrieving and changing the Slot's value. The framework's `Slot` implementation is never subclassed by a component developer.

The `Slot` class provides the following interface methods:

[2] The term "Slot" is borrowed, by way of COAST and its underlying frame kit, from Marvin Minsky's framework for representing knowledge (see [Min75]). The term is maintained to aid the distinction between the regular attributes of an object and the replicated data elements contained in a replicated object, which can have their own behaviour and consistency maintenance provisions.

`void setValue (Object value)`
Sets the slot value to the object passed as parameter.

`Object getValue()`
Retrieve the object currently stored in the Slot as its value.

`boolean isEmpty()`
Returns TRUE if the Slot currently does not hold a value.

`String getContentClass()`
Return the class name of the object currently stored as Slot value. Returns `null` if the Slot is empty.

`boolean holdsRObject()`
Returns TRUE if the Slot currently holds an instance of `RObject` class.

`ObjectID getContentID()`
If holdsRObject() is TRUE, returns the ID of the `RObject` instance held in the Slot. Returns null otherwise.

`addSlotChangeListener (SlotChangListener listener)`
Adds the given SlotChangeListener to the set of change listeners bound to this Slot (`listeners`).

`removeSlotChangeListener (SlotChangListener listener)`
Removes the given SlotChangeListener from the set of change listeners bound to this Slot (`listeners`).

`void changed()`
Constructs a new `SlotChangeEvent` and passes it to the slotChanged method of all SlotChangeListeners currently registered to this Slot.

Def.: RObject
A replicatable object (RObject) consists of a base object, which is the actual instance of `RObject`, and a number of slots, which contain the replicatable data part of the RObject. Each RObject is uniquely identifiable within the distributed system by its ObjectID, which is defined to be unique across all nodes of the distributed system. The RObject class provides accessing interfaces for adding Slots as well as querying the list of Slots and retrieving and changing a Slot's content. For accessing slots within the `RObject`, slots are identified with names. The framework's `RObject` implementation is never subclassed by a component developer.

The basic representation of replicated objects can be seen in Figure 4.2. The notation used in this and the subsequent object diagrams is UML (Unified Modeling Language, see [Fow00] for information about the notation; a brief overview can be found in appendix B).

The replicatable object class `RObject` provides the following interface methods:

`addSlot (Object value, String name)`
Adds a new Slot with the given name and the given initial value to this replicatable object. Creates a new `Slot` instance and adds it to the RObject's internal data structure holding the associated slots.

`addSlot (String name, Slot s)`
add a previously created slot to the replicated object's set of slots

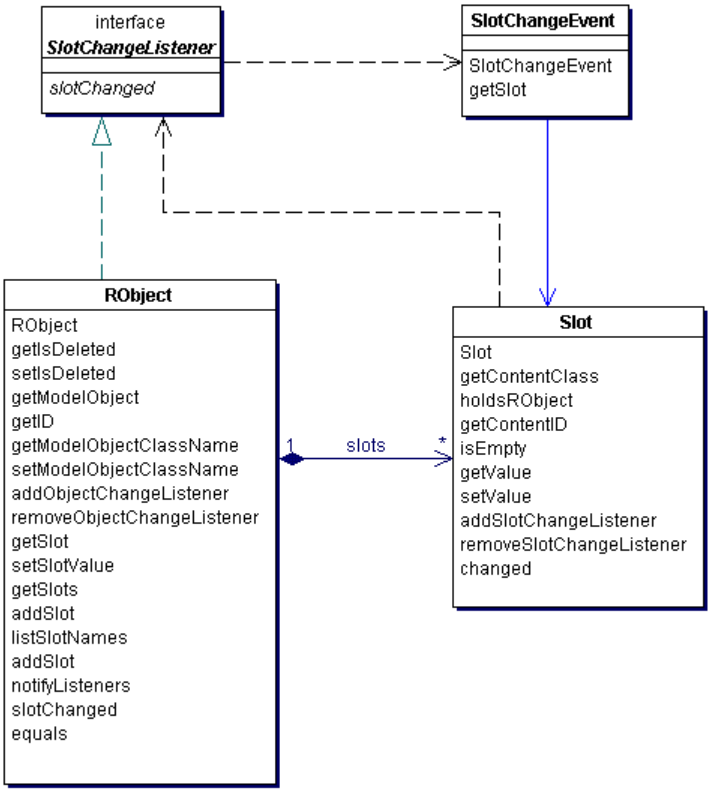

Figure 4.2: Object model class diagram

using the given name for reference. Adds the RObject instance to the new slot as SlotChangeListener.

`ModelObject getModelObject()`
Returns the domain model object to which this replicatable object is bound as a shared data object.

`void setModelObjectClassName(String name)`
Sets the class name of the domain data model instance to which this RObject instance is bound. Called when the `RObject` instance is bound to a newly create `ModelObject` subclass instance. This information is needed to reconstruct the correct `ModelObject` instance when replicating the `RObject`.

`String getModelObjectClassName()`
Returns the class name of the domain data model instance bound to this replicatable object. This information is needed to reconstruct the correct `ModelObject` instance when replicating the `RObject`.

`ObjectID getID()`
Retrieves the globally unique identifier of this replicatable object. All replicas of a replicatable object have the same ID.

`void setIsDeleted()`
Marks the replicatable object instance as deleted. The object is not directly removed, but the deletion is taken into account in subsequent replication operations and the object can be removed by the server's ObjectManager when it decides to do so.

`boolean getIsDeleted()`
Returns TRUE if the object has been previously marked as deleted.

`void addObjectChangeListener (ObjectChangeListener listener)`
Adds the given ObjectChangeListener to the set of change listeners bound to this RObject (`listeners`).

`void removeObjectChangeListener (ObjectChangeListener listener)`
Removes the given ObjectChangeListener from the set of change listeners bound to this RObject (`listeners`).

`Slot getSlot (String name) throws NoSuchFieldError`
Returns the Slot bound to this shared data object under the given name (previously added using `addSlot`). If no Slot of the given name is present, throws `NoSuchFieldError` exception.

`Object setSlotValue (String name, Object o) throws NoSuchFieldError`
Sets the value of the Slot of the given name to the given object. If no Slot of the given name is present, throws `NoSuchFieldError` exception

`Enumeration getSlots()`
Returns an enumeration containing all Slots bound to this shared data object.

`Enumeration listSlotNames()`
Returns an enumeration containing the names of all Slots currently bound to this shared data object.

`void slotChanged (SlotChangeEvent event)`
Change notification method implemented from the `SlotChangeListener` interface. Called by Slots to which this shared data object is bound as a listener, whenever the slot's value has changed. Stores the information about Slot changes locally for later use in `notifyListeners()`.

`void notifyListeners()`
Constructs a new `ObjectChangeEvent`, containing information about all slot change notifications received via the `slotChanged` method and sends this event to all registered `ObjectChangeListener`s. Clears information about Slot changes. Used by transaction management framework (see section 4.3.5) to notify dependent objects about changes when committing a transaction. Ensures that each listener is only called once per changed `RObject`, even if multiple Slots are changed within a transaction.

`boolean equals(Object o)`
Method to determine if two objects are the same. Returns TRUE if o is an instance of `RObject` and for the current object `getID().equals(o.getID())`.

Since slot creation and maintenance is performed at runtime, the type checking involved in maintaining the slot consistency is also performed at runtime as

	X.Read	X.Write
X.Read	No Conflict	Conflict
X.Write	Conflict	Conflict

Table 4.1: Table of Conflict Rules (adapted from [ÖV91], p. 285)

a task of the Object Manager. Since these type constraints cannot be checked at compile time, violation of type constraints has to be indicated to the application through the use of appropriate programming language mechanisms such as exceptions.

4.2.6 Problems with base types

Using the programming language's base data types as slot types can cause a considerable problem for the concurrency control component. If the base data type is used "as is", and instances of this type are stored and returned by reference by the slots, then an application can access the slot's value and subsequently manipulate this value using the regular programming language constructs. These modifications are then not subject to concurrency control and cannot be correctly propagated and co-ordinated.

A simple approach to overcome this problem is to restrict slot operations to simple set and get methods, where the get method returns, for base types, a copy of the slot's value. After having thus retrieved the slot value, the application can manipulate the data and at a later stage reinsert the changed value into the slot using the set method. These set and get accesses would then be subject to the concurrency control of the runtime replication system. A problem with this approach is that no optimizations could be performed by the system, e.g. by using commutativity information about the manipulating operations in order to detect non-conflicting slot accesses and, thus, non-conflicting transactions on the same set of slots. A worst-case approach would have to be taken to detect conflicts between transactions, using the simple set of conflict rules that two Reads on the same slot are not in conflict, while a Write on a slot conflicts with any other concurrent operation on the same slot (see table 4.1).

A solution to this problem is to provide "type wrappers" for the base data types, implementing a specialized invocation interface which has knowledge about the need for concurrency control and supervised access to the slot's data value. Ideally, this type wrapper interface provides the same interface as the programming language types do, thereby completely encapsulating the data object in a transparent way. One important task of these type wrappers is supervising slot accesses and propagating information about the slot value changes, along with semantic information about the slot access operation. This information can later be used by the system, together with the type wrappers, to determine whether two operations are actually in conflict and whether the entire operation (transaction) needs to be aborted due to conflicting operations.

In this approach, the type wrappers have to be specifically developed for each base data type that is to be made available as a potential slot value type. These type wrappers can be provided as part of the supporting toolkit, ready to be used by the application programmers.

As a result of a data value access, the type wrapper forwards the access to the actual data value and also provides a description of the slot access to the runtime transaction management system, which is closely related to the concurrency control mechanism. This information about data accesses is gathered and forms the informational part of a transaction (see section 4.3.5 for more details about transactions). All type wrappers provide one common interface method, which is used to determine whether two operations are commutative or not. Later, when the runtime system needs to determine whether to allow a transaction to commit, the interface methods of the type wrappers are used, by passing the previously generated information, in order to determine an order between the transactions and in order to detect conflicting operations.

4.2.7 Slot and RObject observers

Each slot and RObject can be bound with any number of observer, which are notified when a slot's value is changed. These observer relationships can be used to implement any reaction to slot value changes, such as the recalculation and refreshing of the displays, recalculation of non-shared instance variables or more elaborate actions. In order to be able to react to slot changes by means of an observer, an object needs to implement the `SlotChangeListener` or the `ObjectChangeListener` interface and implement the notification method defined in this interface. This method is automatically called by the runtime system on all currently registered observers when the slot's value is changed. Slot triggers can be used when a component wants to register interest in certain slots and react differently to changes of different slot values. Consider an RObject modeling a paragraph of text with slots representing the start and the end of a text selection as well as the current cursor position within the text. Whenever the cursor position value changes, it may be sufficient to reposition the cursor on the screen without redisplaying the entire section of text.

Additionally, observers can be attached to instances of `RObject`. By attaching an observer to an `RObject` instance, an application can register interest in the fact that the object's data set is changed, regardless of the actual slot that has changed. This can be useful for objects where any change in a slot value has to always lead to the redisplay of the entire object, when partial redisplays of an object are not feasible. Consider an object representing a complex geometrical shape consisting of a sequence of connected dots, a colour value and a position on the screen. No matter which of these aspects changes, the entire shape needs to be redisplayed on the screen in order to present the correct shape to the user.

4.2.8 The Domain Data Model

Using `RObject` and `Slot` instances, domain-specific data structures can be created. These data structures do not yet carry any domain-specific logic, though. The development framework superclass for all domain-specific data models, which needs to be subclassed by component developers is the abstract base class `ModelObject` (see figure 4.3). Each `ModelObject` subclass instance is directly related to an `RObject` instance at runtime. When creating a domain-specific subclass of `ModelObject`, a developer needs to write the method which initializes the associated RObject's Slots, `initializeSlots()`, creating those slots which he needs for the domain data model and providing initial values for each.

In this method, the actual domain model structure is defined. The runtime system ensures that this method is called when a ModelObject subclass instance is created. ModelObject subclasses can be arranged in inheritance structures, just like in any object-oriented data model (making the use of data modeling via ModelObject very similar to the normal object-oriented data modeling).

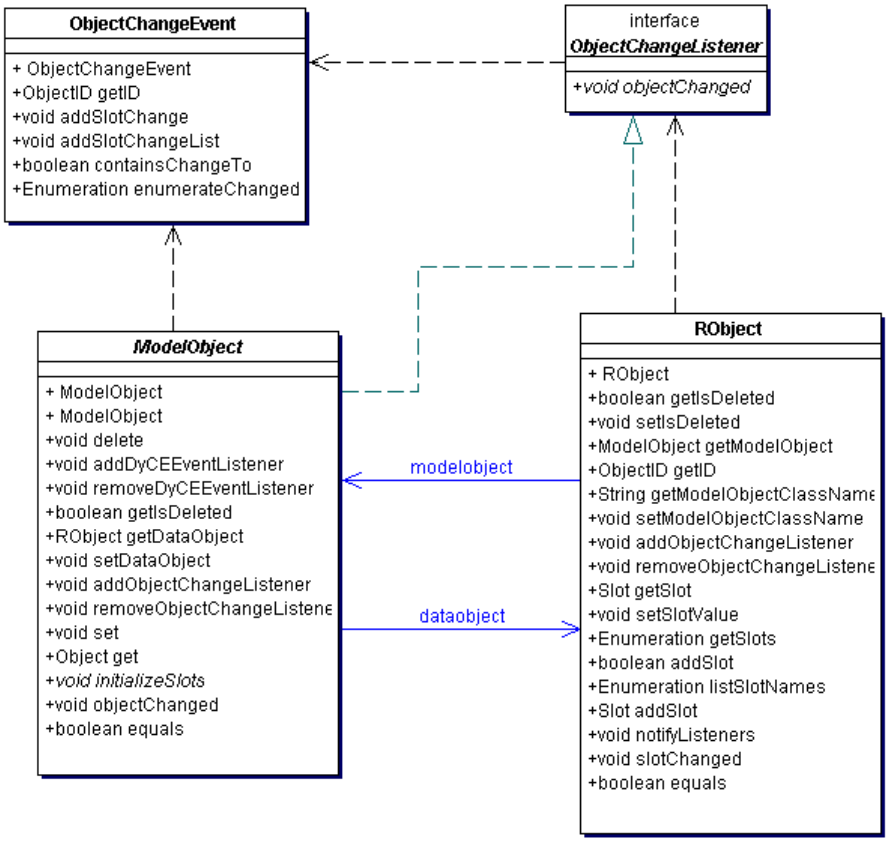

Figure 4.3: Framework base support for the Domain Data Model

In addition to creation of the required Slots in the associated RObject, the ModelObject implementation must provide access interfaces for querying and modifying the data items stored in these Slots. These access methods can then be used to implement domain-specific consistency checks, which cannot be performed by the RObjects or Slots.

The following code fragments are taken from a simple diagram data model used in a shared diagram editing tool. A general base class for all shapes used in a diagram is derived as a `ModelObject` subclass, and from there a specific shape, in this case a rectangle, is derived.

```
public class DiagramShape extends
  GroupComponents.ObjectModel.ModelObject
{
```

```
    public void initializeSlots()
    {
      super.initializeSlots();
      newSlot ("xpos", new Integer (15));
      newSlot ("ypos", new Integer (20));
      newSlot ("color", Color.black);
    }
}

public class DiagramRectangle extends DiagramShape
{
    public Integer getWidth()
    {
      return (Integer)(get("width"));
    }

    public void setWidth (Integer w) throws IllegalArgumentException
    {
      if (w < 0)
        throw new IllegalArgumentException ("width too small");
      set ("width", w);
    }
    // ...

    public void initializeSlots()
    {
        super.initializeSlots();
        newSlot ("width", new Integer (0));
        newSlot ("height", new Integer (0));
    }
}
```

Upon runtime creation of all instances of `DiagramRectangle`, the framework will call the method `initializeSlots()`, which sets up the object's data slots xpos, ypos, color, width and height accordingly.

Also shown in the code fragments are two accessing methods for manipulating Slot contents, `getWidth` and `setWidth`. As can be seen the `setWidth()` method adds consistency checking to ensure validity of the parameters, in this case that a rectangle's width can never be negative. Violation of this constraint is signaled in the way in which this is usually done in Java, by throwing a runtime exception, which includes additional helpful information.

The `ModelObject` framework class provides the following interface methods:

> `abstract void initializeSlots()`
> Framework "hot-spot" method for developer extensions of `ModelObject`. This method needs to be overridden in all `ModelObject` subclasses in order to initialize the concrete domain data model's Slots (in the associated shared data object. Can use the `ModelObject` internal method `void newSlot (String slotName, Object slotValue)` to add a Slot of the given name holding the given initial value.

`void setDataObject (RObject object)`
Set the ModelObject's related shared data object (`RObject` instance).

`RObject getDataObject()`
Retrieve the ModelObject's shared data object (`RObject` instance). If the replicatable object is not yet present on the local machine it is retrieved from the server, added to the ModelObject and returned as method result.

`void delete()`
Delete the model object and the associated `RObject`.
Calls `getDataObject().setIsDeleted()`. This does not directly delete the objects, but merely marks them as deleted. This mark will be taken into account in subsequent replication operations. Objects marked as deleted can be removed by the server's ObjectManager at a later point in time.

`boolean getIsDeleted()`
Query whether the object has been marked as deleted with the delete() method.

`void addObjectChangeListener (ObjectChangeListener listener)`
Adds the given ObjectChangeListener to the set of change listeners bound to this ModelObject (`listeners`).

`void removeObjectChangeListener (ObjectChangeListener listener)`
Removes the given ObjectChangeListener from the set of change listeners bound to this ModelObject (`listeners`).

`void objectChanged (ObjectChangeEvent event)`
Method from the `ObjectChangeListener` interface implemented by the `ModelObject` class. Called by the associated RObject instance when a Slot's value changes (see `RObject` specification). Propagates changes from associated `RObject` to all listeners bound to the ModelObject instance (`listeners`).

`void set (String name, Object value) throws NoSuchFieldError`

Sets the value of the Slot of the given name to the given value (using the Slot's `setValue` method). Throws a `java.lang.NoSuchFieldError` exception if no slot of the given name exists.

`Object get(String name) throws NoSuchFieldError`
Retrieves the value stored in the Slot of the given name. Throws a `java.lang.NoSuchFieldError` exception if no slot of the given name exists.

`boolean equals (Object o)`
Method to determine if the `ModelObject` instance passed as a parameter is equal to the current one. Returns true if o is an instance of `MOdelObject`, or a subclass thereof, and either the ModelObject instances themselves are equal (are actually the same objects) or both model object instances refer to the same shared data object (determined using `getDataObject().getID().equals(o.getDataObject().getID())`).

Discussion of alternatives

The reason for this seemingly complicated way of defining a shared data model, decoupling the data-holding elements (RObject and Slot) from the actual domain logic (ModelObject and subclasses thereof) lies in the benefit of server back-end simplicity and stability gained. The design alternative would have been to allow the developers direct subclassing of `RObject` and extension of this class hierarchy with domain-specific functionality.

The server of the distributed system needs to deal with the data objects: It needs to store them persistently and needs to subject the modifications of the data objects to concurrency control. For this, the server needs to operate with the data object instances (in the case of the design presented above, instances of `RObject` and `Slot`). It is not, however, concerned with application-level semantics, since the replication and concurrency control can be done at the homogeneous shared data model layer (using information about Slot modifications for validation and distribution - the mechanisms for this will be presented in more detail in section 4.3). Therefore, the server does not require the actual domain model implementations.

Would the component developer directly subclass e.g. `Person` from `RObject`, the server would have to deal with `Person` instances (and many more distinctly domain-specific implementations). Any such developer-written subclass can potentially introduce instability, which would affect the entire server. Additionally, dealing with all these `ModelObject` subclasses (holding the implementations in memory, comparing types and instances, etc.) would be a performance problem.

For these reasons, the design decision was made to keep the server object space - and in this way also the interface between client and server - homogeneous.

4.2.9 UML Extension to model shared objects

In the course of designing and developing software systems, it is often the case that designs and specifications need to be written down, as part of the specification documentation or even just to communicate about them with other members of the development team. For this, a common notation is required which is known to all members of the team and which reduces the chance of misunderstandings (which could lead to expensive changes in the design and implementation at later points in time). One such design notation, which has become an accepted standard and is being widely used throughout the industry, is the Unified Modeling Language, UML. The UML is an extensible notation model including definitions of diagrams for most "phases" of the project development life-cycle, from early use-case sketches through to very detailed collaboration and class diagrams.

Since component-based groupware systems, like most software systems, are often developed in a team, it is useful to be able to to model the groupware-specific aspects along with the rest of the design. Especially in development projects where the work is split split up among multiple developers (or teams) working more or less independently, consistent and comprehensive documentation of the design decisions becomes important.

This thesis proposes a number of UML extensions for incorporating the concepts presented in this thesis into the standard UML diagrams. For extension

purposes, UML provides the use of "stereotypes", extension points of the diagram notation which can be used to model domain-specific aspects of the software system (in this case, the domain is the development of component-based groupware). Stereotypes can be easily noted in any UML diagram element, by placing the stereotype name into two angular brackets and adding this to the symbol, e.g. by adding ≪ Customer ≫ to a class symbol denoting a customer in the system. These stereotypes are used to extend the notation to include support for the concepts described so far, shared objects consisting of Slots. In later sections of the thesis, the notation will be extended more, introducing stereotypes for the additional concepts not presented yet.

The shared object model is a static design model describing the structure of the shared data items used in the groupware system. As discussed, the shared object model consists of replicatable objects (subclasses of the `RObject framework class`), which hold the shared data elements as Slots. Conceptually, a Slot of a shared object is similar to an object's attribute. These attributes are modeled in the UML's class diagrams (for an overview over all diagrams available in the UML, see [Fow00]). Therefore, Slots of shared objects are modeled as stereotypes of attributes. In the class diagram, two extensions are made to provide support for modeling shared data objects.

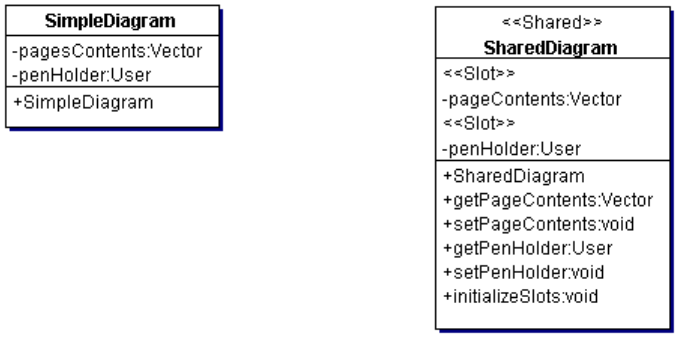

Figure 4.4: UML diagram stereotypes for shared objects

Figure 4.4 shows, on the left, the regular class symbol used in a class diagram. The rectangle representing the class contains three partitions, the top one showing the class name, the middle one showing the attributes of the class and the lowest one showing the operations. In the diagram symbols shown in the figure, there is additional information, about return types and item visibility. These information items can be omitted, e.g. in the early phases of the design when such detail would not be useful. In the class model item extension, shown on the left, stereotypes are added to certain elements. These stereotypes are:

- ≪Shared≫ added as class stereotype to denote the fact that the class is part of the shared data model. Extending a class symbol with the stereotype ≪Shared≫ automatically denotes that the class is derived from the DyCE framework base class `ModelObject`. This information can therefore be omitted from the diagrams for clarity reasons.

- ≪Slot≫ added as stereotype to those attributes which need to be realized as Slots of the shared object model. These Slots need to be initialized in the shared object classes `initializeSlots()` methods. Therefore, this methods is also shown in the diagram element. Extending an attribute with the ≪Slot≫ stereotype implies the appropriate `get...()` and `set...()` methods used for accessing and changing the Slot's contents. These methods can be included in the diagram but can also be omitted for reasons of clarity. For an attribute denoted as a Slot, these methods *must* be present in the implementation, though.

The shared object diagram element can also hold attribute definitions which are not prefixed with the ≪Slot≫ stereotype. This attributes can be realized as regular attributes of the implementation class. They are, by definition, not shared between replicated instances of the model object class.

A number of currently available CASE (Computer-Aided Software Engineering) tools provide the functionality of generating source code from design documents. Using the information about stereotypes added to the diagram, the code generation feature of extensible CASE tools can be enhanced to generate the appropriate source code for the shared model class, i.e. deriving the class from `ModelObject` and automatically generating the entire body of the `initializeSlots()` method, defining all Slots from the model with the appropriate types. For all attributes with the ≪Slot≫ stereotype, the appropriate `get...()` and `set...()` methods, including the method body, can be automatically generated following the pattern that an attribute with the specification ≪`Slot`≫ `myAttribute:aType` results in the generation of two methods (shown for Java as implementation language):

```
public void setMyAttribute (aType aValue)
{
  // ... method body;
}

public aType getMyAttribute ()
{
  // ... method body;
}
```

for setting and retrieving the Slot's contents.

4.2.10 Object Structure - Summary

The OO model used in the data representation component is shown in Figure 4.5.

Slots are accessed through slot access methods in RObject, similar to the Access Objects in [BCM95]. The framework class `RObject` provides the basic mechanisms for creating new slots, accessing and modifying slot contents. The slot accesses are forwarded to the appropriate slot instance, which performs the actual access and returns the required return values.

`RObject` and `Slot` instances form the passive data model of the application, i.e. they contain no processing logic other than what is required to perform the specified operations and maintain consistency of the replicated object set. The

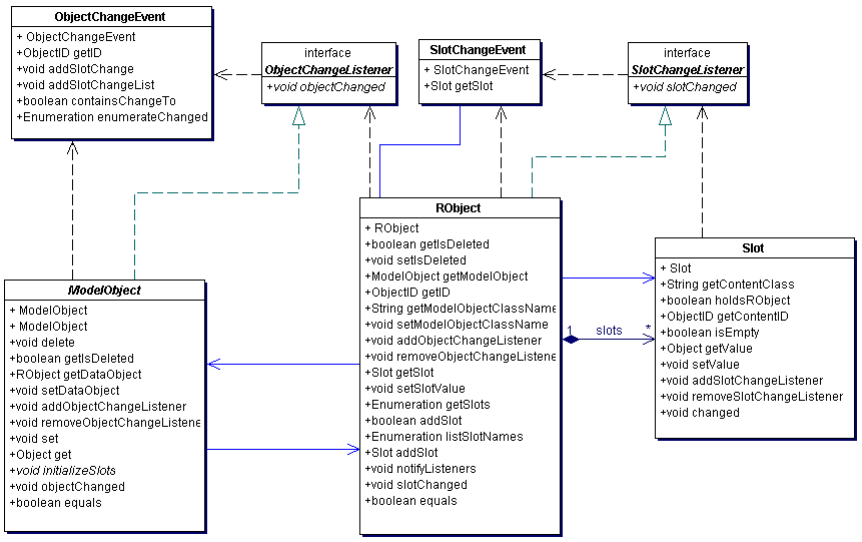

Figure 4.5: Object Representation Model

actual application functionality is built as an object-oriented class hierarchy based on the `ModelObject` class and the framework's support for data storage and object replication purposes. In this way, a single complex data model can be consistently shared between several different applications, each implementing specific functionality on top of the data model. The next section will present how the data models created in this fashion are replicated between the server and the clients.

4.3 Object replication

In order to provide multiple people with access to shared objects, the components need support by the runtime system. The objects need to be kept in a consistent state - changes to a shared object at one site need to be notified and propagated to all components currently accessing this object. Additionally, the components need to perform updates of the display whenever component state changes in order to present all users with a common view of the shared objects. The process of distributing copies of data objects over multiple sites and keeping these in a consistent state is known, e.g. in database work, as *object replication*. This section will first give an overview over different approaches to data replication as well as issues related to replication and will then present the data replication realized in the DyCE framework. More information about replication strategies, along with definitions of the terms used in this section, can be found in [WPS+00]

Replication of data objects or services at all connected sites and allowing concurrent access to the data elements can be used as a means of increasing data availability [3]. In a replicated object setting, there is ideally no single cen-

[3] In [HHB96], availability is defined as "the accessibility of system services to the users"

tral instance of each object upon which all operations (modifications as well as queries of current object state) are performed. Instead, each object is replicated among a number of connected nodes of the distributed system. This replication can greatly reduce the network traffic and thus increase performance, since - given that all objects are somehow kept in a consistent state - query operations (operations accessing the object's state but not modifying it) can be performed by a node on the local object replica and need not be transmitted over the network.

In order to provide optimal support for highly interactive and responsive co-operative applications, the development framework needs to provide mechanisms for object replication between all nodes of the distributed system. These objects must be submitted to a mechanism maintaining global consistency of the co-operatively manipulated shared object space.

4.3.1 Partial Replication

In a highly interactive setting such as a CSCW application allowing users to tailor their collaboration environment by selectively accessing components and data objects (shared artifacts), not all nodes require replicas of all objects at all times. Instead, the nodes only require those objects which make up that part of the object space which the user is currently viewing or editing in the currently used components. Other objects, which are in no way affected by the user's current actions and are not currently being displayed to the user, are of no relevance in the current interaction situation. They can be ignored by the system without any adverse effect on the user interface or the user's actions.

This restriction of replicated data is referred to as *partial replication*, since at a given point in time only a subset of all objects making up the entire shared object space may be replicated to each site. A related concept found in literature is that of *adaptive replication* (see [Wol98] for a more complete definition of the term), where the replication scheme of an object (the set of all nodes holding replicas of the object) is constantly adapted to the current requirements of the system. This is contrasted to *static replication*, where the replication scheme is fixed and precomputed, based on an assumption or prediction of the system's behaviour.

Providing support for partial or dynamic replication in a distributed application can improve the overall system's performance, since:

- only those objects need to be replicated to a node (i.e. transmitted over the network) which are actually currently required; this leads to a reduction in the time required to replicate the objects to a newly connecting or reconnecting node

and it is further stated that "a system is highly available if denial of service request is rare". Replication as a means of improving availability in a co-operative system therefore refers to the goal of reducing the amount of time each user's system is unresponsive to user actions because it is interacting with the other users' systems, e.g. to transfer operation requests or fetch update messages for changed data values. In a centralized system utilizing only single object instances, in which all requests - including read requests - have to be transmitted to a central server or manager process, which schedules and processes the requests and then replies accordingly, a large amount of time is potentially spent on these communications tasks, making the system appear unresponsive or - according to the above definition - unavailable.

- a certain administrative and communications overhead is incurred by replicating objects over multiple nodes: the objects need to be kept in a consistent state in order to allow correct operation of the distributed application; by replicating only those objects which are actually required, this overhead can be reduced, resulting in lower usage of bandwidth and physical memory

In the past years, the computing devices on our desktops have continually become more "network-centric devices", which are more than just the "dumb" terminals known from the mainframe era of computer applications, since they have local processing power equivalent to top-range personal computers while at the same time drawing their data and programs from the network. With the availability of such systems, it seems reasonable to enhance communication and productivity by providing additional co-operative applications to the users. The same holds for the trend to install Intranets, internal network and computing infrastructure based on widely available and accepted technologies such as Web servers and Web browsers. The applications benefiting from this infrastructure can be designed in the form of an object-oriented distributed system using object replication to provide high responsiveness to the user while at the same time maintaining consistency of the common shared artifact, the base of the collaboration. Even within a corporate Intranet the scarcest resource is bandwidth, thus business applications should be designed and developed to, on the one hand, support co-operative problem solving activities between co-workers by profiting from the locally available computing power, while at the same time not waste available bandwidth. Especially when large information structures are maintained and modified co-operatively, it seems undesirable to replicate all available data to all connected systems, leading to the need for partial replication mechanisms which detect which data elements are currently required by which nodes and transparently replicate them to the consumer nodes.

Dynamic Replication

An algorithm for dynamic data replication in a distributed system is presented in [WJH97]. In this article, Wolfson et al describe the Adaptive Data Replication algorithm (ADR). This algorithm maintains a connected replication scheme within a distributed system exhibiting a tree-based topology (in the course of the work by Wolfson et al, this topology is later extended to a graph-based network topology). The replication scheme (the set of all processors in the system currently holding a replica of a data item) is maintained and adapted by periodically performing a set of three tests: the expansion test, the contraction test and the switch test. In each test, information about the past number of read and write requests as well as the total number of requests processed by each node is used to determine whether to

- expand the replication scheme by replicating a data object to a node of the system not previously part of the replication scheme but at the border of the replication scheme (i.e. a node not currently holding a replica of the data item but directly connected to a node that is part of the data item's replication scheme),
- reduce the replication scheme by releasing a data item's replica at one of the nodes at the fringe of the replication scheme (i.e. a node currently

holding a data item's replica which is connected to a node which is not currently part of the replication scheme),

- switch a data item's replica from a node currently holding a replica (where it is subsequently removed) to a node not currently holding a data item replica.

Using these three tests, an "amoeba"-like replication scheme migrates through the network, expanding towards those nodes which require local replicas of the data item, since they frequently access the item and can thus greatly benefit from the local copies, while at the same time, the contraction test and the resulting reduction of the replica set leads to "moving away" from processors not currently accessing the data item and thus a reduction of control and update messages which have to be propagated to all replica items, even those which would currently be dispensable.

Performance evaluations of this algorithm show the benefits of the dynamic replication in terms of the cost incurred by propagating read and write accesses through the network. It is shown that over a period of time intervals the replication scheme converges towards the optimal replication scheme for the current read/write activity in the system. When the read/write characteristics of the system (expressed as schedules consisting of read and write requests over time) change, the replication scheme is adapted accordingly, to once more converge to the optimal.

A comparison of the adaptive replication scheme with a static precomputed replication scheme and with a theoretical lowest bound on the cost of the replication shows that the ADR algorithm is superior to static replication schemes since it constantly changes the replication scheme to match the current read/write situation of the system.

Lazy Replication

The lazy replication (also called "lazy loading") (see [LLSG92]) approach to partial data replication is much simpler that the ADR approach. Lazy loading typically does not rely on an active distributed algorithm. Instead, it relies on a system's ability to detect if it needs to fetch an object before it can proceed.

Assuming a system of interconnected objects which can potentially be replicated, a lazy replication approach will only fetch a replica of an object when that object is actually accessed by the system. As long as an object reference is not resolved (and the referenced object accessed), the object is not available on the system. The opposite approach, in which all references between objects are followed upon replication of an object and the entire network of objects is sent immediately, is called "eager replication".

The lazy replication approach is simpler to realize than ADR and other dynamic replication approaches, since it does not require the ability to access non-replicated objects over the network (e.g. using remote proxies). The object access mechanism can rely on the fact that an object replica will be locally available as soon as it is accessed (but not before it is accessed the first time). Assuming that not all object references are resolved automatically when accessing an object, then lazy replication replicates less objects than eager replication. The lazy replication approach needs support from the runtime system or development framework in order to detect when an object reference is followed and an

object is accessed. Accessing the object needs to be deferred until the required object has been replicated.

4.3.2 Discarding of replicas

A requirement that stems from the previous discussion is the need to be able to discard replicas when they are no longer needed at the current system. Such a situation can arise, for instance, when a user stops working on one section of a common document (composed of object from the common shared object space) and focuses his attention on another section of that document. Once all operations performed on the previous section have committed and the section is no longer displayed on the user's screen, the replicas can be discarded. If this discarding of replicas is not provided, the set of replicated objects at each node continually grows and while the user navigates through the document data, large portions of the shared object space are gradually loaded into the local system, subjecting them to all the control mechanisms responsible for replica management and overall consistency of the shared objects. This behaviour is especially undesirable when one takes into account that a large-scale co-operative system could potentially include more data objects than can possibly be stored in the local system's physical memory - consider a co-operative Web browser which gradually builds a local copy of the entire Web content. Instead, mechanisms are required which allow the system to detect which replicas are no longer needed in the current interaction situation so that these replicas can be safely discarded without adverse effect on the user interface and overall data consistency.

These detection mechanisms need to be sufficiently intelligent to allow interactive navigation of the shared object space, especially backtracking (going back to previously viewed sections of the object space). Taking the Web browser example, it seems reasonable that a page's contents are not discarded as soon as a hyperlink to another page is followed. Were this the case, the previous page would have to be reloaded (replicated again) when the user, upon discovering that his navigation did not bring him where he actually wanted to go, immediately upon seeing the new page pressed the "back" button of his browser. Instead, the system needs to maintain an intelligent caching strategy (as many browsers do) which retains replicas for a certain period of time before discarding them. Current Web browsers feature a page cache, which is not strictly speaking a means of replication as addressed by this thesis, but rather a "dead" copy. The elements in the browser cache are not subjected to consistency control and update mechanisms. Changes in the original Web page are not detected until the page is revisited and, even then, insufficient change detection mechanisms sometimes result in an outdated cached version of the page being displayed even though the actual Web page has since been modified. No corrective action is taken to remedy this situation and it remains left to the user's discretion whether he wants to initiate a reload of the page in order to be sure the browser is displaying current data.

An additional performance improvement can be achieved by not only caching a number of replicas to allow the user to backtrack through his actions, but also prefetching a certain area ε around the user's current "position". Since not all distributed applications have a straightforward document structure such as the hypermedia structure in the Web, the partial replication mechanism needs to be tailorable to suit the application developer's needs. For instance, the algorithms

for detecting the optimal caching duration and ε-prefetch boundaries need to be provided in a way which is configurable or which can be exchanged by new mechanisms.

The problem that arises when discussing partial replication strategies is how to detect when objects are required at the current node. Taking into account a large set of objects interconnected by object references, it is not sufficient to follow the references from any replicated object and to also replicate those objects referenced by them, since this would automatically lead to a complete replication of the entire object space. Instead, partition boundaries within the document model have to be identifiable and objects need only be replicated up to these partition boundaries. An easier approach is to replicate on access: when an object is first accessed by a client, it is replicated to that client and the client is added to the replication management system since changes to the object need to be forwarded to that client as well.

4.3.3 Distributed Garbage Collection

An interactive object-oriented system allowing users to create, manipulate and delete objects and object relations produces what is referred to as garbage: objects which are no longer referenced by any other objects and are thus unreachable and no longer needed by the system. Several object-oriented programming languages feature what is called a garbage collector: a system component, often part of the runtime system's memory management system, which is able to detect unreferenced objects and remove them from the application's heap, freeing space for newly created objects. Many garbage collection algorithms operate on a mark-and-sweep algorithm or a variation thereof (see [JL96] for details about garbage collection algorithms), resolving object references in the object graph and marking those objects which are traversed. Once a mark-and-sweep pass has completed, any objects not marked can be considered unreachable (i.e. garbage) and can be removed from the application's memory space without adverse effects on the other objects.

In a distributed application, garbage collection becomes a more difficult task since objects that are no longer referenced within one node's object space may well still be referenced by objects contained in another node (see [PS95] for more details on Distributed Garbage Collection). Additionally, in a replicated setting an object replica is a member of a *replica group* - a set of objects on various hosts which are all replicas of a common data object. Objects in a replica group need to be available and accessible in order to allow correct operation of concurrency control algorithms. Two invariants hold which make distributed garbage collection in a replicated setting a complex activity:

- if an object is garbage collected on one node it needs to be removed from the replica group on each system in order to remove it from the concurrency control and replica management;

- since the objects in the replicated system are managed by some form of replica management component which keeps track of all replicated objects, a mere mark-and-sweep style of garbage collection would never detect any garbage-collectable objects, since all objects are at least accessible via the replica management system.

As previously discussed, in order to really benefit from the partial replication, it seems desirable to be able to not only discard any unneeded replica objects but to also remove them from the replica management and concurrency control mechanisms.

For these reasons, the garbage collection mechanism needs to be tightly integrated with the replication mechanism, turning it into a distributed garbage collection mechanism. The system needs a means to detect those replicas that are no longer needed and to subject them to the garbage collection mechanism.

4.3.4 Object Replication in DyCE

The object replication mechanism in the DyCE framework uses a variant of the Lazy Replication mechanism. In the approach as realized in DyCE, all objects initially reside on the server and they are replicated to the clients as soon as they are accessed by them. Supporting object replication is part of the functionality of the Object Manager. Object Management is a service provided by the DyCE server's `ObjectManager` and accessed by `ObjectManager` instances on the clients. The server's Object Manager holds all available RObjects and manages their persistence using an object-oriented database.

As has been discussed in the section regarding Lazy Replication, this approach requires identifying the moment at which an object is accessed. Therefore, all object accesses need to be performed using the client's `ObjectManager` instance. The client's `ObjectManager` holds the object replicas available on the client. The set of objects stored in the client's Object Manager is a pure subset of the objects held in the server's Object Manager.

As has been discussed in sections 4.2.5 and following, data objects in DyCE are divided into the domain model part, derived as a class hierarchy from `ModelObject`, and the pure data part, consisting of `RObject` and `Slot` instances. Since the `ModelObject` instances may not hold shared data elements, only the RObject instances are relevant for replication and persistency purposes. Using the `ModelObjectClassName` property of the `RObject`, the necessary `ModelObject` instance can be recreated at any time by creating a local instance of the correct class.

For retrieving the desired Objects, the Object Manager provides a number of interface methods. Recall from the section about the data object representation that each object in the system is identified by a unique identifier of type `ObjectID`. The class diagram of the client and server Object Managers as well as the common object management interface is shown in figure 4.6.

When a Groupware Component needs to retrieve a data object, it uses the client's `ObjectManager` interface method `public RObject getObject (ObjectID objID, ObjectID clientid)`, giving the ID of the desired object. If the object is available in the client Object Manager's `objectList`, it is returned directly, otherwise it is first fetched from the server's Object Manager and added to the local object space. In this way, there is a central point at which object access can be intercepted and suspended until the object has been successfully replicated.

When an object is replicated from the server to the client, it is serialized and sent over the network in binary form. In this process, the contents of any Slot holding a reference to an RObject or a ModelObject are replaced with `SlotContentPlaceholder` instances. Slots holding primitive or other data values are serialized into the object and included in the transmitted data stream.

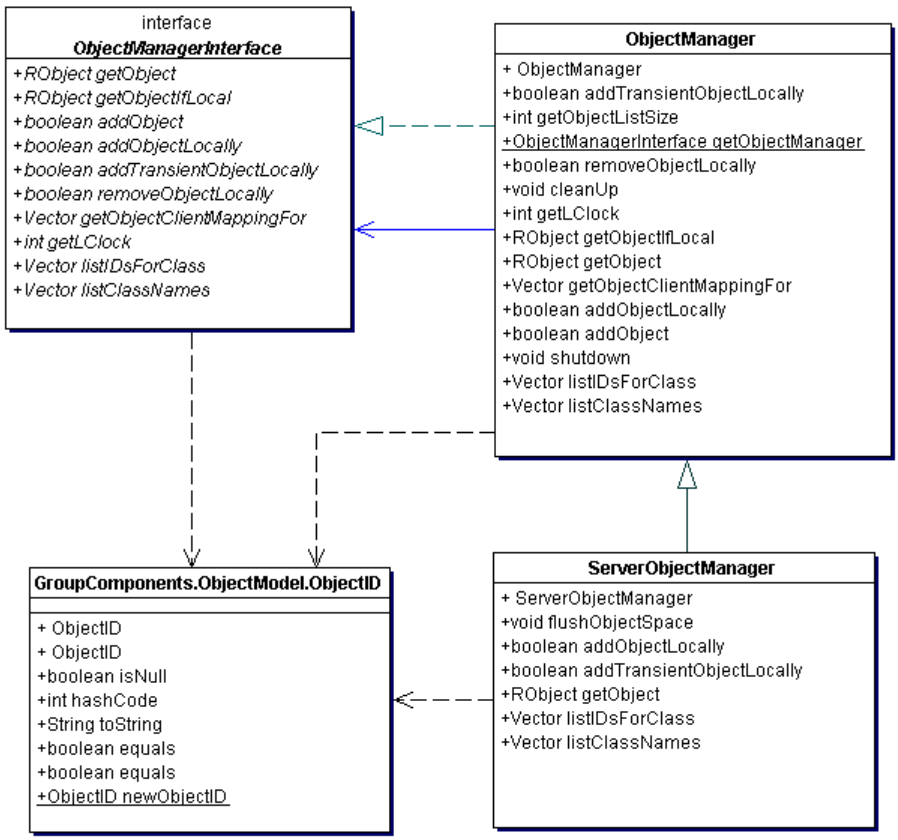

Figure 4.6: ObjectManager class diagram

On the client, the SlotContentPlaceholders are resolved as soon as the Slot's contents are first accessed. The ObjectID wrapped in the placeholder is used to fetch the required object (which is done just like described above). In this way, the system avoids replicating entire sub-nets of linked data objects in cases where only a few slots of a few of these objects would have actually been relevant.

For each replicated object, the server's Object Manager maintains information about the object's replication scheme (the list of clients currently holding replicas of the object) in its `objectClientMapping` data structure. Here the server has access to the replica group of each object. When a client `ObjectManager` requests an object from the server, it passes its unique client ID which the server enters into the `objectClientMapping` structure. This information is needed for change propagation, performed by the system's Transaction Manager, which will be described in the following section.

The ObjectManager service provides the following interface functions:

`RObject getObject(ObjectID object, ObjectID clientid)`
Retrieve the object of the given ID from the Object Manager. If getObject is called on the client Object Manager and the object is not available in the client's object space, retrieve the object from the

server's Object Manager, adding the client to that object's replication scheme. If the method is called on the server and the object is not available in the server's object space, attempt a fetch from persistent storage. Return null if object is not available.

`RObject getObjectIfLocal(ObjectID object, ObjectID clientid)`
Retrieve the object with the given ID from the Object Manager if it is available locally. Do not attempt to fetch the object over the network if it is not locally available.

`boolean addObject (RObject object)`
Add the given object to the Object Manager. If called on a client, add the object to the server's Object Manager as well. When adding the object to the server's Object Manager, the object is automatically made persistent in persistent storage.

`boolean addObjectLocally (RObject object)`
Add the object to the Object Manager. If called on client, do not attempt to send to server. If called on server, add object to persistent storage.

`boolean addTransientObjectLocally (RObject object)`
Add the object to the Object Manager. If called on client, do not attempt to send to server. If called on server, do not add object to persistent storage.

`boolean removeObjectLocally(ObjectID id)`
Remove the object with the given ID from the Object Manager's object set. Even if called on server, do not remove object from persistent storage.

`Vector getObjectClientMappingFor(ObjectID id)`
Return the replication scheme for the object of the given ID - returns a list of IDs of all clients to which the object is currently replicated.

`int getLClock(ObjectID id)`
return the logical clock value for the object of the given ID. See section 4.3.5 for use of logical clock values in transaction validation.

`Vector listIDsForClass (String classname)`
Return a set of all object IDs of `RObject` instances with a ModelObejctClassName equal to the given class name. Used to retrieve all instances of a certain class. When called on server, also returns IDs of all matching objects in persistent storage.

`Vector listClassNames()`
Returns a set of all ModelObjectClassNames of all `RObject` instances stored in the Object Manager's object space. If called in the server, also returns all ModelObjectClassNames for all objects in persistent storage.

4.3.5 Object Consistency: Transaction Management

When replicating objects through the distributed system, there must be a mechanism which supports keeping all these objects in a consistent state, i.e. propagating changes performed at one system to all other systems which hold replicas

of the object. Since we are concerned with a distributed co-operative system supporting synchronous collaboration between multiple users, we are faced with the issue of multiple users making simultaneous but conflicting changes to the same object(s), which could potentially lead to inconsistent displays and a disruption of the collaboration. Approaches for maintaining such consistency can be roughly categorized into one of the following two classes:

- **Conflict Prevention**: Using mechanisms such as distributed locks or semaphores, the systems are prevented from making conflicting changes. Before a system can perform changes to an object, it needs to acquire the related lock(s). If a desired lock is not free, then another system is currently holding the lock and therefore the changes cannot be performed. After all required changes have been performed and the related locks have been released, the changes are propagated to all systems holding replicas of the affected objects, thus making all systems consistent again. Another example of such an approach is floor control, where the application prevents users from performing operations when they do not "have the floor". Such floor control is often also linked to the user interface, in order to indicate to the users that they cannot currently take an action.

- **Conflict Resolution**: In this class of approaches, a system is initially allowed to perform changes to objects. If these changes are in conflict with changes done on other machines (which can be detected using appropriate Concurrency Control mechanisms), then specific conflict resolution mechanisms are needed to resolve these conflicts and create a consistent state again. Conflicting operations can be undone (thereby undoing the work of at least one of the users) or they can be reordered, transformed and performed in different order.

A more detailed treatment of the technical aspects of concurrency control mechanisms in groupware systems can be found in [EG89]. Greenberg and Marwood ([GM94]) discuss in which ways concurrency control and choice of concurrency control policies influences the user interface. After all, when designing groupware systems we are faced with supporting groups of users in a distributed setting and not just a set of connected systems.

Transaction Validation

DyCE uses Transactions to group object manipulations and accesses and uses a centralized transaction-based concurrency control for maintaining data consistency. Each DyCE client wraps operations which access the shared data model into transactions. For validation and distribution reasons, a transaction consists of three parts:

- **Read-Set T.read** : Ordered list of read operations performed on Objects' Slots (ordered in the order in which the read operations were performed during the transaction).

- **Write-Set T.write** : Ordered list of write operations performed on Objects' Slots (also ordered chronologically).

- **Logical Timestamp T.lclock**: Vector of RObject lclock (logical clock) values (tuples of ObjectIDs and numeric lclock values) at the point at which the objects were modified.

For all data access operations performed on the shared data model, there has to be a running transaction. Each access to a Slot, either through `setValue` or through `getValue` methods (for respectively writing and reading the slot) adds an operation object to the running transaction. For Slot `s1` of RObject `r1`, reading the Slot's contents through `getValue` in Transaction t1 appends a `readOperation` object to `t1.read`. Writing (modifying) the slot's contents through `setValue` in Transaction t1 adds a `writeOperation` object to `t1.write` and a tuple containing `r1.id` and the current RObject timestamp value `lclock` of r1 to `t1.lclock`. When the transaction is committed, the timestamp values of all affected RObjects (an RObject is affected if a writeOperation has been performed on it) are advanced. The transaction is sent to the server for validation.

Transaction Types

Transactions can be distinguished according to which forms of operations (read, write, create) on the shared data model they allow. For performance and efficiency reasons, the framework distinguishes between three different types of transactions with different characteristics (see table 4.2).

Transaction Type	Permitted Operations	Description
Modify Transaction	read, write, create, change	Basic transaction for all accesses to shared data - subject to concurrency control
Display Transaction	read	Groups local display operations, allows no modification of data, not subject to concurrency control
Validate Transaction	read, write, create, change	used *only* on the server, in the phase of validating and replaying a transaction

Table 4.2: DyCE transaction types

The basic transaction, which allows all forms of data access and manipulation is the *modify transaction*. This transaction is performed by a client and gathers up all operations on data objects. When a modify transaction is committed, it is sent to the server for validation. For reasons of efficiency and application response time, the clients perform transactions optimistically, that

is transactions are performed, the user's display is updated and work is allowed to proceed, even though the transaction has not yet been validated. If, during validation, the transaction is found to be in conflict with other previously committed transactions, the local changes as well as any subsequent changes are undone and the shared data model is returned to a consistent state. For this, each client maintains a local *undo buffer* of unconfirmed transactions.

Since during the refreshing of a component's display elements of the shared data model need to be accessed, and all accesses to data objects need to be wrapped in a transaction, DyCE knows a second transaction type, the `DisplayTransaction`. Display Transactions only allow read access to shared data objects. Any modification of data objects will result in a runtime exception being raised by the framework and operations being aborted.

Display Transactions can be used to detect additional object dependencies, since, if an object is in the readSet of a component's Display Transaction, then that object (or rather a Slot thereof) appears to be relevant for the correct display of a component. This indicates that the component needs to be notified when such an object or Slot is changed at a later point in time. This information can be automatically gathered at runtime by the framework, without the component developer having to explicitly add `ObjectChangeListener`s or `SlotChangeListener`s to shared model objects.

`ValidateTransaction`s are used only on the server in the phase of validating and replaying a transaction received from a client. Since the server's object space also deals with `RObject` and `Slot` instances, any modifying accesses to these need to be wrapped in transactions as well. For this, the server creates `ValidateTransaction`s, which allow all kinds of accesses (even though object reads are not replayed and not distributed). `ValidateTransaction`s are created only in the process of validating transactions and are therefore not themselves distributed when committed.

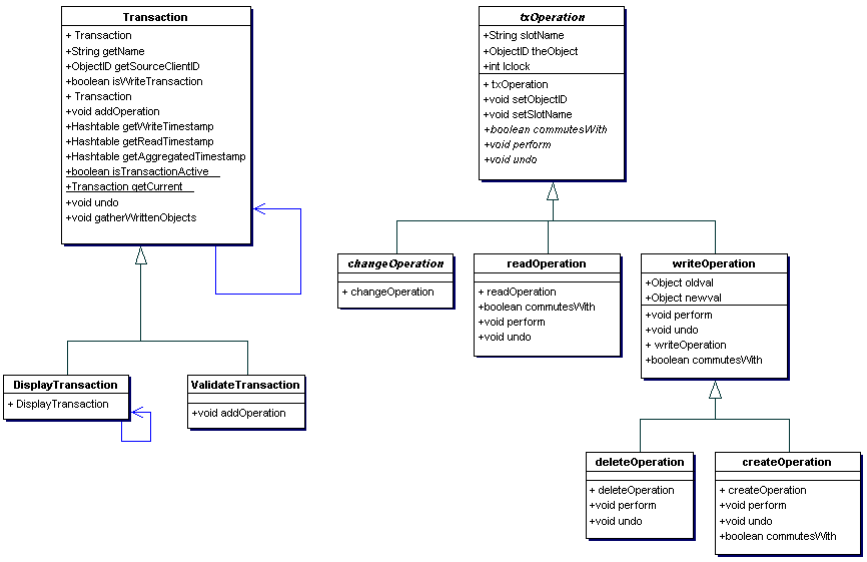

Figure 4.7: Transaction object model

The class hierarchy of transaction objects is shown in figure 4.7 along with the hierarchy of operations that can be performed within transactions. The set of transaction can easily be extended in this part of the framework in a fashion similar to the way in which the `DisplayTransaction` has been derived. In order to create additional special transaction types (e.g. ones which only permit changes to specific parts of an object model, or which restrict permitted transaction size), the `addOperation`, `addWriteOp`, `addReadOp` or `addChangeOp` methods need to be overriden, checking the transaction operations to be added before adding them.

Transaction Properties

The Modify Transactions in the system are required to fulfill the *ACID* properties known from database management systems: Atomicity, Consistency, Isolation and Durability. In [Wei93] these terms are defined as follows (p.330 f):

- *Atomicity* [...] means that each transaction appears indivisible with respect to crashes. In other words, each transaction appears to occur either completely or not at all; partial effects cannot be seen.
- *Consistency* means that each transaction, when executed alone and to completion, preserves whatever invariants have been defined on the system state.
- *Isolation* [...] means that transaction appear indivisible to each other: if a group of transactions is executed concurrently, the effect is the same as if they were executed sequentially in some order.
- *Durability* [...] means that the effects of committed transactions are very likely to survive subsequent failures.

These transaction properties are ensured by the server's Transaction Manager in the following ways. Atomicity is ensured by storing the previous values of the affected Slots in "shadow" slots while performing the transaction. When the transaction is committed, the affected object is written to persistent storage. In the case of a crash, the original, unchanged object can be retrieved from persistent storage in the state in which it was previous to the transaction. In the case of a transaction abort (or rollback) on the client, the shadow values are written back to the Slots, restoring the state before the transaction. Perservation of consistency is the responsibility of the domain-specific access methods in the `ModelObject` subclasses. The various methods for setting Slot values need to check the domain-specific consistency constraints. The isolation property (which is also often called *serializability*, in effect states that transactions cannot influence each other) is ensured by the fact that a client's Transaction Manager does not process any transaction received over the network while a local transaction is being performed. And, vice-versa, no local transaction is allowed to begin while the Transaction Manager is processing transactions received from the server. In this way, the only way that object state can change *during* a transaction is by operations performed in that transaction. These operations are gathered in the transaction object, sent over the network to the server for validation and distribution. The transaction cannot depend on any observable changes that occurred "outside" of the transaction. The transaction

serialization mechanisms that take place in the server's Transaction Manager ensure that no two concurrent transactions are allowed to commit which affect (modify) the same slots. One of these will be undone. Therefore, the result of the serialization of committed transactions will be the same regardless of the order in which the transactions are actually performed. The durability property is ensured by storing the `RObject` instances in persistent storage. When a transaction commits, the `RObject` instances stored in persistent are updated accordingly. This way, crash recovery for many forms of system crashes can be performed simply by reloading the RObject instances from persistent storage. Since only consistent RObject states (the states of RObject instances at the end of a transaction's committing, which are by definition consistent) are placed into persistent storage, a consistent point from which to resume can be loaded from the storage.

4.4 Components in the DyCE Framework

The object diagram in figure 4.8 presents the core of the DyCE framework, the classes which form the framework's "hotspots" for extension and the classes for managing the framework extensions:

- `MobileComponent` - the base class for all Groupware Components,
- `ModelObject` - the base class for all domain model classes,
- `RObject` and `Slot` - the constituents of the generic data object model, created and referenced through domain-specific `ModelObject` subclasses,

The relationships between `RObject`, `Slot` and `ModelObject` have already been explained, therefore the details of the `RObject` and `Slot` classes have been hidden from the diagram - please refer to figure 4.5 for details. `MobileComponent` is the class to be subclassed when developing a new Groupware Component. In order to make the new component usable in the framework, the following methods need to be implemented with component-specific behaviour:

> `void prepareGUI()`
> This method must contain all GUI initialization code required for the component (creation and initialization of GUI elements such as text elements, labels, etc.) The method is called by the framework before opening a Groupware Component on a machine.
>
> `void giveTaskBindings (Vector tasks)`
> Add the set of tasks published by this component, as instances of `ObjectComponentTask` to the list passed as collecting parameter to the method. See section 4.6 for detailed information about the task model.

When a Groupware Component is instantiated on a client, the framework calls the `ModelObject` subclass constructor to create the object, then calls the method `setModel (ModelObject model)` to set the Groupware Component's model object. Then a call to `prepareGUI()` is used to set up the Component's user interface.

When a component is actually opened on a client (after its GUI has been prepared), it is notified of this fact through the `componentOpened()` method. Inversely, when a component is closed, it is notified of this through the `componentClosed()` method.

The remaining methods of the `MobileComponent` class seen in figure 4.8 which have not yet been explained are either implementation-specific or are related to the task model, which will be explained in detail in section 4.6. The full Java source codes of an example component and the associated domain data model are shown in the appendix. These illustrate, among other things, how the framework hotspot extensions are used.

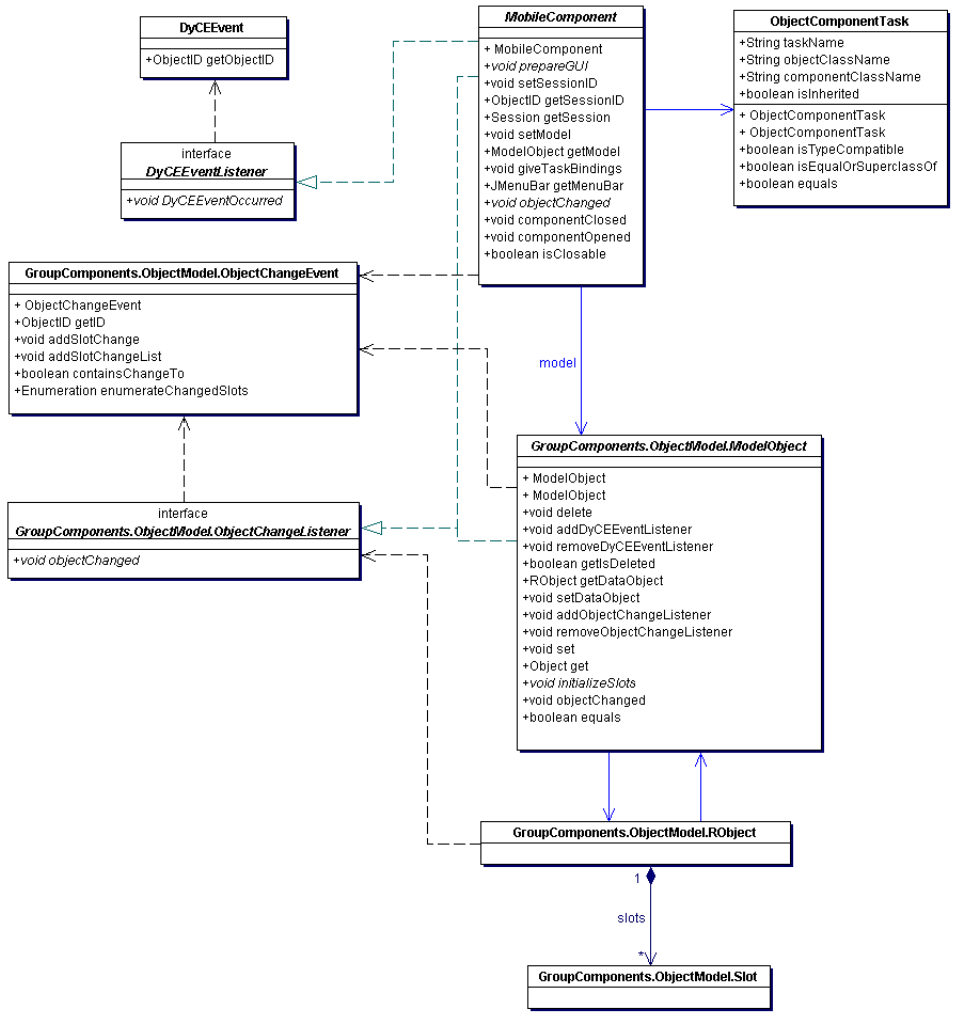

Figure 4.8: Components in the DyCE Framework

4.5 Coupling distributed components through events

In addition to shared data objects, as provided by the `RObject` framework class, a second coupling primitive for the combination of components is that of event coupling. An event in this case can be defined as a special occurrence taking place at a certain point in time but not directly reflected in a change of shared state. These events do not only relate to user interface events, such as the interaction with a specific widget, e.g. the pressing of a button, but also to events *within* a component.

For some component connections, event-based coupling can be more straightforward than coupling through shared state and can provide a meaningful alternative communication primitive. Event-based communication can augment coupling through shared data objects for those cases where a change in complex data structures would be more difficult to detect and react to than the occurrence of a specific event. Take for instance a component where for each user who enters all users in the session should play a certain sound, to alert others of the fact that a new user has joined. When a user leaves, a different sound file is to be played. With only shared data as a communication primitive, all components would need to monitor a shared user list for changes and, when a change occurs, compare the current state of the user list to the previous state, in order to find out whether the change was caused by a user just leaving or by a user joining.

The simpler alternative is to broadcast an event to all connected components, Those components who can correctly interpret the event can then, in the above example, play the appropriate sound file, depending on whether they received a "user joined" event or a "user left" event.

It is important to note that the event-based communication in DyCE is seen merely as a supplement to the communication using shared data objects. While there are some collaboration support frameworks which rely only on event broadcasting (see the overview in the state-of-the-art), such architectures need to make special provisions to support late-comers, asynchronous collaboration, etc.

4.5.1 Extensible event class hierarchy

The event communication between Groupware Components is based on an extensible hierarchy of event classes. The root of this hierarchy, as can be seen in figure 4.8, is the class `DyCEEvent`. This base class can be extended by component developers to model specific events which occur in their components. Any specific events that are to be broadcast can be derived as subclasses, e.g. the event class `DyCEEventUserJoined` which can be used to notify other components about the fact that a user has joined a session.

The `MobileComponent` base class for Groupware Components implements the `DyCEEventOccurred(DyCEEvent event)` method. This method can be overridden in `MobileComponent` subclasses when a component is required to react to DyCEEvents in a specific way.

Each event has a *source*, indicating on which client the event originated. Additional attributes can be specified for event object subclasses, depending on

the requirements of the event.

4.5.2 Object-related event channels

Event distribution in the DyCE framework is tied to the shared objects which are manipulated by the Groupware Components. Event distribution is based on the shared data replication in the distributed system. One could say that each replicated object defines a specific virtual event channel over which event are distrbitued and delivered to all components currently subscribing to this event channel (because they hold replicas of the shared data object *and* have indicated interest in the events broadcast over this channel.

Events are created by creating instances of a specific subclass of `DyCEEvent`. The events are then broadcast by the component using a method inherited from `MobileComponent`, namely `void broadcastDyCEEvent (ObjectID oid, DyCEEvent event);`. The event object is serialized and sent to the server, which distributes it to all connected clients who currently hold a replica of the RObject with the given object ID.

A `MobileComponent` subclass can register its interest in a specific object-based event channel by adding it to a shared data object as a `DyCEEventListener`, similar to the regular event listener architecture in Java. This means that the component will receive any events which are distributed "on behalf of" this shared object (over the object-based event channel).

Each event holds the *lclock* value (logical clock) of the object over which it has been distributed. The value stored in the event's lclock is the logical clock value of the object at the point in time when the event has been created. The object's logical clock value is not advanced when an event is broadcast over it.

4.5.3 Synchronizing events and object modifications

As has been pointed out, the main coupling medium for the Groupware Components remains the shared data object (replicated between sites). This is supplemented with the event-based communication. The event-based coupling of components needs to be synchronized with the modifications of the shared object state which also couples the components. This synchronization is especially important, since the reaction to an event could depend on the current state of a shared data object (e.g. the event to play a sound file as a reaction to a certain event could rely on the fact that a shared data object contains the name of the sound file to be played).

The distribution of object changes in DyCE is based on transactions. Transaction objects are created at the source of an object modification. These transactions are sent to the server, are validated and - if they are allowed to commit - are distributed to all affected clients. Therefore, the event broadcast mechanism needs to be synchronized with the transaction delivery mechanism.

It is the responsibility of the transaction management system to ensure that the following properties hold:

- If an event E has originated at client C1 on object E.obj, then it will be distributed to all clients who currently hold a replica of object E.obj (this set of clients is called "event destination clients"). This includes the client on which the event originated.

- A transaction T is said to have *happened before* an event E if: Transaction T originated on the same client as the event, T has the object E.obj over which the event has been broadcast in its read-set T.read or in its write-set T.write and the lclock value stored in the transaction for that object is smaller than or equal to the lclock value stored in E.
- An event E is said to have *happened before* a Transaction T if: Transaction T originated on the same client as the event, T has the object E.obj over which the event has been broadcast in its read-set T.read or in its write-set T.write and the lclock value stored in the transaction for that object is greater than the lclock value stored in E.
- Any event will be distributed to all event destination clients after all transactions which *happened before* the event have been distributed to the affected clients.
- On each client, any pending transactions which *happened before* a received event E will be processed (replayed locally) before the DyCEEventListeners registered on the object E.obj are notified through their `DyCEEventOccurred` method.
- On each client, any pending DyCEEvent E will be delivered to the registered listeners before a transaction T is processed for which E *happened before* T.

Using these assertions, the runtime system needs to ensure that the shared object state when a component reacts to an event is *at least* the shared object state as it was on the client which originated the event when the event was originated. Note that due to the distributed and non-realtime nature of the Groupware Component framework, the *happened before* relation cannot safely be defined for transactions and event originating on different systems. The above assertions are sufficient, though, that any data states on which an event relies have been created on each client before the event is actually delivered (the event listeners are triggered using their `DyCEEventOccurred` method.

4.5.4 Using Event Communication in Groupware Components

Using the object-based event mechanisms as described above, Groupware Components can communicate by attaching themselves to specific objects as event listeners and by broadcasting events over the virtual event channels supplied by the objects. The motivating example, of playing a certain sound file when a user enters a session and another sound file when a user exists a session, can be achieved by deriving the event classes `DyCEEventUserEnters` and `DyCEEventUserLeaves` from the `DyCEEvent` base class and broadcasting these events over the shared object modeling the session's user list (this object is accessible to all MobileComponent subclass instances in a session - see section 4.7 for more information about sessions). Now, whenever a user enters a session, a component can broadcast an instance of `DyCEEventUserEnters` over the user list's virtual event channel and all components which are already open in the session and which subscribe to events broadcast over the user list will receive the event and react accordingly.

Using this event framework, different components can also react to similar events in different ways. While one component could react to the `DyCEEventUserEnters` event by playing the sound file, another could send an email or perform some change in a data model (e.g. checking workflow data for data relating to this user).

4.6 Task-based programming model

When using a number of components to interact with data objects (or documents), a common mechanism is required to perform the matching between an editable object and the component or application used to edit it. The users have access to a potentially large number of document objects and the system needs to provide the correct component(s) when the user indicates his wish to edit a certain object.

In modern GUI environments, a number of different schemes are employed to perform this matching between a document (often stored in a file in the file system) and an application. For instance,

- in Windows environments, applications are used to create, present and modify documents which reside in the file system. The selection of the correct application for a document is done based on the document file name extension, e.g. a file with the extension ".ppt" is opened using Microsoft PowerPoint, when the user indicates his wish to open the document (e.g. by double-clicking on the document icon on the desktop).

- in the object-oriented desktop environment of OS/2, the Workplace Shell, which is built on top of OS/2's Presentation Manager, the mapping between a document and the application used to edit it is a property of the document object. Regardless of the document's file name, the application noted in the document object is used to open the document. New documents of a certain class are created using templates which include, among other things, information about the related application.

A direct mapping (or binding) between a shared object and a component can prove to be insufficient for a number of reasons.

In typical collaboration scenarios to be supported by Groupware Components, different users perform different roles within the collaboration process. In order to perform their role, the users could potentially need to use different interaction mechanisms(i.e. different components). For instance, in a collaborative learning scenario, the user performing the role of the teacher might need a more sophisticated tool for presenting and modifying slides than the students, who merely use a slide viewer.

The use of Groupware Components in complex collaboration scenarios needs to be controllable by a central control mechanism, similar to a workflow system. Here, merely controlling the invocation of certain objects at certain times may not be sufficient. Instead, the system needs a more differentiated notion of what is going on at the moment. The mechanism developed in this thesis is a *Task-based* mechanism.

4.6.1 Definition of Tasks

As a central abstraction for the development and usage of Groupware Components, this thesis suggests the use of *Tasks*.

The relationship between Component, Object and Task is constrained using the following definitions:

- A *Groupware Component* is used to perform a *Task* on an *Object*.
- A Groupware Component can publish several different tasks on different Object classes.
- Each Task has an identifiable name.
- For a given Object and Task, the required Groupware Component is uniquely identifiable.

Tasks - a formal definition

Formally, a Task is defined as a triple $T = \{TaskID, Class, ComponentClass\}$ where

- *TaskID* is an identifiable ID,
- *Class* identifies an Object class in the system,
- *ComponentClass* identifies the class which implements the component publishing this Task.

A *Task signature* is formed by the Task's name and its argument class, in the same way that a method signature is formed by the method identifier and its parameter types. In order to denote that a Task is "applied to" or performed on an object of a given class, the Task signature is noted using arrow notation. The signature of a Task "edit" which can be performed upon an object of type Document is noted as $edit \rightarrow Document$.

Task polymorphism, inheritance and overriding

Tasks conform to the following *polymorphism relation*:

Task bindings are *polymorph* with regard to the inheritance relations in the object-oriented system used to develop the components: If component C1 publishes a Task T1 on an Object Class A and B is a subclass, then Task T1 also applies to all instances of class B.

Tasks are inherited by components which are derived from other components. If component C2 extends component C1 by inheritance, it inherits all Tasks published by component C1. It can itself of course define additional Tasks.

Tasks can be overridden in subclasses. Tasks in derived components override Tasks from their super-components if the Tasks have the same TasksIDs and apply to the same (or polymorphic) classes.

The inheritance and overriding rules can best be explained using the inheritance graph shown in figure 4.9, which shows one inheritance graph for components (derived from the component base class) and one for the corresponding Model classes to which the Tasks published by the components can be applied. Here, component B, which inherits from component A overrides the Task *edit*

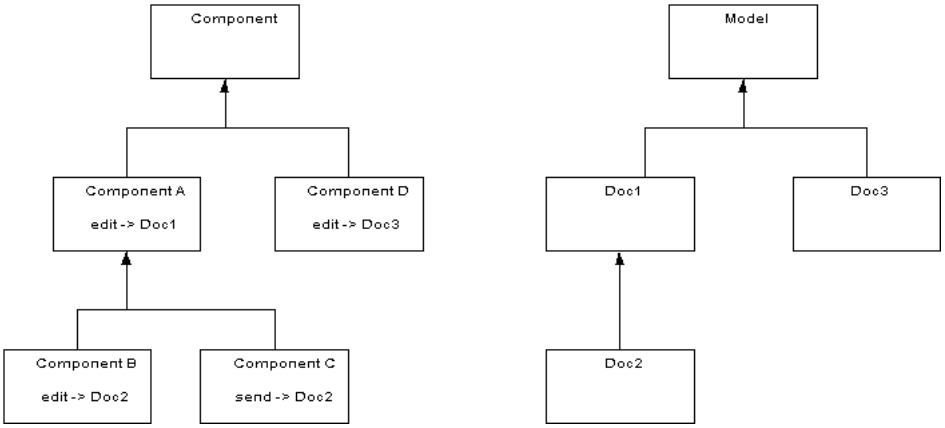

Figure 4.9: Sample Task and Component Inheritance

→ *Doc1* inherited from component B with the Task *edit* → *Doc2*. Since Doc2 is polymorph to Doc1, this overriding does not actually change the applicability of Tasks. Performing Task *edit* on an instance of Doc2 will be handled by component B. component C extends the set of published Tasks with the Task *send* → *Doc2*. This means that the Task send cannot be performed on instances of any other class than Doc2. The Task definition *edit* → *Doc3* by component D is entirely independent of the other Task definitions, since neither component D is related to component A by inheritance, nor is Doc3 related by inheritance to Doc1.

Uniqueness constraint

Tasks in the system are required to be unique in the same way that methods defined in an object-oriented class implementation are required to be unique. A component may publish multiple Tasks with the same TaskID, but it may not publish multiple Tasks with the same Task signature. Tasks with the same signature which are inherited from other components are considered to be overriden, according to the previous explanations, thereby maintaining the uniqueness constraint.

4.6.2 Task Terminology

For the following discussions of Tasks and their use within the DyCE system, it is necessary to establish some terminology.

Publishing Tasks: A task is said to be *published* by a component. Publishing a task means that a component provides the information that it can be used to perform a specific task on a certain Object class. In a discovery process (either at server startup or when a component is checked in), the Component Broker queries each component about the tasks it publishes. For this, the component must provide a method `giveTaskBindings(Vector tasks)`, using the parameter tasks as a *collecting parameter* for gathering up all tasks published by the component.

Component Lookup: The process of Component Lookup retrieves the component defined for a given Task on a given Object class; i.e. lookup(Task, ObjectClass) returns the ComponentClass identifiers from those Task tuples Ti where Ti.Class = ObjectClass and Ti.TaskID = Task.

Performing Tasks: Performing a task is similar to issuing a method call to a runtime Object, using another Object as parameter. Performing a Task on an Object yields the component which has *published* the Task, using Component Lookup as defined above.

4.6.3 Tasks as bindings between Components

Tasks are the central (and only) binding mechanism between the Groupware Components in the system. This means that one component instance does not directly invoke another; instead, the component performs a Task on an Object using the runtime system. This Task then yields a component which can be instantiated on the Object.

In the course of this thesis and the related implementation work, it was found that this abstraction of components interacting by performing Tasks on Objects is a very powerful one. It allows complex applications to be assembled using loosely coupled components. Section 5.5 will describe how the task-based programming model is used as a basis for end-user support for combining existing components into new "configurations", which can be deployed through the Component Broker and can be used collaboratively.

4.6.4 Tasks and Reflexive Programming

According to the previous set of definitions, the Tasks provided by a Groupware Component can be likened to methods provided by an Object in an object-oriented system (or the interface methods specified in a CORBA Interface Definition). A task is an operation that can be performed by a class instance and which takes a typed parameter (the shared data object on which the operation can be performed).

The set of Tasks published by a component is determined at runtime, when a new component is entered into the component Repository, or at start-up time, when the server starts.

The publishing of tasks and the querying of task information by the runtime support system can be seen as an example of Reflexive Programming: Reflexive Programming mechanisms are mechanisms which allow an element of the information system to provide information about itself (e.g. an instance could provide the information which methods can be invoked on it) and to discover information about other elements of the program system (e.g. query an object passed as a method input parameter as to which operations are provided by it) (see [Dou95a]).

Several object-oriented development systems provide some means of reflexive programming. Java provides the Java Core Reflection API as part of the core language definition (since Java 2, or JDK 1.2).

The goal of specifying the Task model as an additional reflexive development mechanism is to provide a common programming abstraction for Groupware Components which can be realized in a general support framework, without resorting to the reflection mechanisms of a specific programing language.

In this way, Tasks become first class objects and can be used to control the collaborative system. As will be seen later in the thesis, Tasks can be used to define collaboration sessions or to define workflow-like processes.

4.6.5 Mapping tasks, appliances and users

As has been discussed, task information is provided by component implementations by publishing a set of tasks through the method `giveTaskBindings(Vector tasklist)`. The tasks that are published are instances of the framework class `ObjectComponentTask`, which provides a mapping between a task identifier, a component class name and a domain data model class name.

An important aspect of the DyCE component framework is the support for flexibly selecting components based not only on data models and tasks but also on user and infrastructure information. This selection mechanism is used, e.g., when deploying components for devices with specific characteristics, such as mobile hand-held devices or pen-based devices. This flexible selection mechanism has been introduced into the DyCE framework in order to address requiremenets RU5 and RU9.

Each client logged into the server is uniquely identified on the server by a `userClientTuple` instance (see figure 4.12) and an associated `ClientObject` instance. The ClientObject instance is created by the client application and is transferred to the server in the log-in process. The `ClientObject` instance can be queried about client type identifier (an identifying value relating to the type of device[4]).

In order to distinguish between different tasks for different users, different devices, etc., the task object can make use of this information. When publishing a specific task which only applies to a certain device or a certain user, the component developer can implement a subclass of `ObjectComponentTask`, overriding the framework method `boolean appliesTo(userClientTuple user)` to provide distinguishing functionality (the framework base implementation simply always returns TRUE, i.e. by default tasks always apply) and return an instance of that class when returning the task bindings. For example, in order to publish a task which only applies to hand-held devices of the "PDA" type, the `ObjectComponentTask` subclass can access the `ClientObject` instance and check whether the endpoint type of that client is "PDA". If so, the `appliesTo` method would return TRUE, otherwise it would return FALSE.

When the server's Component Broker accesses the set of published tasks in order to find a component to open for a specific task on behalf of a certain user on a certain client, it checks the task rules as presented before, i.e. it compares the domain data model class information (also taking into account object-oriented inheritance rules), and compares the task identifier. Once a potentially applicable task has been discovered, the task's `appliesTo` method is called to check whether the task is applicable in the current situation. If this call returns FALSE, the task does not apply, otherwise it is added to the set of matching tasks. If at the end of this process several matching tasks

[4]Currently defined are the identifiers STANDARD (a regular PC or other computer), PDA (a hand-held device with limited display size and possibly fewer displayable colours) and APPLET (denoting that the end-point is running the client application as a Web Applet, regardless of specific machine type). Additional identifiers can be added as more devices with specific distinguishing characteristics need to be supported

have been found, preference is given to those tasks which are published not as instances of `ObjectComponentTask` but rather as instances of a subclass thereof. The rationale behind this is that a component which publishes a task which *specifically* applies to a certain end-point or user, is better suited for the current situation than a component which publishes a task which simply applies by default.

4.6.6 UML diagram extensions for modeling tasks

Section 4.2.9 presented extensions to UML class diagrams for noting shared object diagram elements. It did so by using the stereotype mechanism available in UML to specify stereotypes for a couple of design elements. In order to be able to model component-based groupware built with the DyCE framework, it is also necessary to provide the means for representing the elements of the task model in the appropriate diagrams. In the case of the task model, we are faced with two views of the system: The *static* view represents instances of the model as presented above, with component implementations publishing tasks on shared model classes. Additionally, there is a *dynamic* view which needs to be modeled. This view corresponds to the use of components in a system which is actually being used, where specific component instances are bound to specific instances of the shared object model. This view is later relevant when representing sessions in the running system.

The static view of the task model is presented as an extension of UML's class diagrams, just like the extension for the shared data model. Here, the diagram is extended to model the tasks published on shared object classes.

Figure 4.10 shows a static model for components interacting on a shared structured document. Two Groupware Components, a document editor and a document structure viewer publish tasks on shared object class representing a (very simple) structured document. As previously defined, the shared model object is extended with the stereotypes ≪Shared≫ and ≪Slot≫ and the component class is extended with the stereotype ≪Component≫. The tasks published by a component class are represented as extensions of the class diagram element "dependency" (the dashed arrow), extended with the stereotype ≪Task≫. The task link can only be drawn *from* classes of the ≪Component≫ *to* classes with the ≪Shared≫ stereotype. Since each component class implementation needs to provide a method `giveTaskBindings(Vector tasks)` for querying the set of tasks published by this component, these methods are modeled in the diagram as well.

4.6.7 Task Model - Summary

The Task model used as the basis of the component-based groupware development system described in this thesis is summarized in figure 4.11. This figure shows the conceptual elements involved in the Task model, their relations and the respective cardinalities.

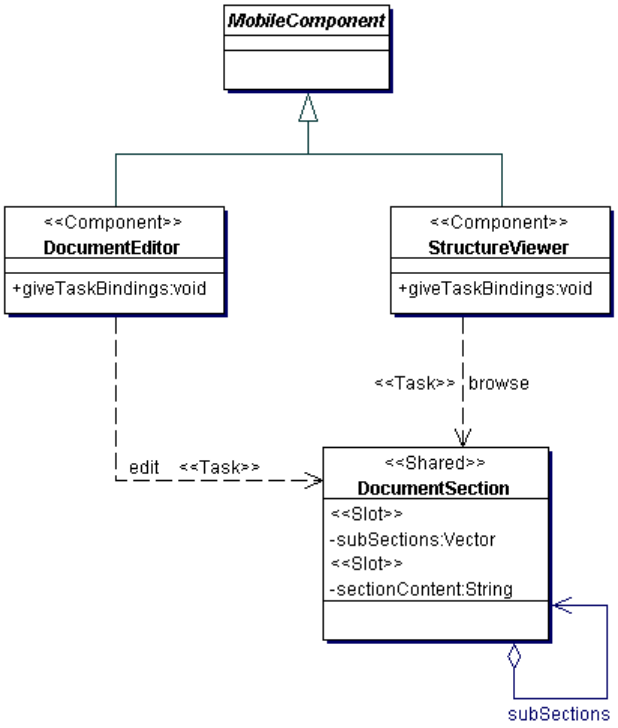

Figure 4.10: UML diagram stereotypes for elements of the static task model

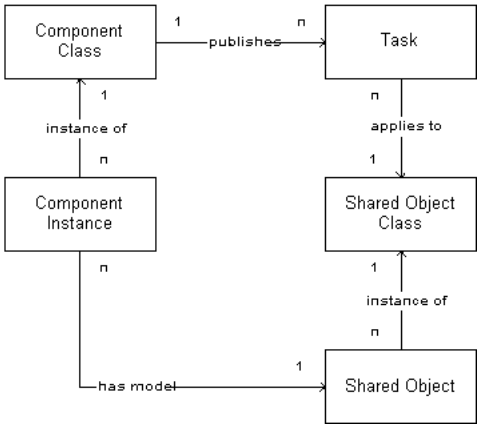

Figure 4.11: Task Model

4.7 Session Management

A *session* is the concept used to model a group of users working together. Since session information needs to be replicated throughout the system, a session object in the system is modeled as a DyCE replicated object. A session maps a set of users onto a set of `TaskObjectTuples`, which in turn relate a task to an element of the shared object space. The basic session model is shown in the diagram in figure 4.12.

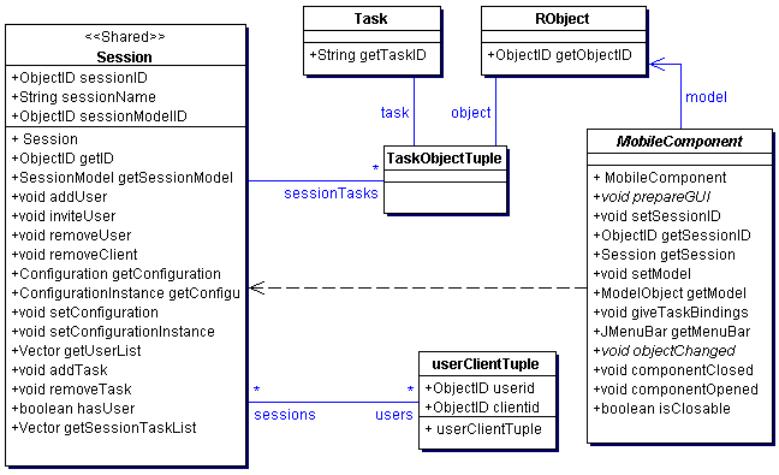

Figure 4.12: DyCE Session information model

4.7.1 Session Support

As can be seen from the diagram, a session groups together tasks which are performed on RObject instances (modeled as `TaskObjectTuple` instances) by all users currently in the session. Since the same user can be logged in at multiple clients (running multiple instances of the client on one or more machines), users are represented as tuples containing not only the information about the user, but also about the specific client to which this session item refers. This way, a user can be in different sessions on different clients at the same time, e.g. when using a mobile device in addition to his desktop machine.

The session management system (which is provided as part of the DyCE framework and which is available on the server as well as on each client), provides a number of operations to control session membership:

Joining a session (`joinSession(userClientTuple user)`): The user is added to the session. The session's user list is updated and the session information is replicated to all currently active clients. If the added user is currently logged in to the system, the session is opened on his client machine.

Opening a session: When opening a session on a user's client, the initial set of `RObjects`, linked to the session through the `TaskObjectTuples`

is replicated to the client machine. The tasks stored in the session information are then performed on the shared objects, resulting in loading and opening of the component which published the given task. On the user's client machine, a session window is opened, which contains the resulting components. When a user logs into the client, any sessions which he or she belongs to are retrieved and automatically opened. This is a way to support resuming running collaborations.

Leaving a session (`leaveSession(userClientTuple user)`): The user at the given client machine is removed from the session's user list. The session window on the client's machine is closed. Should the session contain no more users, it is removed altogether.

Performing a task (`performTask (Task t, RObject o)`): Performing a task within a session adds the corresponding `TaskObjectTuple` to the session. The task is automatically performed on all active users' clients by the client's `SessionManager`. The related `RObject` is replicated and the Component Broker is used to determine the appropriate Groupware Component. This component is then opened in the session window. The users can now perform the task at hand.

Removing a task (`removeTask (TaskObjectTuple t)`): Removing a task removes the given `TaskObjectTuple` from the session's list of tasks. The client's `SessionManager` closes the component which has been opened for this task in the session window. The remaining components are not affected by this operation. Should the session contain no more tasks, it is removed altogether.

Note that the session contains no direct component information but only elements from the task model. Using this information, the necessary component information can be fetched at runtime. This way, a session can be suspended on one system (with a certain set of components) and can be resumed on another system (which could potentially yield a different set of components). Also, different users can open the same session and receive a different set of components. These design choices have been made to support a flexible collaboration environment within which users can collaborate with a variety of different components.

It is also important to note that any `RObject` can be referenced through the tasks performed in several sessions. Components in different sessions which are invoked on the same `RObject` are, in effect, coupled: They reflect the current state of the shared object, regardless of the session from which it was changed. Additional tasks invoked on the object within a session, though, are shared only within that session, even though they may reference objects shared in multiple sessions.

Early usage experiences have shown that, depending on the realization of the components, such parallel modifications of shared objects by users who are not "in my session" seemed strange to some users. In the current model, sessions do not contain any notion of access control or locking. As the user experiences show, such a concept may need to be introduced in the future, along with restrictions on the parallel use of shared objects in multiple sessions by different groups of users.

4.7.2 Sessions and Group Awareness

The previous discussion can be taken as an illustration of the need for providing group awareness information. The session information can be used to provide a number of group awareness elements:

- Which users are in the same session? This information shows which users will receive instances of components which are newly opened within the session. A simple user list within an open session window could be sufficient to display this information.

- Which users are currently accessing the same model object? This information can be used to find out which users would be affected by changes made to a shared data item. A reasonable presentation of this information can be used to alleviate the previous experience that users of the running system were confused as to which other users were currently accessing (and potentially modifying) a shared object.

- Which tasks are currently being performed on a shared data item which is being seen in this session? This can be taken to show the complexity and diversity of the collaborative setting.

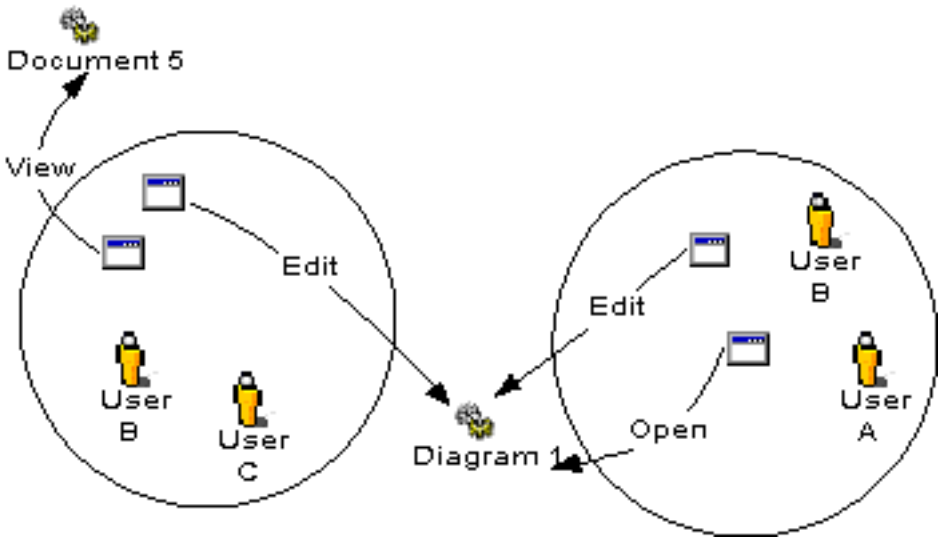

Figure 4.13: Suggestion for a session-based group awareness view

Additional group awareness (e.g. the plans of other users) cannot easily be deduced from the session model information. The above information could be displayed to the users in a "session map" (see figure 4.13, similar to a fish-eye view, which would always show the model object of the currently focused component (i.e. the one having input focus in the window management system) in the center and the related information about users, sessions and tasks surrounding it.

4.7.3 UML diagrams for dynamic session models

In order to describe collaboration situations, which always take place in sessions, dynamic session model diagrams are used to model concrete running sessions, with potentially multiple components actually being used on shared objects. These diagrams need to illustrate the following concepts: A number of component instances (at least one) access a shared object instance (i.e. the component has the shared object instance as its model). The components were invoked through specific tasks. These concepts are shown in the diagram elements in figure 4.14.

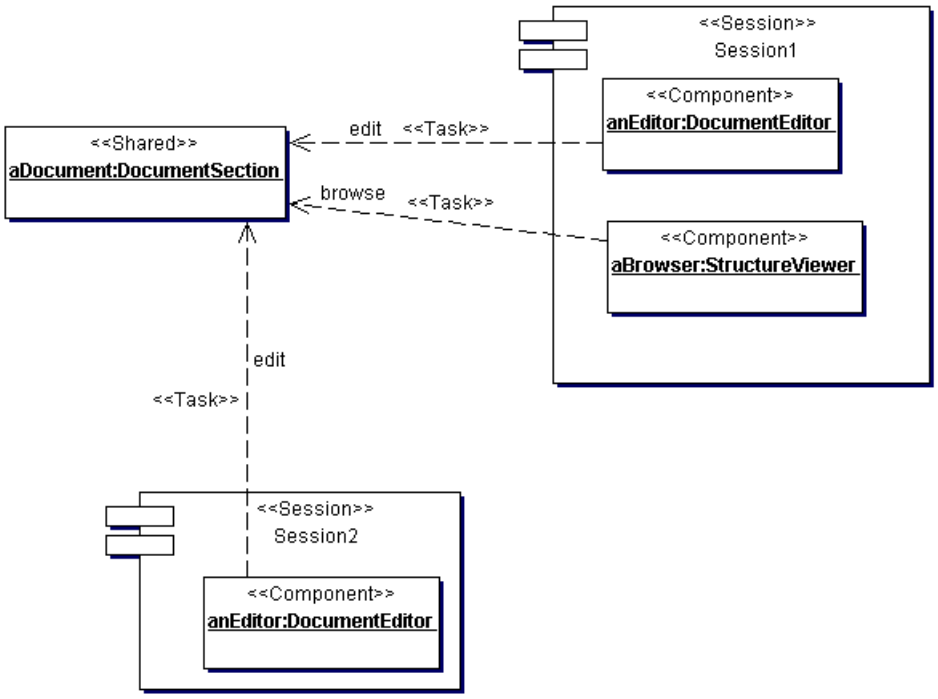

Figure 4.14: UML diagram stereotypes for running sessions

The session model diagrams are extensions of the UML diagram type "deployment diagram". Deployment diagrams model how the elements of the running system are distributed and how the collaborate. In this extended diagram type, the previously defined UML stereotype extensions are reused. Additionally, sessions are modeled as extensions of the diagram element "Component", using the Stereotype ≪Session≫, as a diagram element which can include parts of the running system. In the case of a session model diagram, the session items hold component instances which have been invoked on an element of the shared object model through a specific task. The task is shown on the "dependency" link, extended with the ≪Task≫ stereotype.

The ≪Component≫ elements, instances of Groupware Component classes, are each bound to a specific instance from the replicated shared data model.

This relationship is implied in the task relationship between the component instance and the shared model instance, since the instance on which a component is invoked through a task automatically becomes that component's data model.

Note that the session element in the diagram does not give any indication about the fact on how many systems the session is currently open. If this information needs to be modeled, it can be shown (see figure 4.15) using the deployment diagram's "node" element (the boxes in the figure) to represent a specific node of the system on which a session is being used.

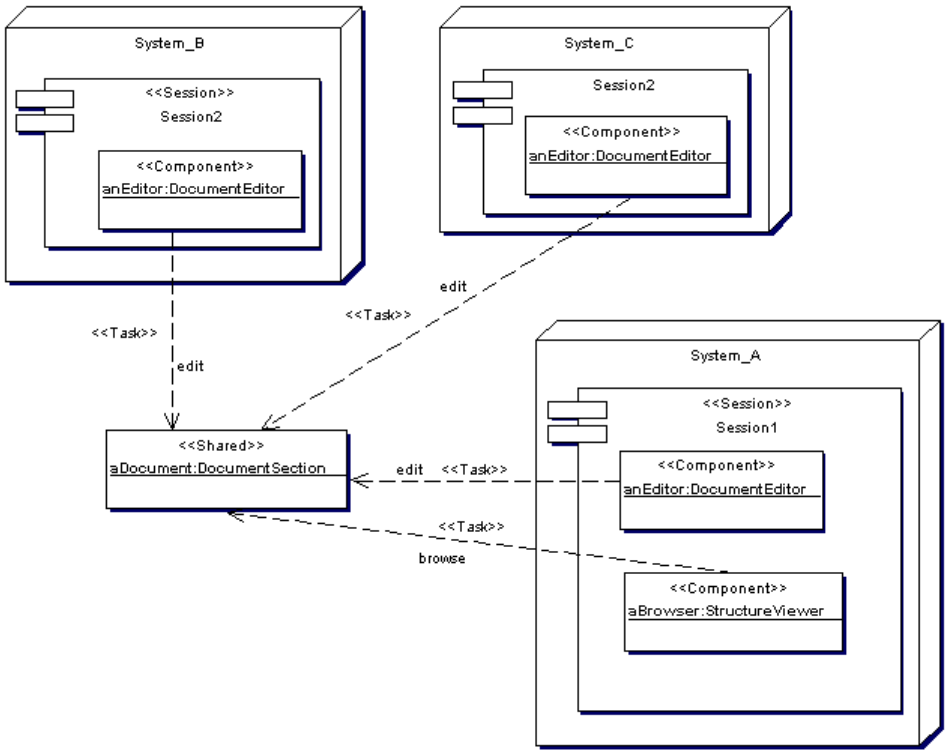

Figure 4.15: UML diagram stereotypes for running components on specific nodes

Even when modeling the specific nodes on which the sessions are running, the shared object model instances remain outside of the node element (and each is only shown once), since the shared object instances are automatically replicated to those nodes which require the object instances and the components in the various sessions are, conceptually, only interacting with a single model instance.

4.8 Server-Based Components

In complex collaborative processes, there are a number of operations which need to be performed only once and which should be performed centrally (c.f. requirement RD9).

As one example for such a centralized "one-off" operation, consider the enactment engine in a distributed cooperative workflow setting. A number of clients share a common workflow model within which they are collaborating. The enactment of such a workflow, i.e. the switching between process steps, the propagation of information elements, is an activity which is needed to coordinate *all* clients and their activities. Hence, it is not directly attributable to one specific client. Also, the enactment engine of a distributed workflow process should obviously also run when no clients are currently connected (e.g. in order to trigger time-dependent actions).

It follows that such an engine should not be realized as a client-side functionality. Rather, it should be server-based and execute independently of (but interacting with) the current state of the clients.

Other examples for such server-side functionality include monitoring centralized resources, performing scheduling or processing tasks, etc.

In order to support server-side components a specific subclass of `MobileComponent` is used as superclass for all server-side components. Server-side components publish tasks just like all components and they also provide a user interface in the same way as all other components. When a task invoked on a replicated object yields a subclass of `ServerComponent`, this component is instantiated on the server. The component has full access to the shared data objects and can modify them as needed. A Workflow Engine could, e.g., monitor the system time and the current workflow model, advance tasks or add time-dependent to-do items to a user's to-do list. Figure 4.16 shows such a configuration, using the UML extensions described in this thesis. A Workflow Engine server-side component is running on the server, invoked on the shared workflow data through its task "enact". The users' sessions include WorkflowEditor components, used to view and interact with the shared workflow data (and in this way with the WorkflowEngine component). Note that no specific UML extensions are proposed for server-side components, since this fact can be indicated by placing the session containing the server-side component into a "Node" diagram element depicting the server.

Since sometimes even server-based features need user intervention, server-side components can also provide a user interface. This can be as simple as displaying current status information, but it could also provide users with access to the server with more elaborate interaction means. An instantiated server-based component's user interface is dynamically added to the server's UI through the use of tiled tabs.

In order to support server-side components, session management (see section 4.7) needed to be extended slightly, so that a server-based component is only instantiated once, even though the session is entered many times by many different users.

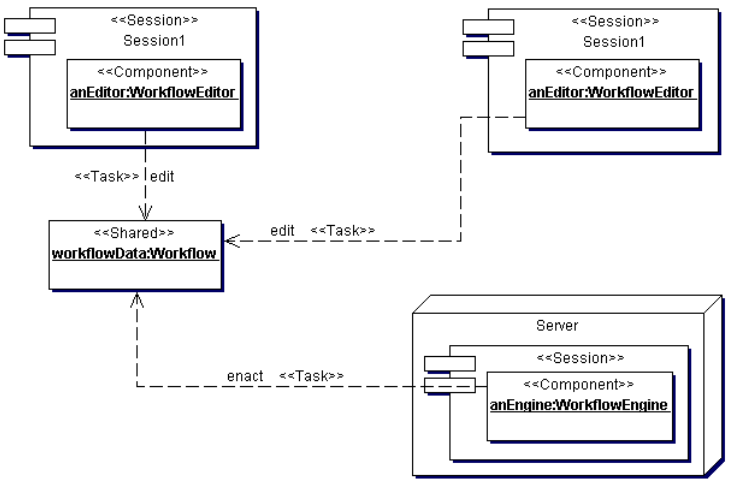

Figure 4.16: Server-Side component example

4.9 Help to the end-user when tailoring

The DyCE framework, with its component-based approach to constructing groupware environments, provides users with the means to extend and tailor their work environment. According to [Mor97] (as cited in [KT99]), three levels of system tailoring can be distinguished:

- **customization** - selecting among a set of predefined configuration options,
- **integration** - linking together predefined components within or between (an) application(s), and
- **extension** - improving the implementation by adding new program code.

The components in the DyCE framework are used to allow users to perform tailoring of the second type: Users can introduce additional tools into the system, invoke various tasks on shared data objects and combine components into configurations (see section 5.5). Such an environment needs support for allowing users to become familiar with the system, to learn about the available components and to receive guidance from the system.

In the DyCE framework, this help comes in several ways. For each shared object used within the system, the users can use simple GUI operations to receive a list of tasks which can be performed on this shared object. Since each task contains a readable name, this list is assumed to be a readable list of options available to the user.

Additionally, each component is expected to provide a help page (or pages) in HTML format. This help, which can contain figures, external references, etc., needs to be accessible through a standard mechanism (i.e. each component's help file is expected to be called `help.html` and reside in the component directory). It is packaged with the component when deploying the component.

The user can access this help file within the client desktop with a "help" menu option in the component window and in the component browser (a desktop palette showing a list of all available components). In this way, components are expected to be sufficiently documented in order to support the users' learning process.

An additional approach, which is also applied in [WHRT00b] and [WHRT00a] is to learn the use of the different components from peers working within the same system. By working together with other users on shared objects using components which are linked to other components, users are expected to develop an understanding of the possibilities offered by the groupware system, to learn *from* and *with* the other users of the system.

4.10 System Architecture

Following the general concepts of Groupware Components and the programming support provided for them, the system architecture and system design will now be presented in incremental steps, each highlighting an important aspect of the overall system.

The conceptual system architecture is shown in Figure 4.17, which depicts a client machine connected to a server over the network. In the following subsections, a brief overview over the functional modules of this architecture will be presented, before the individual functional modules are specified in more detail in subsequent sections.

4.10.1 Server Architecture

The server provides the services which the clients use in order to provide the collaboration support. In keeping with the CORBA way of describing a distributed system, the services are provided by service objects which implement a predefined interface. The service interfaces of the server modules are included in the relevant subsections of the system design.

The server includes the following modules:

Component Broker: Maintains the server side of the component-based architecture. Manages the available components, stores and retrieves them using the Component Broker and distributes them over the network. Since there is a close relation between the interactive components and the objects created and manipulated by them, the Component Broker is closely tied in with the Object Management Service. The details of the Component Broker and the components managed by it will be presented in subsequent sections.

Component Storage: Used for persistent storage of the components. Components are fetched from the component storage when requested by a client. The component storage stores the components in executable "parcels" which can be transmitted across the network and executed at the client node.

Object Manager: Manages the shared data objects at runtime. The object management service is accessed by the clients when fetching objects, creating new objects, etc. This service also uses the Object Storage to handle the persistent storage of objects.

Object Storage: Stores the data objects persistently. Typically this task is performed by a database system, providing an appropriate retrieval and query

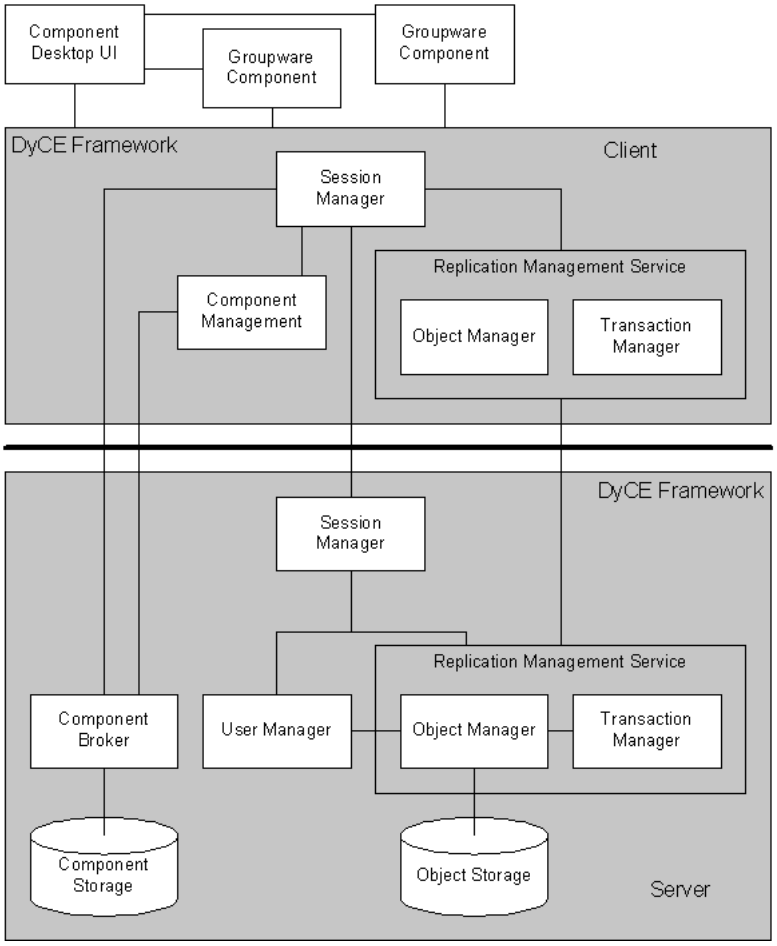

Figure 4.17: System Architecture

interface.

Replication Management Service: This service is used to control replica distribution, replica groups and data consistency. The Replication Management Service keeps track of which objects are replicated at which client nodes.

Transaction Manager: The server side of the transaction management. In order to provide object consistency, the system uses a transaction-based approach: Clients perform transactions on the shared objects. These transactions are validated and distributed by the server's Transaction Manager. The Transaction Manager needs to interact with the Object Manager and the Session Manager in order to modify the server's object replicas and distribute the transaction information to the correct sites.

Session Manager: Manages the running sessions. Offers service interface to add users to a session and to remove them from sessions, to create new sessions and to perform tasks within sessions.

User Manager: Manages the user entries in the system. Interacts with the Session Manager to manage the user lists of sessions; holds and makes available

information about the running system: Which users are currently logged in at which clients?

4.10.2 Client Architecture

Each client system has a user interface, consisting of the component instances which provide the application functionality. In addition to this, the user has access to a "Component Desktop User Interface" which provides access to the available components.

The components at the user interface access the replicated data elements through the use of an Object Manager, which encapsulates the object pool and controls the components' accesses to the objects. Closely related to the Object Manager is the Replication Management Service, which keeps track of the replicated objects and the current replication situation (e.g. which objects are also replicated at which other nodes of the system, etc.).

The Replication Manager is tied in with the Replication Management Service as well as the Object Management Service on the server, since any object which resides on a client node also resides on the server and is therefore automatically a replicated object (while the inverse does not hold true - an object on the server need not always be currently used at a client node, hence it need not always be replicated).

The Component Management sub-system provides support for component access and component execution. On the client, the Component Desktop User Interface interacts with the Component Management sub-system to access component information and retrieve component implementations from the server's Component Broker.

The Component Manager is the client of the server's Component Management Service, it is able to query the available components, and retrieve components. It is also responsible for propagating notifications to the user interface when the set of available components changes. Also, the Component Manager keeps track of which components are currently loaded on the client system. The Component Manager is related to the Replication Manager, since the information from both is required when a new object is replicated which needs a specific component to be displayed or manipulated.

The Component Execution module, as the name would imply, is the basis for executing the components once they have been fetched through the Component Manager. It keeps track of which components are currently executing on which shared objects in which sessions.

The Component Desktop UI is the user interface built on top of the Component Management and Component Execution as a starter application. This is the application part with which users interact before tackling their actual tasks using the Groupware Components -in this way the Component Desktop UI is comparable to an operating system's GUI desktop application.

4.11 Groupware Components - Summary

Groupware Components form the basis of the component-based groupware framework DyCE. Using the framework, developers can implement new Groupware Components, which can be deployed to users through a Component Broker.

Groupware Components are collaboration-aware; coupling of components is realized using the replicated data-sharing approach: The shared data models are developed based on the support provided by the framework. Shared data objects and component implementations persistently stored in a server and are available through a client application (req. RU1). New data objects can automatically be dynamically replicated. Concurrent access to and modification of shared data objects is handled by a transaction-based concurrency control mechanism, which provides support for highly interactive groupware applications by allowing the users to continue with their work while their operations (the transactions) are still being validated. Avoiding the use of object locking increases the potential for conflicting operations (leading to undoing users' actions at a later point in time) but permits maximum parallel activity.

The shared data model implementations are developed separately from the components, are external to the components and are attached to the components at runtime. This allows coupling different components on the same data model (fulfilling requirement RU8). The mapping between shared data model implementations and the component(s) used to interact with these data elements is provided by the central programming model, introduced in this chapter: task-based programming. This reflective approach allows the Component Broker to query a component's tasks and allows runtime identification of the components required for a specific task to be performed on a shared data object (fulfilling requirement RU2).

Tasks being performed on shared objects form the basis of the framework's session management. New objects and new tasks can be introduced into running sessions and additional users can be invited into sessions. The entire session model is also based on DyCE's object replication mechanisms and is therefore automatically shared in the distributed system. Using the information about tasks and objects from the sessions, coarse-grained group awareness (i.e. who is working on which objects in which session(s)?) can be provided (req. RU3). More fine-grained group awareness is highly dependent on component user interfaces and interaction semantics. Ubiquitous access to the collaboration infrastructure (RU5) and mobile work (RU9) are supported by the device-specific extension features of the Task model (see section 4.6.5). Support for multiple simultaneous collaboration modes (RU6) is provided by the session model, which is based on the task information provided by the components. Components can be used in individual or group sessions, an individual session can be transformed into a group session (making the transition from individual to collaborative work) by inviting other users into the session.

Using Groupware Components, flexible groupware environments can be constructed. Components are deployed to all other users through the Component Broker, making the component-based collaborative system easily extensible (RU7). Components can be dynamically invoked either by other components or by user interaction. In this way, the users' collaboration environment can evolve over time as the content of their work evolves. End-User tailoring of the environment is supported by providing users with the ability to combine components into configurations, which can be deployed through the central server and can be shared with other users.

The object-oriented framework provided for implementing the domain data model and the Groupware Components, along with the UML extensions proposed in this chapter support the developers' reuse of existing programming

knowledge (RD1). Reusable shared data models (RD2) can be developed on the basis of the `ModelObject` section of the DyCE framework and can be used in several components due to the dynamic task binding between the components and the domain data models. The use of specific framework classes for creating the domain data model allows transparent replication of the data elements which are to be replicated (RD3); non-shared data items (RD4) can be implemented using the standard programming language mechanisms (i.e. without using the framework support provided by the `ModelObject` and `RObject` classes). Access to collaboration information (RD5) can be performed through the interface methods of the `Session` class instance, which each opened Groupware Component can reference. The deployment of newly developed components (RD6) is the functionality provided by the Component Broker. Server-side components (RD9) are provided as a specific feature of the component framework (see section 4.8). The remaining developer requirements are implementation-specific and are addressed in the concrete implementation, which is described in the next chapter.

Chapter 5

System Implementation

The concept of Groupware Components has been implemented in Java in the DyCE (Dynamic Collaboration Environment) development framework. This chapter will present the central implementation details of the different services and architectural components included in the DyCE system design presented in the previous chapter. Experiences from the implementation of the DyCE framework and sample components will be discussed.

5.1 DyCE System Architecture

The system architecture of DyCE can be seen in figure 5.1. Implementation details of most of the architectural modules will be presented in the following sections.

The DyCE framework realizes collaboration in the form of a distributed system, with multiple clients connecting to a single server. The server consists of the following modules with the following responsibilities:

- **Session Manager:** Implementation of the Session Management Service (see section 4.7).
- **Component Broker:** Maintains the Component Repository, manages and provides access to the registered Groupware Components.
- **HTTP Server:** Used for downloading Groupware Component implementations to the client. Also used for integration of DyCE collaboration tools into Web-based environments. Maintains HTML document base for use in Groupware Components (e.g. as help files).
- **Object Manager:** Maintains the persistent Object Storage. Manages shared objects (elements of the Shared Data Model level, instances of `RObject` and `Slot`).
- **Transaction Manager:** Performs transaction validation and concurrency control, as well as distribution of validated transactions to the other connected clients.
- **ComponentBrokerServer:** Mediates between the clients and the various server modules accessible through RMI (Remote Method Invocation,

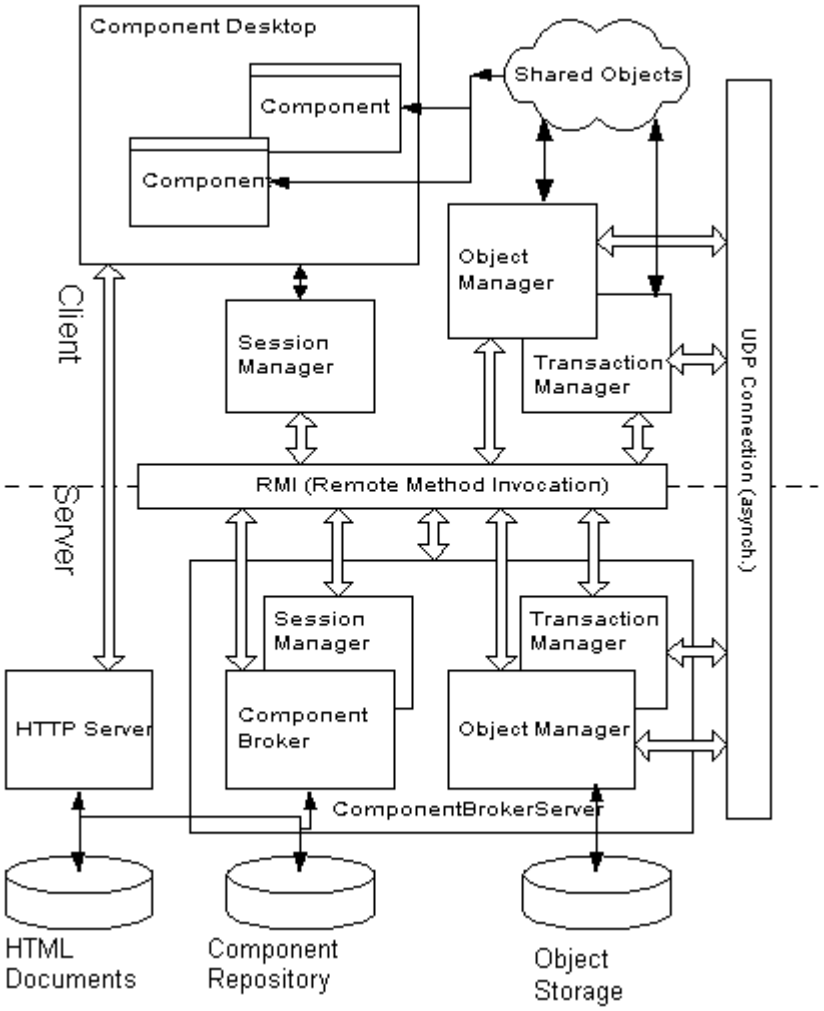

Figure 5.1: System Architecture

see section 5.2.1). Creates and publishes the RMI server modules and performs lookup operations on behalf of the clients.

The clients each contain the following architectural modules:

- **Component Desktop:** Client application GUI for accessing Groupware Components.
- **Session Manager:** Client-side module of the distributed session management service. Maintains the list of sessions active on the client. Creates new sessions, joins and leaves sessions by interacting with the server-side session management.
- **Object Manager:** Client-side module of the distributed Object Management system. Maintains the set of replicatable objects currently replicated

to the client (**Shared Objects** in the architecture diagram). Retrieves object from the server when needed.

- **Transaction Manager:** Client-side module of the Transaction Management system. Creates transactions, maintains transaction queues and sends transactions to the server for validation. Received transaction validation responses and interacts with the Object Manager to maintain consistent object state.

5.2 Communication in DyCE

Since the resulting system described in this thesis is a distributed co-operative system, attention has to be paid to the underlying communication mechanism used to link the system. There are a number of different options available as a basis for communication in distributed systems, each with its distinct advantages and disadvantages. Client/Server or Server/Client Communication in DyCE can be categorized according to the different requirements of timeliness of the communication. *Synchronous* communication is used for those communications between systems which require a direct response without which processing cannot proceed (i.e. the initiator of the communication is suspended until the response is available), such as a part of a calculation that needs to be performed on a different host and is required before the entire calculation can be continued. An example for a communication channel fulfilling the requirements of synchronous communication is RPC (Remote Procedure Call). This form of communication is often also called "in-process" communication, since the process context or flow of control seems to pass through the synchronous call from the sender to the recipient and back again. *Asynchronous* communication between client and server (in either direction) is used for communication items which can be likened to message delivery: The sender creates a message and sends it on its way to the recipient. It is the responsibility of the underlying support framework to ensure that the message reaches the recipient. The sender can proceed, without waiting until the message actually reaches the receiver. In this form of communication, it is not of interest to the sender that the message reaches the client immediately, or that an immediate response is available. Asynchronous communication can be compared to sending a message (and receiving a response) through email. This form of communication is also referred to as "out-of-process".

It is important to note the dual use of the terms synchronous and asynchronous in the course of this thesis. On the one hand, we can distinguish between synchronous and asynchronous collaboration (relating to the perceived immediacy of response from the other user and to the time bounds within which changes made to a shared artifact at one site are observed at other sites); On the other hand we distinguish between synchronous and asynchronous communication in a distributed system, as described above. These two issues are not entirely independent, but there is no direct relation in the form that synchronous collaboration necessitates synchronous communication. The one use of synchronous/asynchronous relates to a quality of the overall system, as perceived by its users. The other use relates to implementation-specific details which influence the system design but not the user functionality.

5.2.1 Java RMI

Java RMI (Remote Method Invocation) is the Java standard for communication in object-oriented distributed systems. Similar to the CORBA (Common Object Request Broker Architecture) distributed object architecture, RMI publishes objects' interface methods and allows locating objects and invoking their methods over the network. RMI can be likened to a object-oriented RPC, which, unlike CORBA, is not language-independent. DyCE uses Java RMI for synchronous communication between the client and the server (in both directions).

Overview over Java RMI

In an RMI-based distributed system, a machine (typically a server) publishes a service object through a naming service. This service object conforms to a programming interface (i.e. provides a number of public methods which can be called) which is known to the client and which the client implementation uses to generate stub objects. These stub objects act as local proxies to the RMI service object, receive method calls from the client object and forward them over the network to the appropriate service object.

The following is an example for a simple RMI service object, providing a single service method for adding two numbers. The interface describing the service offered by the server object, `CalculationServiceInterface`, specifies the operations available to clients of the server object. The actual implementation of this interface, `CalculationServiceImpl`, is a class which implements the actual functionality required for servicing the incoming request (i.e. adding the two numbers passed as parameters and returning the result).

```
public interface CalculationServiceInterface
  extends java.rmi.Remote
{
  public int addTwoNumbers (int num1, int num2) throws
  RemoteException;
}

public class CalculationServiceImpl
  extends UnicastRemoteObject
  implements CalculationServiceInterface
{
  public int addTwoNumbers (int num1, int num2) throws
  RemoteException
  {
    return num1+num2;
  }

  public void publish()
  {
    Naming.rebind(localHostname+"/Calculate", this);
  }
}

public class CalculationClient
```

```
{
  public void doCalculation()
  {
    try
    {
      CalculationServiceInterface calculator;
      calculator = Naming.lookup (remoteHost+"/Calculate");
      if (calculator == null)
        return;
      int result = calculator.addTwoNumbers(5,4);
    }
    catch (Exception ex) { ex.printStackTrace(); }
  }
}
```

While this example appears trivial (and has been stripped of much of the required programming logic for illustrative purposes), it shows the main consistuents of the RMI object-oriented inter-process communication:

- A public interface (`CalculationServiceInterface`), which is available and known to all potential clients of a service.
- A service class (`CalculationServiceImpl`), implementing this interface (and potentially adding functionality which it needs to perform its operations but which is not part of the public service interface). An instance of this class (the service object) can be published through a naming interface, so that clients can look it up and invoke its operations.
- A client object (`CalculationClient`), which uses the RMI naming service to "discover" the location of the service object. For this, the client object requires a name (which can be fixed and well-known, like "Calculate" in the above example, or can be looked up from some other source). Using this name, it retrieves a reference to the service object and invokes the operations which are part of the service's public interface.

The RMI support layer has the responsibility of receiving the method calls, marshaling parameters and return values and invoking the correct object's method over the network.

Java RMI in DyCE

In DyCE, Java RMI is used for all synchronous communication between the client and the server (in both directions). The services published by the server are available through RMI invocations. Figure 5.2 shows the hierarchy of RMI interfaces and classes used in DyCE.

These services have the following responsibilities:

- **ComponentBrokerServer**: The central service object of the server's RMI structure. The ComponentBrokerServer manages the instances of all other public services (as described below) and is used to retrieve these service objects. The implementation decision has been made not to publish *all* RMI server objects through the naming interface (thus necessitating a

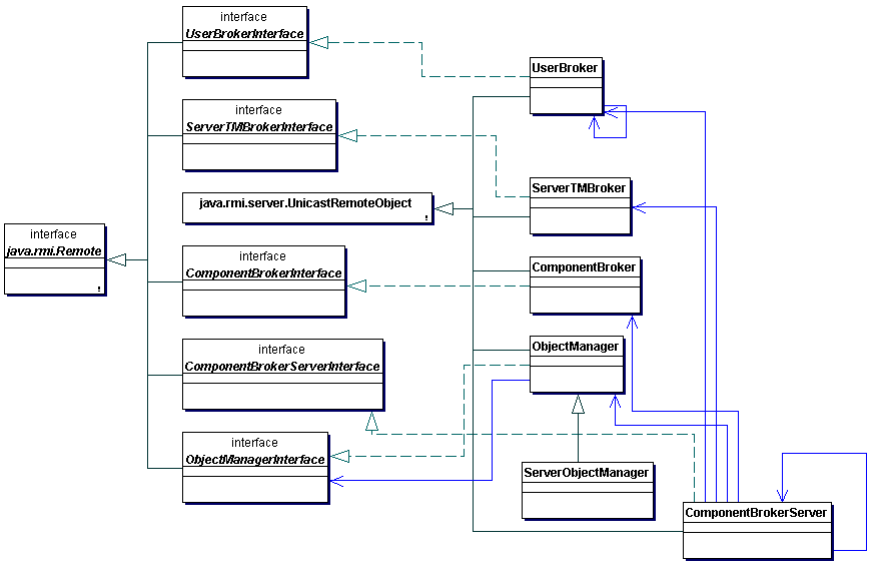

Figure 5.2: RMI Service object hierarchy

large number of public and well-known names which need to be resolved using the RMI naming service), but rather to publish only one instance of the ComponentBrokerServer on the server using a well-known RMI name and using this to retrieve references to all other RMI server objects. In this way, the set of available server objects can easily be extended (simply by adding them to the ComponentBrokerServer), without requiring more published RMI names (which would, in most cases, need to be hard-coded into the application). An additional large benefit of this RMI indirection is that we gain location transparency for the various server objects. The different RMI server objects, whose references are retrieved using methods of the ComponentBrokerServer, could be located on multiple machines on the network (e.g. the ObjectManager could reside on a machine with a large and fast database back-end, while the ComponentBroker could reside on a machine which is more suitable for component management). This distribution could be made transparent to the clients, since the methods of the ComponentBrokerServer could, before they return the RMI server references, do an additional step of lookup, locating the server objects on another machine, In this way, a client would retrieve from machine A a reference to a service on machine B and invoke that without actually having to deal with the fact that it is operating in such a complex network infrastructure. This support for location-independent service objects can be exploited in settings where performance is of great importance and where locating all service objects on one machine would be a performance impairment[1].

[1]Note that this service distribution, while realizable with very little changes to the current implementation has not been fully implemented in the DyCE framework; the discussions of system architecture, etc., in this thesis therefore apply to the case where all services are located

- **ServerSessionManager**: The server's session management service. Invocations from the client to this service are used to query session state, add sessions, etc.

- **ObjectManager**: The server's object management service. This service provides access to all shared data objects (instances of `RObject`) and makes these available to the clients. Synchronous invocations to the object manager are used to query object manager state, add new objects, retrieve objects when required, etc.

- **UserBroker**: Server-side service which manages all users known in the DyCE system. This service is used to retrieve user information, validate logins, etc.

- **ComponentBroker**: The service managing all Groupware Components registered in the DyCE server. The ComponentBroker manages the component implementations as well as the task binding information. Calls to the ComponentBroker are used to map tasks to component implementations, to register new components, etc.

- **ClientObject**: Callback RMI service object published by the clients to allow callbacks from the server to the clients. Usually, RMI communication is uni-directional, i.e. a client invokes a method on the server and receives a return value, but the server has no way of contacting the clients. By creating a ClientObject on the client and passing its reference to the server upon connecting, the server has a remote reference which it can use to invoke callbacks on the client.

- **ClientSessionManager**: The client side of the distributed session management. The server uses callbacks to this object to add new sessions on the client, query client session state, etc.

5.2.2 Asynchronous network communication layer

As has been described above, communication channels in a distributed system can be divided into synchronous and asynchronous communication. Synchronous communication in DyCE is performed over RMI. It is used for invocation of service objects to perform a variety of services needed in the system. The much larger part of communication (in terms of volume and frequency) in DyCE is the asynchronous communication part.

Asynchronous communication in DyCE is used as the basis for the object replication mechanisms, to maintain object consistency across sites. When accessing or modifying shared objects, the client creates transaction objects (see section 4.3.5). These transaction object include all read and write accesses to slots of shared objects. Once a transaction is committed, it is serialized and sent over an asynchronous communication channel to the server. The server receives the transaction, performs validation operations and sends the transaction and additional control objects to the connected clients, again using asynchronous communication.

---

on the one central server.

An asynchronous communication channel has been chosen for this communication, since the transaction updates do not require synchronous delivery (i.e. the client does not need to be suspended until the server has received the update transaction, the server does not need to be suspended until all clients have received the transactions, etc.). Quite the inverse holds, in fact. Transaction-based communication, accesses and changes to shared object occur very frequently in the distributed cooperative system. If the clients were to deliver this information synchronously, this would have very serious impact on the system performance. This has been validated in an early implementation version of the DyCE framework, where synchronous RMI calls were used to forward transaction objects from the client to the server for processing [2]. As the number of clients connected to the server grew, the system was perceived to slow down more and more and performance degraded to a point where the system was no longer usable interactively. More detailed performance measurements of RMI-based communication in distributed systems have been done elsewhere (e.g. in [NPH99], [KFCO99]). These evaluations indicate the performance overhead incurred by using RMI to pass messages (objects) from the client to the server, as opposed to more low-level communication mechanisms.

The alternative to this approach is to use asynchronous message passing, directly using TCP/IP sockets. Asynchronous communication is especially applicable in the given communication setting, since the client does not expect a direct return value from the server, in response to the transaction that was passed from client to server. For socket-based communication, the Java network library provides easy to use socket and stream implementations, allowing server socket creation, connecting to a server socket, setting up streams (communication structures ensuring sequential FIFO delivery of messages) and transmitting Java objects over these streams. A second generation of the network transport module in the architecture has been implemented using message streaming over TCP/IP sockets. This has been found to be a significant improvement over the RMI-based implementation, since the client was no longer suspended until the server had fully received the data packet containing the transaction.

Tests of this new implementation in have shown another performance bottleneck which starts to affect system performance as soon as more than five clients are connected to the server and are active simultaneously. This bottleneck is due to the connection-oriented nature of TCP/IP socket communication. For each client connecting to a server socket (via a "bind" call, see [Tan96] for details about network programming abstractions and operations), a separate socket is created to which the client is permanently connected - TCP (Transmission Control Protocol) is a connection-oriented flow control protocol built on top of the IP (Internet Protocol) low-level packet-based data transmission layer. The socket to which the client is connected is dedicated to this one client and is not available for input from other clients. Typically, a multi-threaded server implementation is used for servers (e.g. in most HTTP Web servers): After receiving the bind request from the client and setting up a socket connection, a separate process thread is forked for handling each connection, since data could

[2] No actual performance measurements have been performed at this stage of the implementation, therefore no numerical data is available. The value of comparing such data to current implementation benchmarks would be highly limited, though, since the implementation of DyCE has progressed so much in the meantime that no objective comparison would be possible anyway.

arrive over any of the connected sockets at any point in time. This reading thread then performs a "read" operation on the socket, suspending until a data packet has arrived. The reading thread reads the entire data packet from the network and forwards it to other modules for further processing, before it loops and waits for the next packet. From this, a number of performance bottlenecks arise when larger numbers of clients are connected to the server:

- Connection management: Since a dedicated connection is established from each client to the server, the number of simultaneous socket connections grows as each new client connects. With a high number of connected clients, this can create resource bottlenecks on some operating systems (while the limits on concurrently active sockets are more theoretical in nature, using multiple threads on multiple sockets does incur a high cost in thread context memory, allocated buffer spaces, etc.).

- Thread contention: As described above, a separate service thread is used for each incoming socket connection. As long as concurrent operation of the clients is low, this approach works very well: Suspended (non-active) threads require little performance overhead. As a data packet arrives on a socket, the corresponding suspended thread is resumed, the data can be read and processed. This can be done with little interference with the rest of the system. As the number of clients grows, though, the probability of simultaneous activity (and simultaneously arriving data packets on multiple sockets) rises. The probability of such simultaneous operation is extremely high in the application domain of the DyCE framework - interactive synchronous groupware - where many users are by definition cooperating and concurrently viewing and manipulating data. As more and more clients are active and sending data to the server, we are faced with the issue of thread contention: Multiple threads needing to become active at the same time in order to process the incoming data packets. This leads to increased demand for thread scheduling, thread context switching (suspending one thread, restoring the context of another thread and resuming that thread) and - ultimately - for synchronization: the incoming data needs to be put into data queues and structures ready for processing, these accesses to the shared data structures need to be synchronized across multiple threads in order to prevent chaos. Thread management and maintenance as well as synchronization use a large part of the system performance.

For these reasons, which directly influence performance scalability with respect to number of connected clients, it was decided to re-implement the network communication layer yet again, this time going from TCP to UDP (User Datagram Protocol). UDP communication is not connection oriented. Data items to be transmitted are wrapped in so-called datagrams, which make up the basic unit if transmission in UDP communication. The server merely needs one UDP socket for incoming packets, all UDP packets arrive at this socket and can be read by a single thread. Regardless of the number of connected clients, a single UDP input socket is sufficient, since all clients can send their data packets to the one UDP server socket.

Since, unlike TCP, UDP performs no transmission or error control, UDP provides less guarantees for message delivery:

- Guaranteed delivery: UDP packets are not guaranteed to arrive at the target socket. Packets can be lost somewhere along the way (especially, IP routers can discard UDP packets if buffer space is insufficient).
- Packet order: UDP packets are not guaranteed to arrive in the same order in which they were sent. Even if multiple packets are sent from the same sender to the same recipient, they can arrive in any order at the recipient (provided that they arrive at all, see previous item). This is due to the fact the UDP is a connectionless transmission mode, where subsequent packets sent from one sender to one recipient can take entirely different routes through the network.
- Packet duplication: UDP packets sent from the sender only once can nonetheless arrive at the recipient multiple times.
- Restricted size: UDP packets are constrained in terms of the size of the data they can carry. The high bound of this restriction is usually 64kB, but the size can be further restricted by intermediate routers, buffers in the machine's IP stack, etc.

Since UDP provide so few transmission guarantees, it became necessary to implement the required flow and transmission control in the DyCE network communication layer. A network layer has been developed, which provided the functionality of sending any serializable Java object from the sender to the recipient. Using a combination of sequence numbers and buffers, packet loss, packet sequence errors and packet duplication are compensated. The network transport layer performs the following operations:

- **Object serialization**: The Java objects to be transmitted from the sender to the recipient are serialized, i.e. they are transformed into a low-level byte array representation which can be converted back to the original Java object at the receiving site. In doing so, knowledge about DyCE `RObject` structure is used to provide optimized serialization: If a reference value in the object to be sent refers to a DyCE `RObject`, this value is replaced with a placeholder value wrapping the RObject's ID. The object actually referenced is not serialized along with the original object. In this way, the object serialization avoids sending DyCE data objects from client to server redundantly.
- **Data packeting**: The data block resulting from serializing the Java object to be sent can be much larger than can be transmitted in a single UDP datagram. Therefore, data blocks which are too large need to be split into multiple smaller blocks before they are transmitted.
- **Packet flow control**: In order to compensate packet loss, packet duplication and packet order errors, the sending and receiving side of the network communication layer employ caches and buffers for storing transmitted packets and for storing packets which cannot yet be processed, since previous packets have been lost. Using a simple communication protocol between sender and receiver, based on sequence numbers of packets, the receiver can request missing packets to be retransmitted. Upon receipt of a retransmission request, the sender fetches the reference packet from

its send cache and retransmits it. In order to compensate for delivery problems along the way (e.g. overflowing network buffers in intermediate routers), the algorithm for retransmission is self-pacing: Each packet contains a delay counter which is used to calculate a transmission delay between the sending of this packet and the subsequent packet. Multiple requests for retransmission of a packet increase the packet delay value, in effect reducing the potential for network congestions, since the network is given time to deliver the one packet before the next is transmitted. Successful reception of a packet is confirmed using a confirmation packet, the sender can then remove the packet from its packet cache.

- **Packet reassembly and deserialization**: After the receiver has received all packets belonging to a serialized object, it can reassemble the packets in the correct order (using the packet sequence numbers) and deserialize the Java object contained therein. This Java object is placed into an incoming queue ready for processing by the other system components.

The error control algorithm implemented for the UDP-based delivery uses an outgoing queue for unconfirmed packets in the sender, along with a sequence number counter. The receiving side uses two packet queues, one for caching packets received out-of-sequence (`sequenceCache`) and one for caching packets received in the correct order but which are parts of a multi-packet message of which the final packet has not yet been received (`packetQueue`). Additionally, the receiver maintains a counter for the last correct sequence number received from each potential sender (`last_correct_seq(sender)`). Each packet carries a unique sequence number, `p.seq`, for which there is an ordering relation "$<$", to discover whether the packet sequence number is before a given other sequence number (i.e. if a packet's sequence number is lower than another sequence number). The algorithm for packet processing at the receiving side is shown below (in a Java-like pseudo-language, for illustrative purposes).

```
/* Method called to process an incoming packet */
receiver.handle_packet (packet p)
{
  if (p.seq <= last_correct_seq(p.sender)
  {
    /* The packet is a duplicate of one that has
       already been processed successfully */
    discard_packet (p);
  }
  if (p.seq > last_correct_seq(p.sender) + 1)
  {
    /* At least one packet lost between the last one successfully
       processed and the one received. Store current packet for later use. */
    sequence_queue.add (p);
    sequence_queue.sort;
    // Request a resend of the missing packets
    for i=last_correct_seq(p.sender) to p.seq-1
      p.sender.request_resend(i);
  }
  if (p.seq = last_correct_seq(p.sender)+1)
```

```
  {
    /* The packet is correct in the sequence */
    last_correct_seq(p.sender) = p.seq;
    if (p.status = MORE_TO_COME)
    {
      /* The packet is part of a multi-packet message
        of which more packets will follow */
      packet_queue.add (p);
    }
    if (p.status = LAST_OF_SEQUENCE)
    {
      /* The packet is the last (or only) packet
         of the message */
      packet q;
      /* Concatenate all pending packets of this message */
      while (packet_queue.size > 0)
      {
        q = q + packet_queue.first_element;
        packet_queue.remove_first_element;
      }
      q = q+p;
      /* q now contains the entire message and can
         be deserialized and processed */
      object o = q.deserialize;
      o.process;
    }
  }
  if !(sequence_queue.is_empty))
  {
    /* Does the sequence queue contain elements which can
       now be processed? */
    while (sequence_queue.first_element.seq =
      last_correct_seq (sequence_queue.first_element.sender)+1)
    {
      packet p = sequence_queue.first_element;
      sequence_queue.remove_first_element;
      this.handle_packet (p);
    }
  }
}
```

Even though the network transport layer as described above seems to duplicate a large part of the transmission control properties already present in TCP, it is an important prerequisite for high scalability. With this network transport layer, larger numbers of clients can be handled without serious performance degradation on the server side.

Using the re-implemented UDP-based network communication layer, a test of connecting more than five clients to the DyCE server at the same and performing interactive operations on all five clients simultaneously was repeated. The resulting performance was perceived to be higher than in the RMI-only case

and also higher than in the TCP implementation. No systematic performance evaluations have been done, though. systematic tests of system performance under heavy load need to be done in the future. Also, additional steps need to be taken to improve system performance even more, to allow DyCE to scale to collaboration sessions including twenty or more active users.

Multicast delivery as a potential extension

In DyCE communication occurs between a single server and multiple clients. The previous section described how UDP packet delivery is used to implement asynchronous message-based communication. In the communication layer as currently implemented in DyCE, it is the server's responsibility to find the right clients to which to send an object update packet. In many collaborative situations, the set of connected clients does not change often. Also, in many cases all clients taking part in a collaboration session will receive much the same data packets, since the client object managers will to a large extent hold replicas of the same objects.

This fact can be taken advantage of to optimize the network interaction done by the server. Currently, if a transaction is to be sent to ten clients, the network transport layer is invoked ten times in order to deliver the transaction for each client. The network transport layer serializes the data packet ten times and sends ten (or n*10, in the case of the transaction needing to be split up into n packets) UDP packets over the network. Using UDP multicast, this network traffic can be reduced and the performance (and therefore the throughput) of the server can be improved.

Multicast packet delivery is a means for sending a single data packet to multiple recipients. IP multicast transfer is similar to UDP in that it is packet-based and not connection based. Instead of sending a packet to a single recipient, it is sent to a *multicast group*, which other systems need to join in order to receive the packets (see [HSH99] pp. 476ff for more details).

The network transport layer based on UDP provides much of the functionality which would also be needed when using IP multicast: packeting of data, sequence management, handling of packet loss, reassembly of packets at the receiving end. A logical extension to this would be to send the objects not to a single recipient but to a group of recipients which could be matched to a multicast group. Extensions to the network layer would need to be done to take into account that different clients could receive different packet sequences, with some packets reaching some clients, others reaching other clients (and in any order). This could mean more flexible decisions to resend a lost packet to an individual client (thus in the worst case sending a single packet more often than in the non-multicast case) or resending it to the entire group (e.g. if more than 50 percent of the connected clients requested a packet resend within a certain time frame. Additionally, the server would need to perform management of multicast groups and multicast addresses.

More complex connection management, using a combination of IP multicast and direct delivery, would need to be performed in situations where not all clients are on a single sub-net. Multicast packets cannot be expected to go across sub-net boundaries. In fact, in most cases they don't (the exception being the use of the MBone, the Multicast backbone, a globally available IP multicast communication structure based on explicit bridges between sub-nets

participating in the MBone). In configurations where clients are on different sub-nets, mechanisms would need to be employed in order to find an optimal grouping of the clients into a combination of multicast groups and direct UDP packet delivery.

All in all, using IP multicast packet delivery could prove to be a worthwhile extension to the network transport layer, proving especially beneficial in those cases where a large number of clients are active in the same collaboration session(s) and need to have replicated many of the same objects. A more detailed examination of the multicast communication issues and an implementation of the extended network transport layer is beyond the scope of this thesis.

5.2.3 Hypertext Transfer Protocol - HTTP

The Hypertext Transfer Protocol, HTTP, is the standardized protocol used - initially - for communication between Web browsers and Web servers. HTTP is a stateless request-response type protocol used for requesting items of information (the contents of a Web page, an image or other data forms) from a server. HTTP transport uses a single server socket (port 80 is the port usually used for HTTP communication) to which clients connect. A request packet, including among other items information about the requested resource and the expected return formats, is sent from the client to the server. This analyzes the request, locates the requested file (or, as is increasingly the case, generates result content on-the-fly) and returns the data to the client. The client/server connection is then closed (please note that this description is a simplification; all details of the HTTP protocol specification can be found in [FGM+99]).

The DyCE server includes a complete HTTP server, capable of both HTTP download as well as HTTP uploads. DyCE uses the HTTP protocol to download the component implementations from the Component Broker and also uses HTTP download images and other resources from the Web server integrated in DyCE. Additionally, components deployed via the server from a client are uploaded to the Component Broker using HTTP put.

A clear advantage of using HTTP transfer for these forms of transmission is the ability to use standard HTTP transfer support already available in the Java runtime library. HTTP download is one of the ways in which the Java class loader downloads class implementations, e.g. when running an Applet in a Web page. This mechanism needed to be extended to make it suitable for locating and downloading Groupware Components from the server (e.g. more sophisticated methods to locate the component implementations on the server than a single HTTP URL needed to be added), but the Java programming library provides a solid basis on which to base these extensions.

5.3 Integration with the Java standard API

Since DyCE is a development framework for the Java programming language, it is important to examine how the idioms and design patterns chosen for the framework integrate with the remainder of the rather large standard Java programming library, e.g. the parts used for user interface development. An important requirement (RD1) stresses the need for developers to be able to make use of as much of their previous programming experience as possible.

5.3.1 The Swing GUI library and reusable cooperative widgets

Groupware Components are visually interactive, i.e. they have strong ties to the user interface. The user interface development library introduced in recent versions of the Java SDK is the Swing GUI development framework. The Swing library, which was introduced as a successor and an extension of the Java Abstract Windowing Toolkit (AWT), provides a class hierarchy consisting of nestable Containers and components, which can be used to build complex graphical user interfaces.

The components of the Swing library follow a modified Model/View/Controller approach: A graphical component, e.g. a table (`JTable`), acts as both the view and the controller and has a data model (a subclass of `AbstractTableModel`). Modifications to the component's model are notified to the component through listener interfaces (e.g. the table class `JTable` implements the interface `TableModelListener` which provides a way in which the table's data model can signal changes to all components which are currently registered as its listeners.

The DyCE framework extends the Swing GUI library with the Groupware Components. The groupware Component base class, `MobileComponent`, is a subclass of the Swing library's class `javax.swing.JPanel`, i.e. it represents a rectangular area in the user interface and can contain any other Swing components. In this way, DyCE Groupware Components can be used in many places in user interfaces created using the Swing library, e.g. in dialog windows, in complex layouts, etc.

Maintaining view consistency

Additionally, the mechanism for notifying DyCE Groupware Components about changes to the shared data model mirrors related mechanisms in the Swing library: A group of Java interfaces, `ObjectChangeListener` and `SlotChangeListener` are used to bind Groupware Components to objects of the replicated object set and to be notified about changes occurring on these objects. The components can then update their local displays accordingly, reflecting the new data state at the user interface.

As has been discussed, DyCE separates the data (or domain) model objects from implementation of the components working on these objects. When a component is invoked on a shared object, by performing a task, the shared object is bound to the component as its *model*. According to the Model/View/Controller pattern of application development, the DyCE component acts as both the view *and* the controller.

Similar to the approach used in Java's GUI framework, Swing, the component is bound to the shared object as an observer (in the DyCE framework, the component becomes an `ObjectChangeListener` on the shared object). Changes to shared objects are always done within transactions (see section 4.3.5). In the validation phase of a transaction (either when a transaction received from the server is being replayed on the client or when a transaction originating on the client is confirmed by the server), the framework gathers all `ObjectChangeListener`s registered on the affected objects and notifies these (similar to the listeners in the Swing framework, a listener interface method, `objectChanged(ObjectChangeEvent o)`, is called, passing additional informa-

tion about the changes that occurred in the event object. It is the components' responsibility to react to these notifications and update their display accordingly.

This form of notification by model changes through the observer or listener mechanism is close to the way in which GUI programming is done in the Swing framework. The chosen paradigm should therefore be familiar to Java programmers.

DyCE extensions of Java widgets

In order to relieve developers of the burden of implementing similar GUI items over and over again, DyCE provides a number of "DyCE-enabled" Swing components. These are specialized extensions of Swing widgets (such as a specialized text input field or a specialized checkbox) which can be bound to specific Slots of the shared object model (for instance, the DyCE-enabled text field can be bound to any Slot containing String data). The Swing widgets have been extended to automatically reflect user input (e.g. editing of the text) in the bound Slot contents and to react to changes of the shared data model by automatically updating their display. By using such DyCE-enabled Swing widgets and binding them to elements of the DyCE shared object model, developers can more quickly assemble collaborative user interfaces. The extended widgets conform to the same programming interface as the ones with which developers are already familiar, with the added requirement to bind a DyCE-enabled widget to a certain Slot. Thus, such extended widgets can be used almost interchangeably with regular Swing widgets.

It is important to note, though, that such DyCE-enabled Swing widgets are used *as part of* Groupware Components and are not themselves Groupware Components. Extended Swing widgets do not apply to a certain model object implementation (they are merely bound to Slots which are contained *in* the shared object implementation), nor do they publish tasks which can be invoked.

5.3.2 Implementing Transactions in Java

As has been discussed in section 4.3.5, accesses to the shared data objects need to be encapsulated in transactions. These transactions need to be written by the developer implementing the shared domain data model or the components. For performance reasons, the DyCE framework does not use locking mechanisms to synchronize access to shared resources. Rather, transactions are allowed to proceed (optimistically) on the client originating the transaction. When a conflict is detected post-hoc, the effects of those transactions which were in conflict with other transactions need to be selectively undone (see [Wei93]).

A transaction performed on one of the nodes of the distributed system can be understood to consist of three phases: the preparation phase, the transaction operation phase and the transaction commit (validation) phase. In the transaction preparation phase, all necessary initializations are performed which are required to allow the transaction to be performed. A transaction object needs to be initialized, ready to add the operations performed in the transaction. In the transaction operation phase, read and write accesses to the shared data objects are performed and the appropriate information is added to the transaction object. In the transaction validation phase, the transaction is wrapped up, sent to

the server for validation and added to a local "pending" queue. After the server has reached the decision whether to allow the transaction to commit, it sends a confirmation or cancel message to the originating machine and distributes the accepted transactions to all other connected systems. If the transaction is not allowed to commit, the system on which the transaction has originated needs to perform an undo of the transaction and any transactions performed in the meantime.

When providing framework support for programming transactions, the characteristics of the components need to be taken into account, especially the multi-threaded nature of the Java programming language. Transactions need to be performed in a thread-safe manner; this means that even in a multi-threaded application, the ACID properties of transactions need to be maintained: a transaction being performed on one execution thread must not be able to affect the operation of transactions on another thread.

An initial approach to programming support for transactions in the DyCE system consisted of a `TransactionManager` class with the appropriate support methods for the transaction processing, implementing the following interface:

```
public class TransactionManagerInterface
{
    // Get the TransactionManager for the local system
    public static TransactionManagerInterface getTM();

    // Start a new transaction
    public boolean beginTransaction();

    // Retrieve the current transaction
    public Transaction getCurrentTransaction();

    // Commit the current transaction
    public boolean commit();

    // Abort (undo) the current transaction
    public boolean abort();
}
```

Using this interface, transactions could be programmed in user code, which accessed values store in the instances of the shared data model. A transaction could be begun using the `beginTransaction` method. This would do the necessary initialization operations for the transaction preparation: Create an instance of the `Transaction` class, initializing the transaction's write set and read set. Operations performed on the Slots of the shared data objects result in the appropriate `txOperation` instances being added to the running transaction. When committing a transaction with the `commit` method, the transaction, including its read set and write set was serialized and sent to the server for validation. The transaction was added to a pending transaction queue, waiting for confirmation or cancellation from the server. A typical piece of user code, in a shared data model of a geometrical shape, would look like this:

```
// Move the shape by an x and y offset
```

```
public void moveShapeBy (int dx, int dy)
{
    TransactionManager.getTM().beginTransaction();
    int x = get("XPos");
    int y = get("YPos");
    x = x+dx;
    y = y+dy;
    set ("XPos",x);
    set ("YPos",y);
    TransactionManager.getTM().commit();
}
```

The problem with these segments of user code, which caused unpredictable effects in the course of highly active collaboration sessions with components built on the DyCE framework was its lack of thread-safety. It is easy to construct cases of transactions constructed as above running in different threads which influence each other. Using this form of in-line transaction programming, such effects are hard to avoid. Java offers the `synchronized` keyword to synchronize multiple threads on a certain object (i.e. only one thread may be active in a synchronized method at a time). This keyword is offered to allow Java programmers to prevent multi-threading problems. Unfortunately, simply making the transaction method `moveShapeBy` in the above example synchronized would not have solved the concurrency problems, since other methods in other objects (which would not be affected by the `synchronized` keyword) could be performing operations which affect the operation of the transaction. Taking into account the fact that also creating new transaction objects and committing transactions can affect each other, the problem lies not only in the accesses to the Slot values. Merely making the beginTransaction and commit methods synchronized would also not have been sufficient, since conflicts can also occur at any point in the user program between these two instructions. It would be impossible (and error-prone) to explicitly synchronize all segments of user code where transactions are being created, Slots are being accessed, etc.

The solution to this problem is to use the Java programming language feature referred to as anonymous inner classes (unnamed classes encapsulated within the implementation of another class, visible only within the context of the surrounding class) and providing a base framework class `Transaction` from which user transactions can be derived as anonymous inner classes, can be instantiated as transaction objects and can be performed. In this way, the transactions become indivisible and synchronizable transaction blocks, which can be evaluated under control of the DyCE framework. The `Transaction` base class provides the single entry point for performing the transaction - the `doIt()` method, which cannot be overridden in subclasses. User code is implemented in a transaction body method, `txBody()`:

```
public class Transaction
{
    protected Object txResult;

    // . . .
```

```
    // Abstract method for implementing user code
    public abstract void txBody();

    public synchronized final Transaction doIt()
    {
        beginTransaction(); // Make a transaction
        txBody();           // Perform the user code for the transaction
        doCommit();
        return(this);
    }

    public Object result()
    {
        return txResult;
    }

}
```

By making the doIt method synchronized and final, it is ensured that only one thread may be active in the doIt method at a time. The synchronized construct is re-entrant, though, meaning that the same thread may run through the doIt method several times. This new and thread-safe `Transaction` class can now be used in user code as follows:

```
// Move the shape by an x and y offset
public void moveShapeBy (int dx, int dy)
{
    new Transaction()
    {
      public void txBody()
      {
        int x = get("XPos");
        int y = get("YPos");
        x = x+dx;
        y = y+dy;
        set ("XPos",x);
        set ("YPos",y);
      }
    }.doIt();
}
```

This implementation of transaction blocks exhibits several features which are desirable for the transaction model in the DyCE framework. Transactions can be *nested*: A single thread can perform as many nested calls to the doIt method as it likes. The implementation of beginTransaction only needs to verify whether there is already a running transaction before initializing the read set and write set (so that no pervious operations are accidentally removed). In everyday development of DyCE components, the code written in the txBody method very often calls other methods of other objects which also contain transactions. These transactions are now nested into the surrounding transaction. When the outermost transaction exits its user code routine, the `doCommit` method commits

it along with all nested transactions. The solution is *thread-safe*: Due to the assertions of the `synchronized` construct, no two different threads can be active in the doIt method at the same time. After the first transaction has entered the doIt method, subsequent calls to doIt on other threads are suspended until the initial transaction commits. Also, there can be *no dangling locks*: When the transaction code leaves the doIt method - whether by an error condition or after successful completion - no locks or semaphores remain to be removed in order to permit other transactions to proceed.

5.3.3 Type wrappers for Java types

As has been discussed in section 4.2.6, knowledge about the functionality of complex data structures can be used to increase possible concurrency of operations. As specified in the previous chapter, Slots provide two basic methods for accessing them: getValue() and setValue(). These accesses are monitored, wrapped in transactions and transmitted via the server to other clients. Using such elementary Slot operations, a Slot can contain any data type and Slot accesses always conform to the following simple pattern:

1. Read the Slot's contents into a temporary variable
2. Modify the object or data value
3. Write the changed value back to the Slot.

In the case of complex or large data structures (e.g. a Slot containing a list of elements in a Vector or other list structure), writing the changed Slot contents in step 3 results in a transmission of the entire data structure contents to all connected clients (since the framework can have no knowledge of what exactly changed in the data object and has to assume that an entirely new object has been written into the Slot). This approach, while homogeneous and functional, can lead to a large volume of unnecessary data being transmitted, if many small changes are made to large data structures.

In order to improve on this situation, the DyCE framework provides DyCE wrappers for common complex data types. So far, wrappers have been developed for the linked list class `Vector` and the `Hashtable` data structure. Additional data type wrappers will be developed as when required. The DyCE type wrappers inherit from the Java base classes and are therefore polymorph with these classes. Along with these new subclasses, extensions to the Transaction class hierarchy presented in section 4.3.5 have been implemented. All modification methods in these classes (methods modifying the data object's contents) have been overridden according to the following pattern:

1. Check for running transaction. Throw exception if no transaction active;
2. Create instance of appropriate `changeOperation` subclass, storing information about current state for later undo operations;
3. Perform requested change on local data structure;
4. Store information about the change to be done in the `changeOperation` subclass instance;

5. Add `changeOperation` subclass instance to running transaction.

For each overridden modification method, an appropriate subclass of `changeOperation` needs to be implemented, which is able to perform and undo the changes requested by the method invocation.

When a transaction received over the network is being replayed locally by the Transaction Manager, the `changeOperation` subclass instance method `perform()` will be called, to replay the change locally by applying the change on the local replica of the data structure.

As an example, consider the following code snippets from the DyCE-enhanced `Vector` (linked list) data structure, specifically the method and change operation class for adding a new element to the list:

```
/** Helper class for Vectors in the data model */

public class Vector
    extends java.util.Vector
{

    // .....

    /** Wrap addElement for ModelObjects, thereby hiding
    the Vector's access to RObjects from the client.
    DyCE-Aware! Adds the appropriate changeOperation */
    public synchronized void addElement (Object o)
    {
        if (o instanceof GroupComponents.ObjectModel.ModelObject)
            super.addElement (((GroupComponents.ObjectModel.ModelObject)o).
                getDataObject());
        else
            super.addElement (o);
        try
        {
            if (oid == null)
            {
                return;
            }
            // Create the changeOperation instance and add to
            // running Transaction
            Transaction.getCurrent().addOperation (
                new VectorAddElement (
                    slotname,
                    oid,
                    ObjectManager.getObjectManager().getObject (
                        oid, ClientObject.getMyClientID()).getLClock(),
                        o)
                );
            // Here we need to notify the Slot of changes
            GroupComponents.ObjectModel.Slot.changed (oid, slotname);
        }
```

```
            catch (Exception ex)
            {
                ex.printStackTrace();
            }
        }
    }

    /**
    * changeOperation for adding an Element to an
    * instance of GroupComponents.Utilities.Vector.
    */
    public class VectorAddElement
        extends VectorChangeOperation
        implements Serializable
    {
        Object addedObject;

        public VectorAddElement(String sn, ObjectID o, int lc, Object added)
        {
            super (sn, o, lc);
            addedObject = added;
        }

        /** Perform this operation simply by actually adding the object */
        public void perform()
        {
            Vector v = (GroupComponents.Utilities.Vector)
                (getAffectedObject());
            v.addElement (addedObject);
        }

        /** Undo the effects of this operation */
        public void undo()
        {
            Vector v = (GroupComponents.Utilities.Vector)
                (getAffectedObject());
            v.removeElement (addedObject);
        }
    }
```

Since the DyCE transaction framework ensures that operations are performed and undone in order, the changeOperation implementation can perform the requested functionality by adding the given object when performing the operation and removing it when undoing the operation.

For Slots containing such DyCE-enabled data types, the component developer is no longer restricted to using only the read/modify/write interaction paradigm. Rather, he can use the more natural way of interacting directly with the complex data structure implementation and rely on the fact that the framework will handle these changes correctly and maintain all object replicas' consistent state.

5.4 The Groupware Desktop

The Groupware Desktop is the standard client application available as part of the DyCE framework. The desktop (see figure 5.3) provides end-users with support for accessing the Groupware Components and shared objects. Additionally, it provides some awareness information. It has been developed to fulfill the requirement RU1, providing access to shared artifacts and collaborative tools.

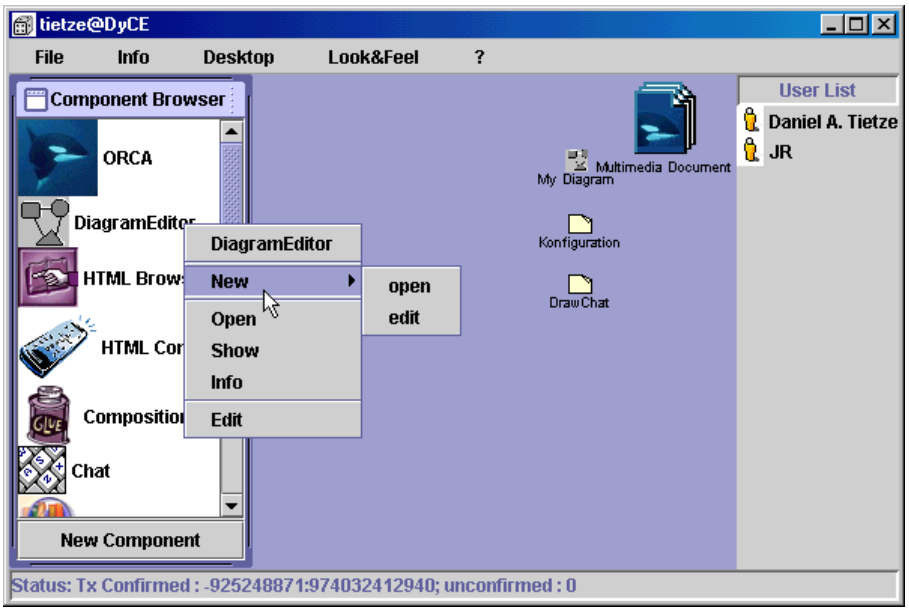

Figure 5.3: Groupware Desktop client application

The Groupware Desktop shown in figure 5.3 provides the following features:

- **Component Browser**: The Component Browser (in the window on the left part of the screenshot) provides direct access to all Groupware Component currently registered in the system. By interacting with the Component Browser, users can create new shared objects and invoke tasks upon these. Invoking the tasks leads to creating a new session, and loading and opening the necessary Groupware Component in this new session, as has been previously described. The screenshot shows the popup menu used by the user to create a new model object instance for a shared drawing and invoke one of the tasks "open" and "edit" on the newly created instance. This is the way in which end-users create new sessions and set up new documents which they wish to use and edit collaboratively.

- **Registering a component**: Transferring components uses the JAR (Java Archive) mechanism supported by the Java Development Kit. A JAR file is a compressed archive of all class implementations and resources belonging to a Groupware Component. This includes any icons or other external support files required by the component implementation. Using the "New Component" button on the user interface, end-users can register

new Groupware Components to the Component Broker. After selecting the component implementation JAR archive, setting a name and an icon to use for that component, the component implementation is uploaded to the Component Broker. The newly registered component shows up in all connected users' Component Browser and can be used right away, without the need to shut down and restart the clients. This is the means by which users can dynamically extend the working environment (see req. RU7).

- **Desktop**: The desktop part of the Groupware Desktop provides functionality similar to the desktop environments from familiar operating systems. Users can place shortcuts to shared objects on their desktop for easy access to these objects (see icons e.g. "My Diagram" in screenshot). Again, a popup menu provides direct access to the task information stored for each of the model object classes and allows invoking a task on a shared object, creating a new session and opening the necessary Groupware Component on the shared object.

- **User list**: The user list, on the right-hand side of the window, provides a list of all users currently logged in to the system. This list can be used to contact other users and to invite them into running collaboration sessions.

When the user invokes a task on a shared object, either from one of the icons on the desktop or from the Component Browser, a new session is created, containing the component discovered for that task. For each session, an independent window is created which contains all Components used in that session (see figure 5.4). In each session window, there is information about which users are currently in the session (see top bar in right-hand window). Users can be invited into running sessions e.g. by dragging the representative of the user from the Groupware Desktop user list to the user list in the session window.

The menu (see indicator in figure 5.4) in the component-related frame in the session window allows invoking additional tasks on that component's data model. These tasks can be invoked within the same session (thereby adding a component to the running session, performing the task for all users currently in the session) or in a new session, thereby spawning an independent session on the same data model. By using multiple windows to each represent a running session and providing access to elements of the shared object space and components, along with the possibility for opening new sessions and adding components to running sessions, users are provided with a large degree of flexibility in creating their collaborative settings. Individual sessions can be spawned e.g. to work independently on other sections of a shared document, other users can be invited into such spawned sessions. Also, totally independent shared objects can be created and component invoked on them, in order to perform individual tasks or private, non-coupled, work in parallel. Thus, multiple simultaneous collaboration modes as well as transitions between them (RU6) are supported in the Groupware Desktop.

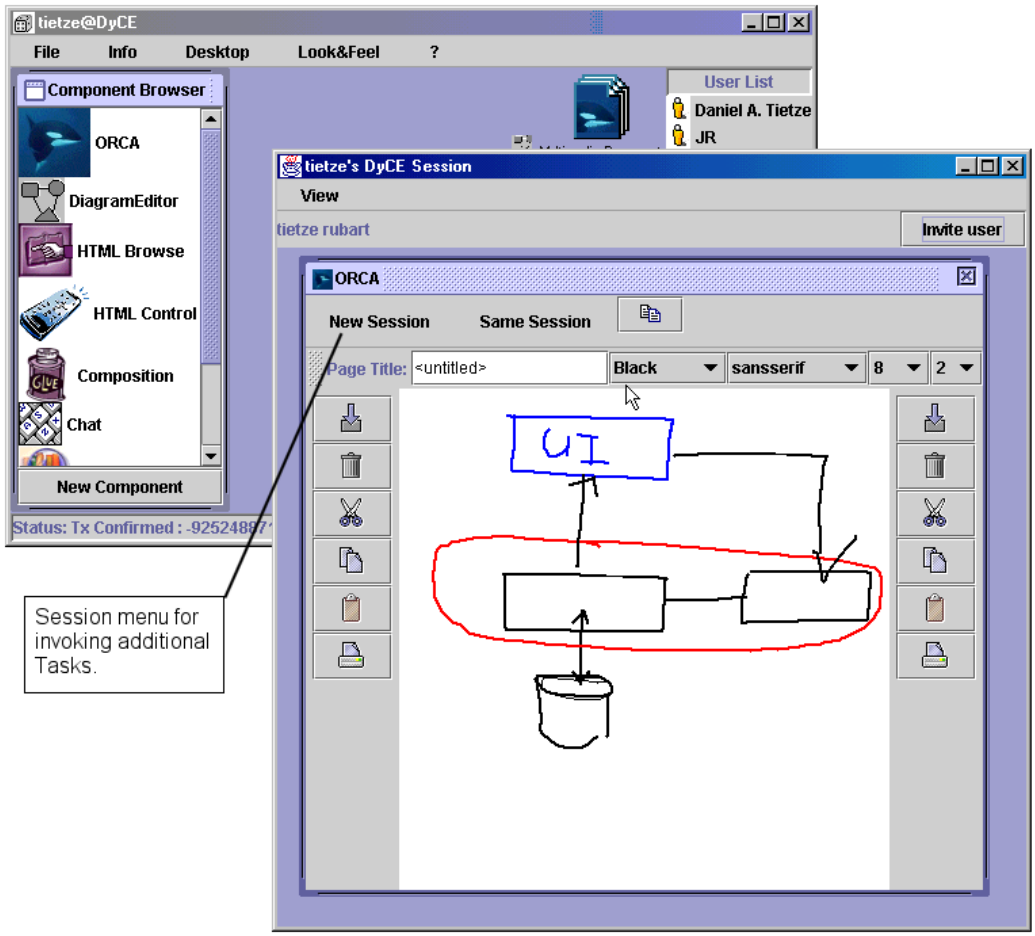

Figure 5.4: Groupware Desktop and a running session

5.5 End-User tailoring

An important requirement for the Groupware Components was to provide end-users with support for combining existing components as the need to do so arises in the course of the collaborative process (req. RU10). The DyCE framework includes as a Groupware Component the tool used for coupling Groupware Components and deploying the newly combined components to other users. Using this tool, Groupware Components can be coupled into so-called Configurations. A Configuration is defined as follows: A Configuration consists of a human-readable name and a set of Configuration Items. A Configuration Item is a tuple $CI = \{name, class, task, rect\}$ where

- $CI.name$ is a human-readable name for the Configuration Item,
- $CI.class$ is the name of a shared object class,
- $CI.task$ is the identifier of a task published on $CI.class$,

- *CI.rect* is a rectangle denoting relative layout information

5.5.1 Use of the end-user tailoring tool

The configurations can be collaboratively created in a visual Configuration Editor (see figure 5.5), which allows naming the configuration and adding new Configuration Items to the configuration. The layout editor (see lower part of Configuration Editor in screenshot) allows relative positioning of the *rect* information for each Configuration Item, using a rubber-banding feature.

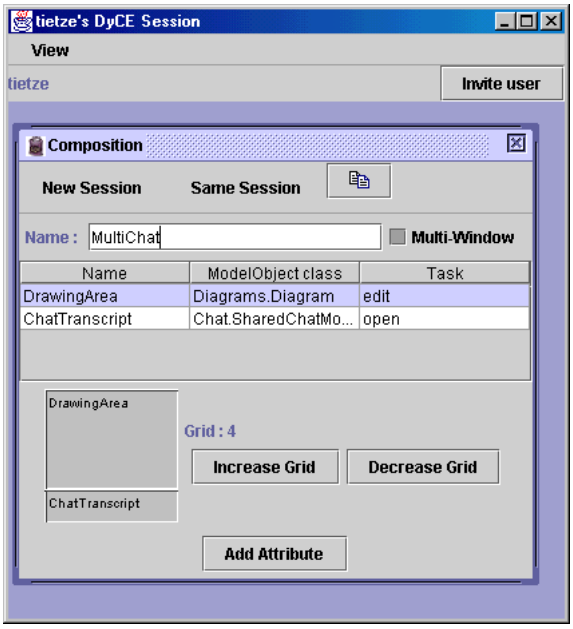

Figure 5.5: Using the Configuration Editor to compose Groupware Components

The shared model of the Configuration Editor is a Configuration object. This Configuration object can be treated like all other DyCE shared objects: A shortcut can be placed on the desktop, its ID can be used to refer to it from other components, etc. Additionally, a configuration can be "started". Starting a configuration results in creating instances of all shared object classes included in the Configuration Items, performing the specified tasks on these and laying out the resulting components according to the layout information contained in the Configuration Items. The resulting combination of components is displayed as a single tool in a new session window, other users can be invited into the session and use the combined components collaboratively. The Configuration Editor shown in figure 5.5 is being used to combine a shared chat component, accessed by introducing a shared chat model and the appropriate task, with a group drawing tool, accessed by introducing a shared drawing model and the appropriate task. The collaborative use of this configuration is shown in figure 5.6. Here, two users are collaborating using the resulting combined component, which provides a combination of chatting and drawing, to support a graphically

enhanced chat session (or to use the chat tool to discuss the contents of a shared drawing).

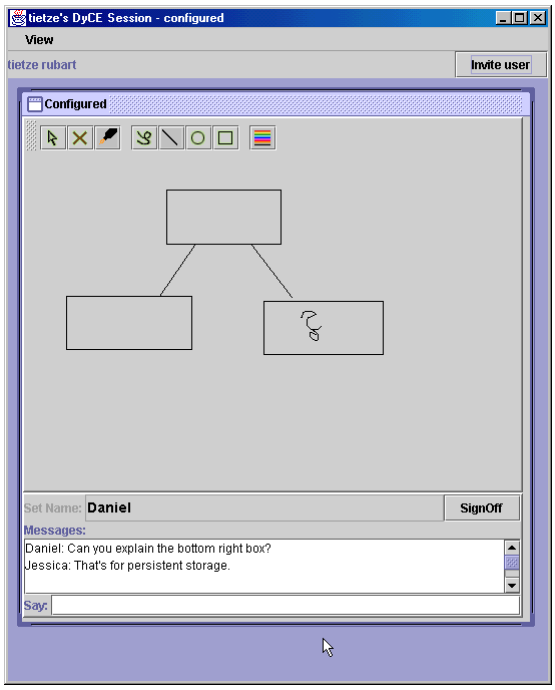

Figure 5.6: Collaborative use of combined Groupware Components

5.5.2 Configuration based on tasks

As can be seen, the combination of Groupware Components does now directly access the components to be combined. Rather, the end-user specifies shared objects and tasks to invoke on them and performs a rough relative layout. This approach carries the flexibility gained from the task model through into combining the components. Instead of specifying fixed configurations of specific tools, the decision of which tools to use is left to the system, based on the information in the task model. Whenever a configuration is started, the tasks contained therein are resolved using the Component Broker and the correct components are identified. Should the set of available components change, or should different components be used for different users, the configuration still remains valid. Even if components are replaced with ones with enhanced functionality, the system ensures that these components work in all configurations based on the shared object space.

5.6 Shared Workspaces on the Web

One requirement stated in chapter 2 is that of ubiquitous access to the collaboration support system (RU5). One computing infrastructure which is be-

coming increasingly available to a wide variety of systems is the World-Wide Web (or Web for short). By making DyCE components accessible over the Web, DyCE-based shared workspaces can be accessed from virtually any Web-enabled system.

5.6.1 Downloading DyCE itself

As has been shown in section 5.1, the DyCE server (the server's user interface is shown in figure 5.7) includes an HTTP server, which allows DyCE to serve itself to Web-based clients. DyCE components can be embedded in Web pages and can be used for collaboration over the Web and for ubiquitous access. Downloading DyCE from its own HTTP server also avoids problems with the "sandbox model", in which Applets may only connect back to the server from which they were loaded. After downloading DyCE from its own server, the DyCE Applet is allowed to connect back to the DyCE server using UDP and RMI, which allows the distributed system to be established.

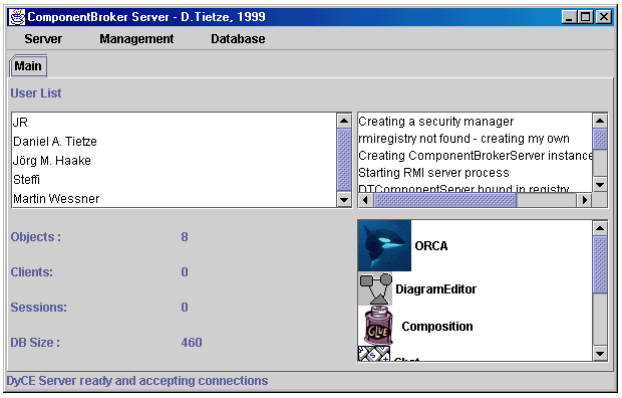

Figure 5.7: The DyCE server ready for use

5.6.2 Transferring Components

Components registered in the Component Broker are stored in JAR (Java ARchive) files, which include the implementation files of the Groupware Component, as well as the implementation files which make up the shared data files. Along with these Java class files, the JAR archive contains any resources, such as icon images, help texts, etc. which are required by the component. In order to allow retrieval of the component and its supporting files at runtime, the Java class loader has been extended to make use of the Component Broker. Any class file or resource file requests which cannot be fulfilled by local access (i.e. by accessing a locally stored file) is delegated to the server's Component Broker, which has access to the previously registered JAR archive and can serve the files from there. Attempting to access local versions of the requested files is a means for providing developers with the ability to to test out their component implementation together with the server, without having to re-register the component after every change. Obviously, for deployed DyCE installations no local

component implementations are used, in order to prevent version mismatches between the collaborators' installations.

Deployment of new components must also be supported. For this, the HTTP server in DyCE also accepts HTTP PUT requests for uploading JAR files containing components. This allows uploading components to the server using an HTML form (see figure 5.8). This HTML form can again be served from DyCE's HTTP server, providing an easy Web interface to extend the system. Similar file uploads to a shared workspace have also been realized in BSCW [BAB+97] but in BSCW this mechanism is used to maintain repository contents, not extend the CSCW system itself.

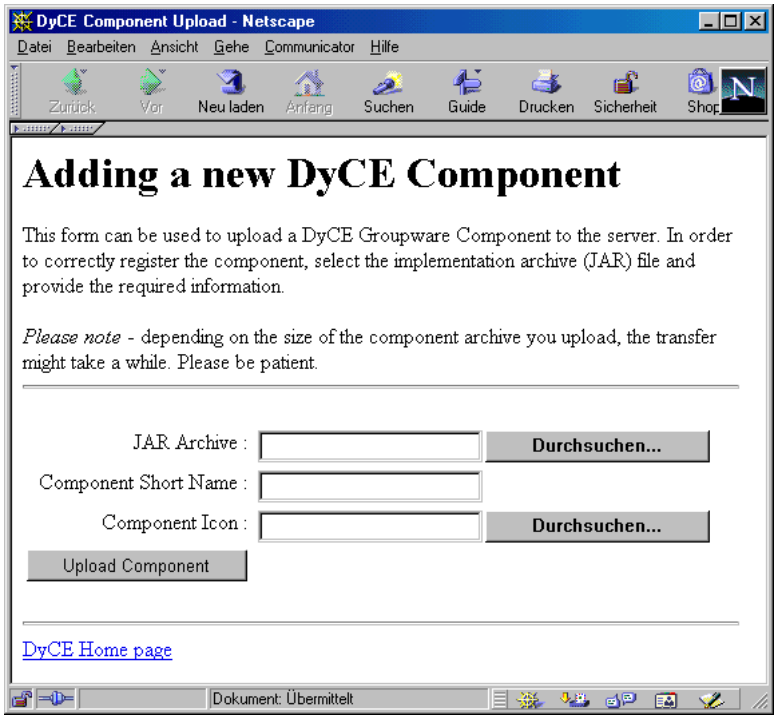

Figure 5.8: HTML form for Component upload

5.7 Experiences from DyCE development

The DyCE framework was developed following an iterative development model (sometimes also referred to as rapid prototyping - a term which seems to sound a bit negative). An initial prototype of the framework and some components using it was initially developed in a rather short time. Subsequently, there were two forms of tests - the framework suitability for developing groupware components was tested by introducing several developers (some very experienced in Java development, others less so) to the framework and having them build additional components. In parallel, the components which were already present were tried in various collaborative as well as single-user settings. The aim

of these tests was to quickly gather requirements and priorities for the next development iterations. Since this was a rather informal process, no formal evaluation has been done and no formal evaluation results can be presented.

In the following iterations of extensions of the DyCE framework, new functionality has been added and the framework itself has been refined in a process that can best be termed stepwise refinement. One of the steps in object-oriented development which is gaining more wide-spread acceptance is that of refactoring - improving the design of existing code to eliminate areas of the design or the implementation which, while they actually work OK, nonetheless *smell bad* (see [Fow99] for more details about this).

In the course of refactoring the framework, more and more parts of the framework were "bootstrapped", i.e. the framework was redesigned and extended to, in effect, use itself. For example, initially there were specific classes (regular Java classes) representing users, sessions, components and other DyCE-internal data structures. This was necessary in order to get DyCE working at all. Once the general functioning of the framework had been established, these data structures were gradually replaced with "DyCE-ified" versions of themselves, so that now most of the DyCE internal data structures are in fact `ModelObject` subclasses and fully benefit from the replication support, concurrency control, etc. Finally, the DyCE framework is a very homogeneous framework, with very little control structures outside of the framework. This bootstrapping approach has proven to be very beneficial in that it aids the framework's comprehensibility (less underlying concepts need to be understood and issues learnt once can be re-applied in various parts of the framework) and its maintainability (changes or improvements to a central mechanism such as the object model or the replication management immediately benefit large parts of the framework).

These benefits are, in our opinion, only achievable in an iterative development approach. The final homogeneous, self-referential design of the framework using itself would have been next to impossible to come up with beforehand (without the previous implementation experience) and would have been very hard to implement and test from the beginning.

Chapter 6

Usage Experiences

The system described in this thesis has been used to develop several collaborative components for a variety of usage situations. This chapter will present a number of systems developed on the basis of DyCE. Each system's design will be presented, followed by a description of how the system exploits the features of the DyCE system.

6.1 Shared HTML Presentations

Using DyCE as a basis, an HTML-based collaborative presentation system has been developed, which can be used for slide-show presentations where one presenter shows a presentation, consisting of a set of slides in HTML format, to a - potentially distributed - group of viewers (e.g. students).

The HTML presentation system consists of four parts (see figure 6.2):

- The *HTMLPresentation model*. This is modeled as a list of URLs referencing the slides making up the presentation and a "current" slide indicator. The slides themselves are stored in the DyCE HTTP server (but could also be fetched from any Web server).

- The *HTMLPresentationController component*. This component publishes the task "control" on the data model class HTMLPresentation. It is used to control the presentation. For this, it allows paging forwards and backwards through the presentation slides, updating the "current" slide in the shared HTMLPresentation model.

- The *HTMLPresenter component*. The HTMLPresenter component is a MobileComponent subclass. It publishes the Task "control" on an HTMLPresentation. When invoked on an HTMLPresentation, the HTMLPresenter fetches and displays the slide indicated in the HTMLPresentation model as the current presentation location. In order to do so, it incorporates an HTML viewing panel (from the standard Java libarary). When the "current" value of the presentation is changed (by the HTMLPresentationController), the new slide is fetched and the display is updated accordingly.

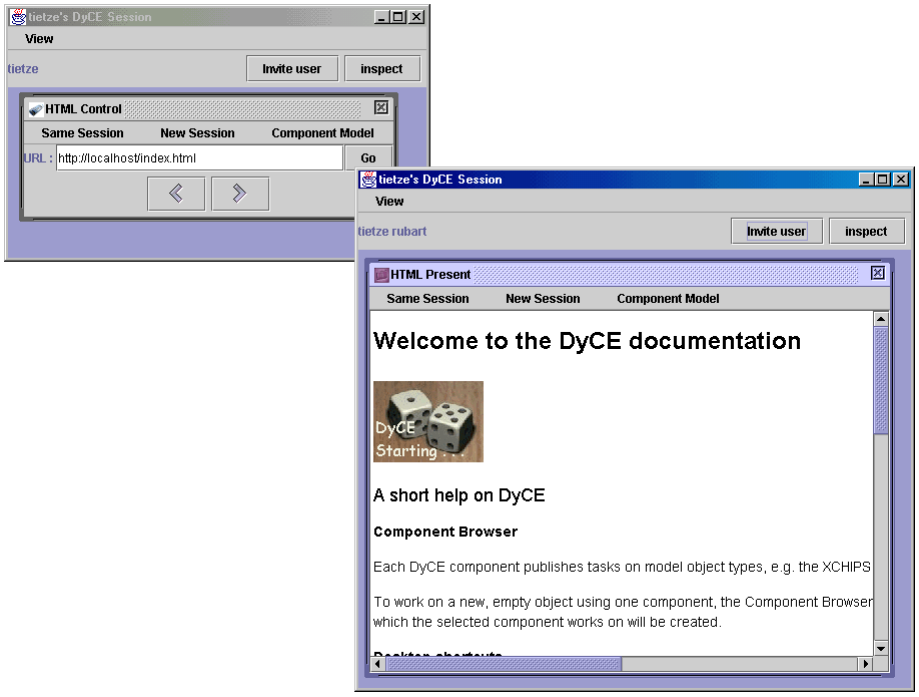

Figure 6.1: Shared browsing in an HTML-based presentation

- The *HTMLBrowser component*. The HTML Browser component extends the HTMLPresenter component. It publishes the Task "browse" on an HTML Presentation. In addition to the shared presentation functionality inherited from HTMLPresenter, the HTMLBrowser component allows true cooperative navigation. If a user follows a link in the HTML page, the shared model is updated and all other users' browsers (or presenters) follow. This allows collaborative exploration of HTML content.

The screenshot in figure 6.1 shows two shared presentation sessions (from the point-of-view of the user controlling the presentation), with the HTMLPresentationController being used in an individual session by one user (the tutor, who is controlling the presentation) and the HTMLPresenter component used collaboratively by two users in a shared session. In the shared presentation component, the users are viewing the presentation page to which the tutor navigates using the control component. As has been explained, when navigation (changing pages) occurs in the presentation controller, the coupled presentation components automatically follow, resulting in a shared presentation.

Using these components on a shared HTMLPresentation model, any number of viewers can simultaneously use the HTMLPresenter to view the presentation which is controlled by the tutor, using the HTMLPresentationController component.

This HTML Presentation example is also a good example of how shared DyCE models can be used to synchronize components accessing an external resource (in this case the presentation of HTML slides stored on a Web server).

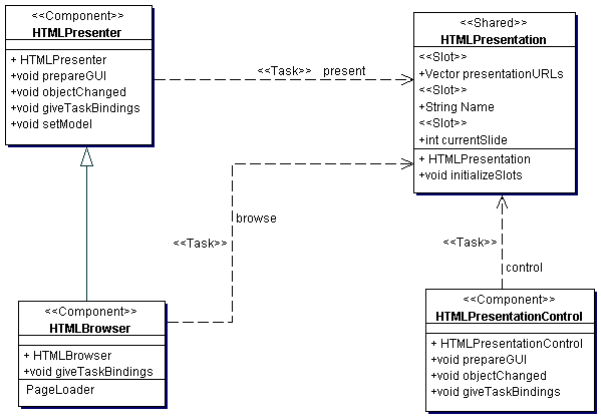

Figure 6.2: Design of HTML Presentation Environment

6.2 Collaborative Hypermedia with Awareness Support

The DyCE system has been used to implement the collaborative hypermedia workspace ideas described in [Haa99]. The goal was to provide a shared Hypermedia Workspace which provides a number of tools with group awareness mechanisms. The awareness-enhanced tools include:

- a group aware search tool for performing complex searches on the shared hypermedia workspace (labeled (1) in figure 6.3),
- a structure navigator providing a hierarchically structured overview over the workspace structure (labeled (2) in figure 6.3) together with awareness information about sessions running on the workspace contents,
- a content editor for shared hypermedia content (not shown).

6.2.1 Design of the Shared Hypermedia Workspace

In order to realize the system, the first step was to develop a shared hypermedia model which was to form the basis of the shared workspace. This hypermedia model was developed as an extension of the DyCE shared object model. A general hypermedia layer was implemented (see figure 6.4), which includes:

- a general link model for typed hypermedia structure which provides support for multiple link schemas (consisting of a set of permitted link types, defined as a triple $\{linktype, sourceclass, destinationclass\}$.
- a hypermedia framework, consisting of a number of interface definitions (Java interfaces); by implementing these interfaces, any object model class can denote itself to be linkable and be included in the hypermedia structures.

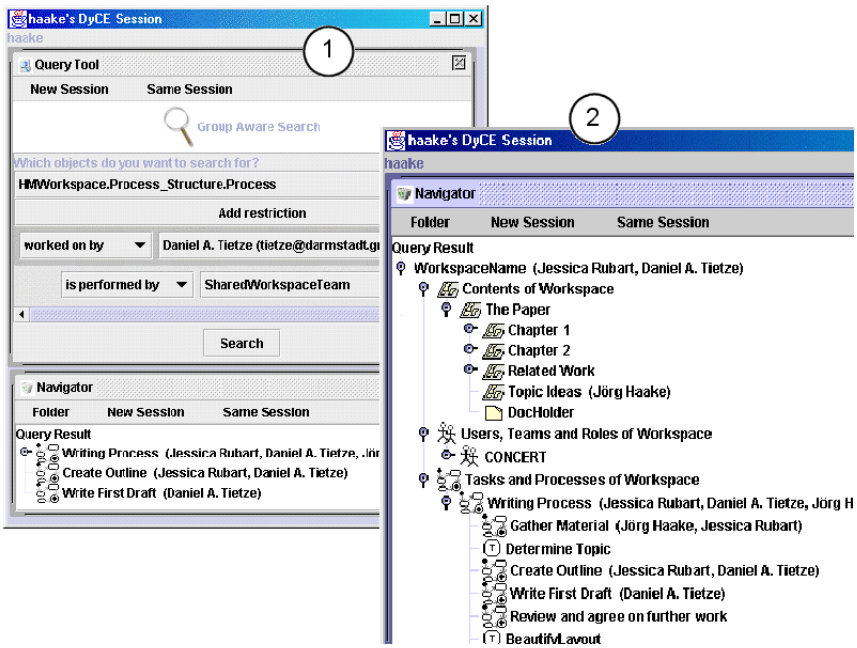

Figure 6.3: Tools of the Shared Hypermedia Workspace

- support classes providing services such as link management, schema management.

The shared hypermedia model consists of three distinct areas, each providing classes with an appropriate nesting structure: Process Structure, Team Structure and Content Structure.

On the basis of this hypermedia layer, a number of components were implemented, which provide the functionality described in [Haa99].

A *shared navigator* visualizes hierarchical structures (such as shared workspaces, query results, etc.) along with awareness information. Using an Explorer-like presentation consisting of nested folders representing composite elements of the structure to be displayed, the users can collaboratively navigate such nested structures, showing or hiding sub-structures as they please.

A *group aware search tool* allows a group of users to co-operatively search the shared hypermedia workspace for elements matching certain query conditions. The query can be formulated interactively, using interaction elements provided by object model implementations implementing the interfaces from a query framework. As the shared navigator is used to visualize the query results from the search tool, the query results are automatically group-aware (i.e. contain information about which of the elements found is currently being used by which users).

The *Workspace Editor* is used to view and manipulate the structured hypermedia workspaces. Whereas the navigator provides a tree-like view of hypermedia structures, the workspace editor provides spatial layout capabilities on two-dimensional workspaces. The workspace editor provides access to all elements modeled in the shared hypermedia model. Root elements of the Con-

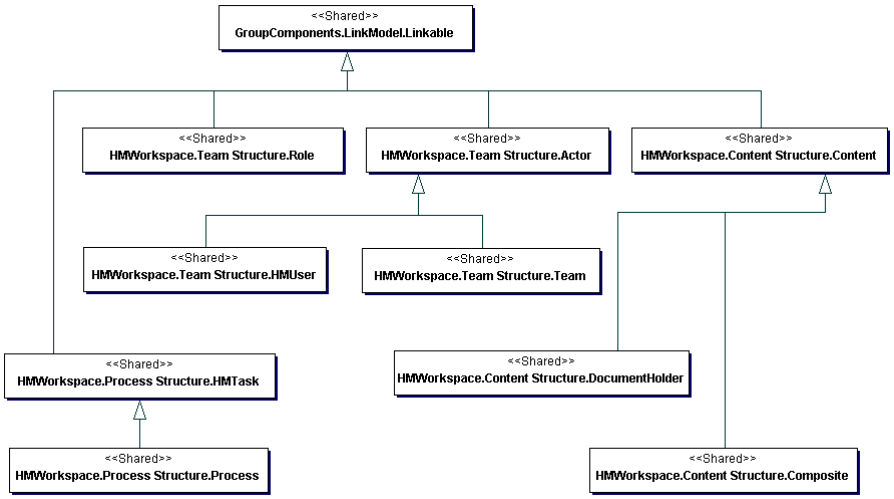

Figure 6.4: Hypermedia Object Structure - Overview

tent Structure, the Team Structure and the Process Structure allow creating nested structures from the hypermedia model. The graphical editor also allows creating the Hypermedia links, according to the current hypermedia schema, which are not directly expressed in the object model. All accessible link types are available and can be instantiated in the workspace simply by drawing a connection between objects in the workspace.

Similar to the Workspace Editor are the editor components for the three areas of the hypermedia model: the *Process Editor*, the *Composite Editor* and the *Team Editor*. A speciality of the Process and Composite editors are that they allow the inclusion of DocumentHolders: Special content objects which can store arbitrary objects from the shared object space. The DocumentHolder objects in the shared workspace provide access to the DyCE Task model - they allow direct invocation of the DyCE Tasks published for the class of which the DocumentHolder content is an instance. So, if for instance the DocumentHolder contains a reference to a Diagram, the DocumentHolder uses the services provided by the ComponentBroker (see chapter 5.1) to discover the DyCE Tasks available for the Diagram class. These Tasks can then be invoked within the same DyCE session or in a new DyCE session. By including DocumentHolder instances within the co-operative hypermedia workspace, the Process and Content structures can be used to structure a shared workspace which provides access to other shared artifacts (of arbitrary type).

6.2.2 Usage Scenario for Shared Hypermedia Workspaces

One possible usage scenario for the shared hypermedia workspaces is a joint writing process collaboratively structured and then performed by a group of users working on a scientific paper.

The group is modeled using the Team Structure. Sub-teams can be introduced in order to distinguish between the authors, colleagues who help in

reviewing the paper and external reviewers.

The structure of the paper is modeled using the nested composite structures from the Content Structure part of the shared hypermedia model. The paper is subdivided according to the chapters: Introduction, Chapters 1 through n and References.

The Process Structure part of the shared hypermedia model is used to design a process model for the shared writing task: Process substructures are created for gathering of related work, joint writing, reviewing by colleagues, managing of literature references and finally external peer review. Within the process structure, elements from the team model are used to assign responsibilities to each user within the process model. Additionally, elements from the composite structure (such as the composites created for each chapter) are included in the process structure to denote that a specific step of the process (e.g. writing chapter one) is associated with a certain composite of the paper (chapter one).

The authors can then proceed to work on the paper, each accessing the composites created for the chapter on which he or she is supposed to work. In a writing process stretching over multiple days or weeks (as writing scientific papers usually does), the search tool can be used to keep track of which composites are currently being worked on, which ones are assigned to a user, etc. Within the composites, DocumentHolder instances can be created for the different elements used in the chapters and a variety of DyCE components can be used to create the contents: figures can be created using a diagram editor component, the text body can be written using a shared text editor component. Material from external sources on the World Wide Web can be referenced using the shared HTML Presentation component.

6.3 Collaborative Extended Enterprise Engineering

In the EU project EXTERNAL, DyCE has been used as the basis for the development of a system for distributed collaborative engineering and operation of extended enterprises (see [WHRT00b]). Here, the collaborative hypermedia workspace implementation was reused and extended with extended enterprise semantics used for creating and executing Extended Enterprise (EE) models. The central component of the collaborative extended enterprise engineering system ist the XCHIPS component (see figure 6.5).

By basing the shared EE modeling system on the flexible hypermedia-based shared workspace, and thus on the DyCE component model, the EXTERNAL implementation was able to achieve significant re-use of previous components, providing a running prototype at a very early stage in the project. This early prototype was deployed very early on in the project to the project partners and was then extended iteratively. Over the course of the project (which is an ongoing project at the time of writing this), the EXTERNAL suite has been extended with additional Groupware Components, which were added to the already deployed system as Groupware Components. These new components include an integration of Microsoft NetMeeting conferencing and application sharing tool, an integration of Microsoft Internet Explorer for access to arbitrary Web content (exceeding the capabilities of the HTML browser already available

as a DyCE component), an extended shared hypermedia-based EE modeling and operation workspace, an integration with external tools from other project partners as well as several other components.

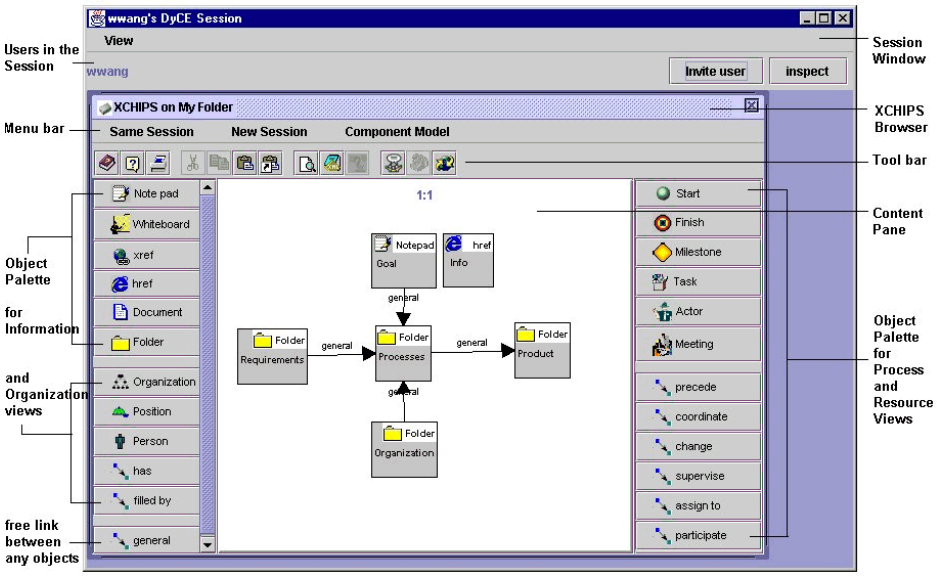

Figure 6.5: The XCHIPS component

By re-using the `DocumentHolder` model from the shared Hypermedia Workspace, EXTERNAL workspaces are directly extensible with any new DyCE components that become available. References to new shared data objects can be inserted into the workspace at any point in time, tasks can be invoked on these shared objects and new components can be introduced into the collaboration setting. This approach, along with a re-use of already existing DyCE components, made it possible to deploy the EXTERNAL prototype at such an early stage in the project, while still maintaining the prototype and adding extensions to the already deployed system.

6.4 Lessons learned from use of DyCE

When using DyCE as the basis for the development of the components and environments described above, the advantages of the flexible component-based development approach became apparent.

Re-use of previously developed components was directly supported and became one major design focus of the developers involved in these projects. For instance, providing a general task-aware container (termed *document holder*) in the shared hypermedia workspace provided a straight-forward way for integrating existing models - and thus Groupware Components - into the shared workspace. The document holder allows the users to invoke all published tasks on a given shared object, thereby providing direct access to other components, which weren't a direct part of the shared hypermedia workspace. In this way,

e.g., it was possible to use a previously developed shared text editor and a shared diagram editor in order to use the shared hypermedia workspace as a tool for a collaborative writing process.

In the past year, the DyCE framework has been used regularly as basis for the development of new Groupware Components by around ten developers (members of the IPSI research staff as well as students). In the course of this work, a wide range of components have been developed, including components for

- shared multimedia presentations using multi-point access to streaming media (see [XFT+01]),
- shared multi-point multimedia conferencing,
- collaborative games,
- shared object-oriented sketches,
- collaborative brainstorming, etc.

As development continues, the set of available and reusable Groupware Components is steadily increasing. By making developers aware of the potential for component re-use, the components built on DyCE can be made reusable and reuse-friendly. In this way, any of the above components can e.g, be used to extend the EXTERNAL shared modeling tool. Should a shared multimedia presentation facility be required in an EXTERNAL shared workspace, the shared object can be added to the EXTERNAL model (by the end-users) and the related Groupware Components can be invoked on this object. In this way, end-user extensibility and modification of their work environment has been achieved.

With regard to the developer requirements shown in chapter 2, a number of experiences were gathered in the course of using the DyCE framework. Over the last couple of months before writing this thesis, several new developers have been introduced to the DyCE framework and are now using the DyCE framework in their everyday project work. The experiences gathered from the introduction of these new developers gave some good insights into the strengths and weaknesses of the DyCE development framework.

The potential for reuse of existing programming knowledge (RD1) was first experienced with the development of a shared multi-user chat component. This component was developed by an inexperienced DyCE developer (but an experienced Java developer) by taking an existing Chat Applet (already conforming to the Model-View/Controller abstraction due to its use of the Swing GUI framework as a basis) and replacing the data model used by the Applet with a DyCE shared data model. This adaptation was done within the course of one day after starting to use the DyCE framework. A couple of the existing components, such as the shared HTML presentation component, demonstrate the value of a common shared data model which can be reused across multiple components (RD2). In the presentation example, a single shared presentation model is used across several components. The shared data models, in combination with support for transparent sharing of data objects (RD3), without having to implement command- or event-based protocols between the components or other communication means, has been observed to simplify design and development discussions and to aid distributed development of components in a team

of several developers. After the developers agreeing on a common data model, complex collaborative environments could be decomposed into several components, which were developed independently by different developers. Due to the unifying nature of the common data model, the independent development and subsequent integration of the components have become comparatively easy. The support for server-side components (RD9) has greatly simplified the design of a set of components for collaborative learning, where functionality regarding the control of learning protocols (structured approaches to tackling collaborative tasks in learning situations) could be implemented in a server-side component, thus resolving complex scheduling and control issues which complicated a previous design. Over the course of the next months, the support for combining existing components into configurations (RU10) will be used to put together an integrated (but adaptable) learning environment for language learning.

In the course of the projects based on DyCE so far, little experience has been gathered with end-user tailoring and combination of the Components. So far, use of the DyCE framework has focused on use of the development framework by developers and use of the resulting Components by end-users. It is anticipated that a new project, scheduled to begin in 2001, will exploit DyCE features for combining Groupware Components into new system prototypes. In this new project, rapid prototyping of Components for computer-based language training will necessitate faster iterations of component development and experiments with new designs. One of the aspects under examination is the layout and composition of tools for language learning and group awareness. Using the component configuration functionality provided by DyCE, it is envisioned that changes to the working environment can be made by end-users or other non-developer members of the research staff, simply by changing the composition of the components available.

Chapter 7

Discussion

This chapter will summarize the main aspects of the thesis and compare the presented approach to the requirements stated in chapter 2. A comparison to the state of the art will motivate the list of the thesis' main contributions. The thesis concludes with a number of topics for future research which can be based on the work presented here.

7.1 Summary

Today's flexible team structures and evolving collaboration processes, as encountered e.g. in creative working environments and in the context of Extended Enterprises, place high demands on the groupware environments aiming to support these collaborations.

The main requirement, from which other requirements can be derived (both in terms of users' as well as developers' requirements) is to have available flexible, extensible groupware environments which can be modified and extended by end-users as well as by developers at run-time, as the users' collaborative processes evolve. This thesis aims at making a contribution to the design, development and deployment of such collaborative environments.

The approach taken is that of component-based development: collaborative environments can be constructed from Groupware Components - interactive collaborative components which can be dynamically deployed, invoked and combined into new configurations which can then be used collaboratively. In order to support the development of Groupware Components, a Java-based development framework has been designed and implemented. This framework, called DyCE (Dynamic Collaboration Environment), is presented in this thesis along with the underlying concepts which were developed to allow the design and implementation of flexible collaboration environments.

The separation between the implementation of components and the shared domain objects manipulated by the components aids the combination of different components and the use of shared artifacts in evolving collaborative processes. Component implementations are stored in a central Component Repository and are dynamically downloaded to the client machines when they are required.

The flexible combination of components and the runtime extensibility of the resulting system are supported by the task-based programming model presented

in chapter 4. Using this model, components are bound to shared object classes and are retrieved from a Component Broker when a specific task is invoked on a shared data object. As has been discussed in section 4.6, this loose binding between components and shared objects allows different components to be invoked on common shared data objects and allows a selection of components based on different types of end-points, different end-user requirements, etc.

Groupware Components can be linked in a number of ways, depending on user preferences and requirements. First of all, components can invoke other components (indirectly) by performing a certain task which is published in the Component Repository by another component. Multiple components can access (display and modify) the same shared data items (potentially in different ways). The DyCE framework ensures that the components are notified and that they are kept in a consistent state. Additionally, object-based event propagation is available between the components (see section 4.5). This allows components to send events to all components currently accessing a shared object. The combination of these approaches gives developers great flexibility in designing the Groupware Components and collaboration environments based on them.

End-user tailorability is supported by providing a graphical tailoring interface which allows users to combine different components. These combined components (Configurations) are also modeled as shared objects, can be shared between users and allow access to groups of components as composed entities.

7.2 Comparison to requirements

This section will present an overview of the way in which the requirements from chapter 2 have been addressed in the system design and implementation.

7.2.1 End-User Requirements

RU1 - Access to shared artifacts and collaborative tools: The shared artifacts, which are modeled as shared objects, are stored persistently in a shared object database. The artifacts, along with the implementation of the components, are distributed in the system at runtime, as soon as they are required.

RU2 - Computer guidance in selecting appropriate tools: This is supported through the use of the task-based programming model. Using these tasks, users can invoke components on shared objects and can quickly find out which components are available for use with a given shared object. Learning how to use the available components is supported through provision of hypertext-based online help for each component.

RU3 - Provision of Group Awareness: Group awareness is available on the task and session layer. The system can provide information about which users are accessing which objects in which sessions (i.e. together with which other users). More sophisticated group and process awareness mechanisms have been built on top of the DyCE framework, e.g. in the Shared Hypermedia Workspace presented in chapter 6.

RU4 Support for synchronous as well as asynchronous collaboration: Synchronous collaboration is supported by the tight coupling between the shared objects and facilitated by the automatic view updating at the GUI level. Dynamic object replication, in combination with consistency maintenance

based on transaction management, as well as the object-based event broadcasting channels, support synchronous collaborations across sites and between components. Asynchronous collaboration is supported by the persistent Object Storage, which is used to store object structures between editing sessions (and therefore serves also as a basis for the exchange taking place in asynchronous collaboration).

RU5 - Ubiquitous access to collaboration environment: The ability to deploy DyCE workspaces over the Web, by using DyCE's integrated Web server as a collaboration-extended Intranet server makes shared workspaces available in many environments.

RU6 - Multiple simultaneous collaboration modes and transitions between them: Users can invoke individual (non-shared) editing or viewing sessions on any shared objects. At any point, additional users can be invited into the editing session, thereby making the transition from private to group work. The design decision has been made to allow concurrent access to shared objects also in multiple private sessions, thereby in effect providing coupled collaboration across sessions. A locking mechanism (or other means to restrict shared objects to private use) could be a useful extension.

RU7 - Dynamic extensions of the collaboration environment: Components can be added to the system (to the Component Broker) at runtime, without the need to shut down running clients and without disrupting running collaborations in any way. Newly added components are registered in the Component Broker using their task interface and are immediately available to all users for collaborative use.

RU8 - Coupling of different tools: This requirement is also fulfilled by the task-based programming model, using tasks to provide a binding between components and shared objects (or object classes). The division between component implementation and object-oriented shared data model allows invoking several different components on common shared data objects.

RU9 - Support for mobile work: Mobile appliances have certain restrictions, e.g. in terms of memory and display size. By extending the task-based programming model to contain information about the infrastructure constraints that apply to a component (e.g. the information that a certain component is "PDA-Compliant"), such devices can be supported and can take part in complex collaborative work situations. Also, the dynamic nature of the system, the fact that components are downloaded when required and shared data objects are dynamically replicated, support the use of such mobile appliances with limited resources in collaborative situations.

RU10 - Combination of existing tools (End-User tailorability): Using the interactive configuration editor, users can combine the available components into configurations, which can then be shared with other users. Also, by introducing new components into the system and invoking components on any shared data objects (through the task information published by the component), users can flexibly extend and adapt their collaborative work environments.

RU11 - High system performance: System performance is addressed in the DyCE framework by the use of dynamic replication mechanisms (which results in only the currently required objects being replicated to a client's machine) and the use of optimistic concurrency control for the users' actions. Also, the UDP-based network layer has been developed specifically in order to improve scalability and performance of the overall system. No systematic evaluation

of system performance and scalability has been done so far, so that concrete performance measurement results cannot be presented in this thesis. Such performance evaluations remain as future work.

7.2.2 Developer Requirements

RD1 - Reuse of existing programming knowledge and experience: Every new development framework requires some time to learn and to become proficient in (see [Pre99]) - DyCE is no exception. Yet, the Groupware Components are based on Java's Swing component set; in fact, they provide a natural extension to these models/toolkits. The use of Java RMI for the object communication also allows reuse of programming experience. The (necessary) design decision to instate a separate type and variable system (the Slots) makes it necessary for experienced Java developers to learn additional concepts.

RD2 - Reusable data models: The decoupling of data objects and Groupware Components presented in chapter 4 makes the shared data model reusable for different components.

RD3 - Transparent sharing of collaboration artifacts: The transparent sharing is achieved through the tight binding between the Object Management and Replication Management and the fact that objects are accessed through the appropriate interfaces, which automatically make use of the Replication Management Service and the Object Management Service.

RD4 - Support for shared as well as private data items: Shared data objects are modeled using the `RObject` and `Slot` classes from the DyCE framework. These objects can automatically by replicated by the system. Non-Shared (local) data items are available by using Java's regular instance and class variables.

RD5 - Access to co-operation information when required: Using the information available from the Session Manager and Component Broker as presented in section 5 allows the creation of group awareness mechanisms such as the user lists present in the sample Groupware Components developed for the group-aware shared hypermedia workspace, described in section 6.1.

RD6 - Deployment of newly developed components: The deployment is facilitated by the Component Management Service GUI which allows importing new Groupware Components over the net, as well as by the feedback mechanisms present in the Component Broker and Component Desktop UI, which give users information about the set of available components and make newly deployed components usable instantly, without the need to restart the client application, etc.

RD7 - Technical scalability of the solution: The collaboration framework presented in this thesis imposes scalability along multiple dimensions. The main determining factors appear to be the size of the shared object space and the number of shared data objects used concurrently as well as the number of simultaneously connected clients. Through the use of dynamic replication and reference to collaboration session information for object discarding, etc., a graceful degradation of performance is expected. On order to address the scalability issues with respect to the number of connected clients, an asynchronous network communication layer based on UDP has been implemented. No measurements of performance issues have been done, though.

RD8 - Support for integration into external architectures: This integration is supported by the HTTP-based access to information from the collaboration situations. Using these standard access mechanisms, DyCE components can be embedded into Web pages, etc.

RD9 - Support for server-side components: Server-side components are supported by a specific base class of the system. Like all other components, these server-side components can access shared objects and publish task information. Invoking such a component (either through another component performing a task or by user interaction) opens the component on the server, with access to shared data objects. A server-side user interface can provide interactive elements (e.g. for monitoring or interaction purposes).

7.3 Comparison to related work

Chapter 3 has presented a number of systems for developing groupware. The comparison to these systems will concentrate on the main deficits identified in those systems with the same focus as DyCE, namely the development and deployment of flexible, component-based groupware (see section 3.4).

The Visual Component Suite system does not give the end-users access to the components which make up the collaborative system. In this approach, users cannot extend the system by adding new components (RU7), not can different components be flexibly coupled on common shared data (RU8), as is supported by the `ModelObject` part of the DyCE framework. Since there is no end-user tailoring support such as the Configuration Editor (see section 5.5), end-users cannot flexibly combine existing components (RU10).

The Disciple system lacks a common shared data model, which can be reused across different components (RD2). Therefore, the system cannot support the coupling of different tools on shared data (RU8). The deployment of newly developed components (RD6) is supported by being able to load JavaBeans components into the shared workspace from Web locations, similar to the functionality supported by the HTTP server part of the Component Broker. Disciple has no equivalent to the task-based programming model though, so users are not provided with support in selecting the appropriate tools for the task at hand (RU2).

The approach to sharing taken in the TeamComponents system (multicast method invocation) results in lack of support for combining different components (RU8). Also, due to the fully distributed nature of the DreamTeam environment underlying the TeamComponents, no support is provided for server-side components (RD9).

The EVOLVE platform provides a high degree of support for end-user tailorability of the collaboration system (RU10), dynamic extension of the collaborative system (RU7) and easy deployment of components and newly tailored instances of components (RD6). The approach of encapsulating the collaboratively edited data items *in* the components and providing data sharing only through synchronized RMI (Remote Method Invocation) access to the shared data items stored in one location, will result in low system performance (not fulfilling requirement RU11), and prohibits the coupling of different tools on the same shared artifact (RU8). In order to address these requirements, the DyCE framework uses a data replication approach and uses transaction management

and concurrency control to address concurrency issues. Also, it remains unclear how the EVOLVE platform can support asynchronous collaborative work (RU4), with users accessing shared objects even if the other users' components (which hold that shared artifact) are not currently available.

The GROOVE system provides a collaboration environment within which workspace contents (shared data items and tools) are fully replicated between users. GROOVE provides comprehensive support for tool deployment (RD6) and synchronous as well as asynchronous collaboration (RU4). The GROOVE approach to data sharing is to encapsulate data changes into "command" objects, which can be serialized and broadcast to all connected clients. This means that GROOVE applications have to be specifically developed with collaboration in mind and there is no transparent sharing of data items (RD3) - the GROOVE tools needs to accept and handle the command objects which encapsulate application-specific behaviour. Also, there is no explicit data model which could be reused across components (RD2); due to this fact, GROOVE does not support the coupling of different tools on the same shared item (RU8). GROOVE does not provide any support for dynamic replication, as is done in DyCE - the entire contents of a shared workspace are replicated to all connected users. It is unclear how this approach scales with workspace size (RD7). Especially, the initial replication of a new shared workspace has been experienced to take a rather long time (this also depends on available network bandwidth), which can hinder spontaneous collaborations. The DyCE framework, on the other hand, only replicates those objects which are currently required, allowing users to start working with a new shared workspace quickly, and replicating additionally required objects on-demand, as work in the shared workspace progresses.

7.4 Contributions to the state of the art

The work presented in this thesis advances the state of the art in the following areas:

Component-based groupware: The approach of constructing extensible groupware from large-scale collaborative components aids flexibility and extensibility of groupware environments. The use of the DyCE framework in a number of projects has demonstrated the applicability and benefits of this approach.

Extensible component-based environments: The concept of loose coupling between the interactive components and the data objects manipulated by them, in the form of the proposed task-based development framework, with its extensions for multi-platform components, can also be used beneficially in the design of single-user extensible environments.

Interactions in component architectures: The ability to combine different (kinds of) components in collaborative sessions, with the possibilities offered by dynamic, run-time extensions, aid the creation of flexible collaborative environments, well-suited to fulfill the requirements posed by today's highly dynamic team-work settings. The ability to couple different types of components on the same shared data objects, with object consistency and object-based event broadcasting mechanisms supporting the coupling of the components, allows the decomposition of the collaborative system into individually deployable and exchangeable functional modules.

Modeling component-based groupware: The proposed extensions of the UML notation aid the design and discussion of groupware environments, especially when building groupware in teams of several developers.

7.5 Future Research

In its current state, the DyCE framework can be (and is) used to develop a variety of components for different collaborative activities. A number of issues remain which should be investigated further.

7.5.1 Support for mobile disconnected work

Mobile disconnected operation (use of mobile devices which are not permanently connected to the network) is one of the scenarios in supporting mobile users. By integrating a versioned object model and by supporting intelligent caching of Groupware Components, support for mobile disconnected work could be provided.

An important part of such research would certainly be in how far disconnected (or intermittently connected) users can be tightly integrated into collaboration processes (e.g. by using intelligent agents to represent the interests of the users while they are disconnected).

7.5.2 Extension of Task framework

Currently, the Tasks presented in this thesis carry little to no syntactic and semantic information. By extending the Task framework to provide syntactic and semantic information (e.g. which tasks are allowed to which users within the context of which other tasks, how many participants are required for a certain task, etc.), more meaningful Task structures could be constructed.

Such an extended Task framework could also be used in other collaboration settings, e.g. as the basis for the POC ("points of collaboration") model described in [WP00].

7.5.3 Support for time-dependent media

Currently, there are no hard constraints placed on the timing and distribution of transactions. The solution chosen in the DyCE framework is suitable even for highly interactive Groupware Components.

For time-dependent media (video or audio clips or even multimedia teleconferencing), timing and synchronization mechanisms need to be developed which allow the modification and presentation of data in collaborative tools to be synchronized with media streams. Additionally, mechanisms need to be developed which ensure that all users in a collaborative session are receiving synchronized media streams (played out at the same time). Lack of such synchronization could severely disrupt collaboration and coordination between the users.

Envisioning a shared workspace as the one presented in the experiences chapter, enhanced with the ability to store videos or audio clips in the shared workspace, the additional problem of late-joiner support arises. How can a late-joiner "catch up" with the group currently using the shared workspace? How can a

sensible point of entry be determined which permits a meaningful continuation of collaboration in the session?

The issue of time-dependent media is becoming more important as we are faced with the development of distributed collaborative learning environments. A large part of learning material (e.g. in the case of language learning) is available as movie or sound files and collaborative learning support is becoming increasingly important in today's business and education settings.

7.5.4 Improved network structure

The current network structure of the DyCE framework uses a star-shaped topology with multiple clients connecting to a single central server. The system performance and scalability can be improved by implementing a more extensible network structure, e.g. using a hierarchical, tree-type network topology, with intermediate caches, routers and multiplexers. Such a network topology can greatly benefit the use of DyCE-based collaboration environments in corporate and other large-scale installations. The router modules required for such network structures should also make DyCE-based environments usable in network environments with firewalls, e.g. by providing so-called application-level gateways which can allow DyCE network traffic to be processed and forwarded through firewalls.

Additionally, object mobility and load balancing mechanisms can be explored, which allow a more optimal distribution of shared objects and load between multiple servers.

As a first step towards this goal, a number of performance measurements of the network transfer, and its influence on overall system performance should be performed.

Appendix A

Implementation of a sample Groupware Component

Shown here for illustrative purposes is the source code of a sample component and its associated data model.

A.1 The sample component

The sample component allows interaction with two data values (which form the data model) - a String value, edited in a text entry field, and a numeric value, modified through two buttons (for increasing and decreasing the numeric value). These values can be modified collaboratively, changes to the data values are displayed to all connected users. he user interface of the sample component is shown in figure A.1

Figure A.1: GUI of sample component

The following sections show the source code of the domain data model class developed for this component and the source code of the component itself. The

source code is fully commented in order to illustrate the main aspects of developing DyCE components and data models.

A.2 The data model class

```
package Demo;

import GroupComponents.ObjectModel.ModelObject;
import GroupComponents.ObjectModel.RObject;

public class DemoModel extends ModelObject
{

    /** "Empty" Constructor */
    public DemoModel() { super(); }

    /** Constructor for passing an existing RObject instance.
    Used when replicating objects, when we fetch an RObject
    instance from the server and locally need to wrap it
    in an appropriate ModelObject subclass instance.
    ote: We only replicate RObjects. The ModelObjects are
    created locally. This is possible, since each RObject
    knows its appropriate ModelObject class and since
    ModelObject subclasses cannot add any replicated
    state outside of the RObject (instance or class variables
    are considered to be local and non-replicated). */
    public DemoModel(RObject o) { super(o); }

    /** public getter for the counter slot. Convert
    Integer value to int for easier handling. This method
    needs to be called within a transaction (this can also be
    a displaytransaction, since no contents are modified). */
    public int getCounter()
    {
        Integer i = (Integer)(get("counter"));
        return i.intValue();
    }

    /** public setter for the counter slot. This method
    must be called in a transaction, which can't be a
    DisplayTransaction, since slot contents are modified. */
    public void setCounter (int i)
    {
        set ("counter", new Integer(i));
    }

    /** Public getter for the counter slot, returning the
    Integer instance */
    public Integer getCounterInt()
```

```
    {
        return (Integer)get("counter");
    }

    /** Initialize the Demo Object's slots */
    public void initializeSlots()
    {
        /* A slot is described by a slot name and an initial
        Slot content object. This slot content object defines
        the slot value's type. The definition of a slot
        called counter, below, is similar to defining
        an instance variable as
          Integer counter = new Integer(1);
        If DemoModel were to extend a ModelObject class
        other than ModelObject itself, we would need to call
        super.initializeSlots here. We don't in this case, as
        the ModelObject base class doesn't have any slots. */
        newSlot ("counter", new Integer(1));
        newSlot ("text", new String ("Hello World"));
    }
}
```

A.3 The component implementation

```
package Demo;
/** Each Component should be in a package. Its model
clas should be in the same package. A package can
hold multiple components (actually, this i what is to be
expected in complex systems). */

import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.Vector;
import javax.swing.JLabel;
import javax.swing.JButton;

import GroupComponents.swing.DyCETextField;

import GroupComponents.ComponentModel.MobileComponent;
import GroupComponents.ComponentModel.ObjectComponentTask;
import GroupComponents.ObjectModel.ObjectChangeListener;
import GroupComponents.ObjectModel.ObjectChangeEvent;

import Demo.DemoModel;

/** Sample component used to initeract with a
DemoModel instance. It displays the counter part of
the DemoModel in a label and provides two buttons
```

```
for increasing and decreasing the counter. The text
part of the model is jointly edited in a shared textfield
(provided as a convenience widget in DyCE). */
public class DemoComponent extends MobileComponent
    implements ObjectChangeListener, ActionListener
{
    /** The label for displaying the counter content from the model */
    JLabel counterLabel;

    /** A DyCE-enabled Textfield for the string content from the model */
    DyCETextField textfield;

    /** Component Constructor, only used for initializing
    REALLY important state. We don't do User Interface
    stuff here, since that is only done in the prepareGUI
    method, which is called by the DyCE framework at the
    appropriate time. */
    public DemoComponent()
    {
        super();
    }

    /** Set up the component's GUI elements. The DyCE
    framework ensures that at this point, setModel has
    already been called so that the component knows
    its Model. This might be necessary in initializing the GUI. */
    public void prepareGUI()
    {
        /** Since we're accessing the model when creating
        the GUI, we'll need to do this within a DisplayTransaction */
        beginDisplayTransaction();
        setLayout (new GridLayout(3,2));
        counterLabel = new JLabel(getDemoModel().getCounterInt().toString());
        add (new JLabel ("Counter : "));
        add (counterLabel);
        add (new JLabel ("Text : "));

        /* Set up a DyCETextField and bind it to the slot
        called "text" in the model. The TextField will take care
        of interacting with the slot contents and keeping current. */
        textfield = new DyCETextField (10,getModel(), "text");
        add (textfield);

        JButton b1 = new JButton ("-");
        b1.addActionListener(this);
        add (b1);

        JButton b2 = new JButton ("+");
        b2.addActionListener(this);
        add (b2);
```

```
        commitDisplayTransaction();
    }

    /** Return the task bindings for this component. The
    DemoComponent publishes the Task "demonstrate" on a
    ModelObject. This information serves as the basisfor
    the dynamic binding between ModelObject instances
    and GroupComponents. */
    public void giveTaskBindings(Vector taskList)
    {
        /* Use method chaining to also fetch the Task bindings
        from superclasses. */
        // super.giveTaskBindings(taskList);

        /* Each Task published by a component is modelled
        by an instance of ObjectComponentTask, which contains
        a Task name, the full name of the ModelObject subclass
        to which this task applies and the name of the component
        class itself. */
        taskList.addElement (new ObjectComponentTask (
            "demonstrate",
            "Demo.DemoModel",
            getClass().getName()));
    }

    /** Helper method for accessing the component's Model */
    protected DemoModel getDemoModel()
    {
        return (DemoModel)(getModel());
    }

    /** Notification method to receive ObjectChangeEvents.
    Overridden from ObjectChangeListener interface.
    Called whenever the contents of this component's
    Model change. */
    public void objectChanged(ObjectChangeEvent ev)
    {
        /* We need to do the GUI updating within a
        display transaction. */
        beginDisplayTransaction();
        counterLabel.setText(getDemoModel().getCounterInt().toString());
        commitDisplayTransaction();
    }

    /** Increase the ModelObject's counter value */
    protected void increase()
    {
        /* This needs to be done in a transaction. Giving the
        transaction an identifying name might later help in
        debugging. */
```

```
        beginTransaction("increase");
        getDemoModel().setCounter(getDemoModel().getCounter()+1);
        commitTransaction();
        /* We're not doing any GUI refreshing, etc. here. This will
        be done when the transaction commits. DyCE will have gathered
        up all objects affected by a transaction and will notify
        their ObjectChangeListeners. That's why we have our objectChanged
        method which does the updating.
        */
    }

    /** Decrease the ModelObject's counter value */
    protected void decrease()
    {
        /** Similar to increase(). See there for details. */
        beginTransaction("decrease");
        getDemoModel().setCounter(getDemoModel().getCounter()-1);
        commitTransaction();
    }

    /** Notifier method from ActionListener */
    public void actionPerformed(ActionEvent e)
    {
        if (e.getActionCommand().equals("+"))
            increase();
        if (e.getActionCommand().equals("-"))
            decrease();
    }

}
```

Appendix B

UML Overview

This appendix presents a brief overview over the notation elements of the Unified Modeling Language (UML), as used in the body of this thesis. For more details about the UML, see [Fow00].

B.1 Standard UML notation

UML class diagram notation basics

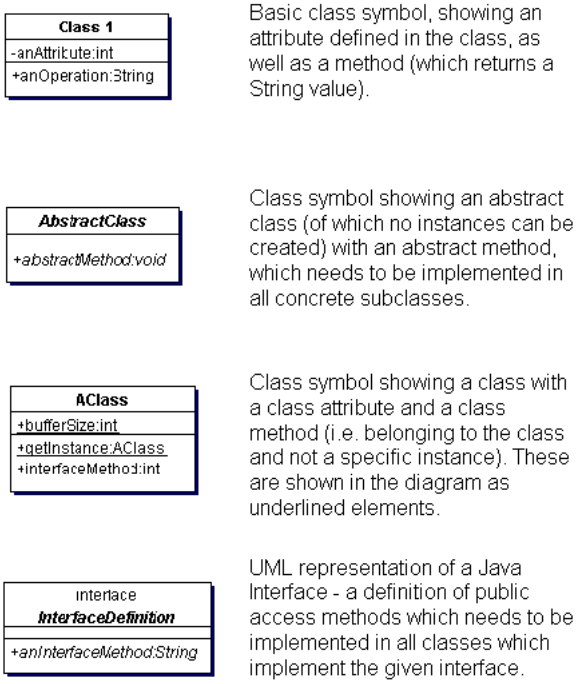

Figure B.1: Basic class representation

Figure B.2: Inheritance Relationships between classes

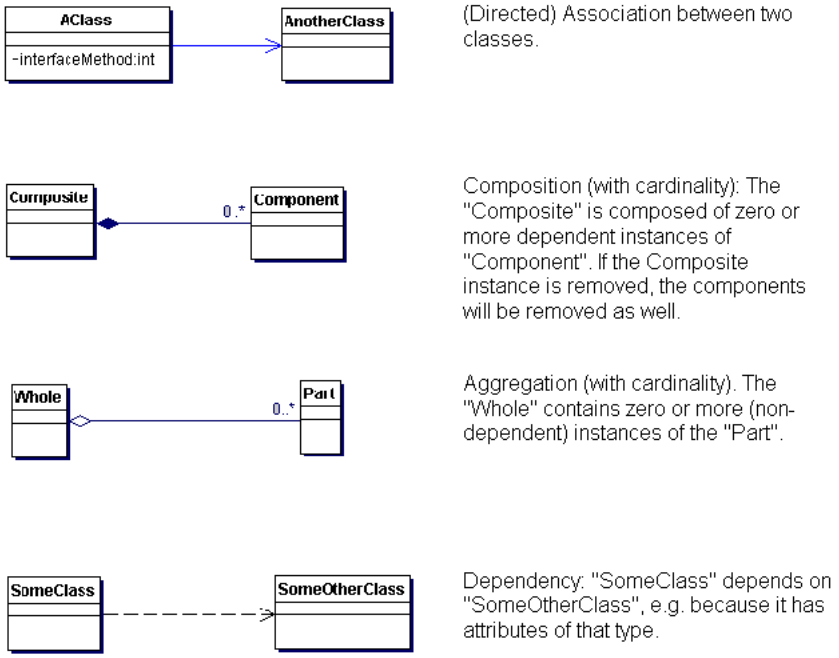

Figure B.3: Other Relationships between classes

B.2 UML extensions for component-based groupware

The following extensions to the UML notation have been defined in this thesis to model collaborative environments consisting of Groupware Components.

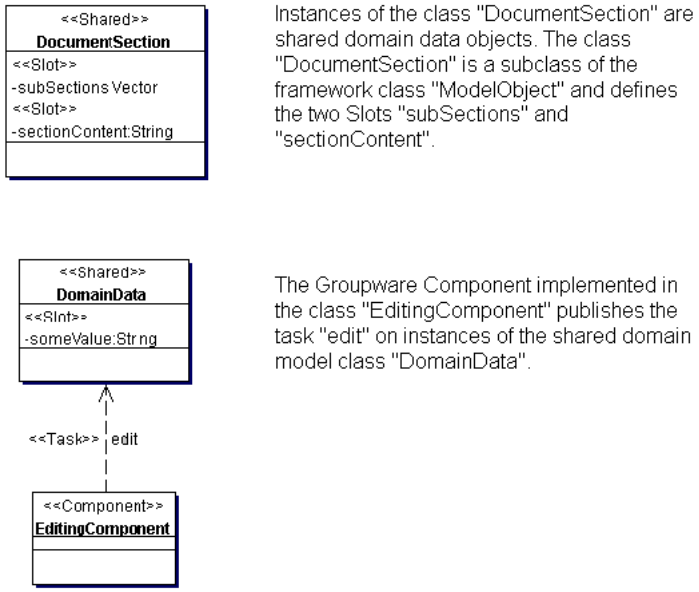

Figure B.4: UML extensions for modeling Groupware Components

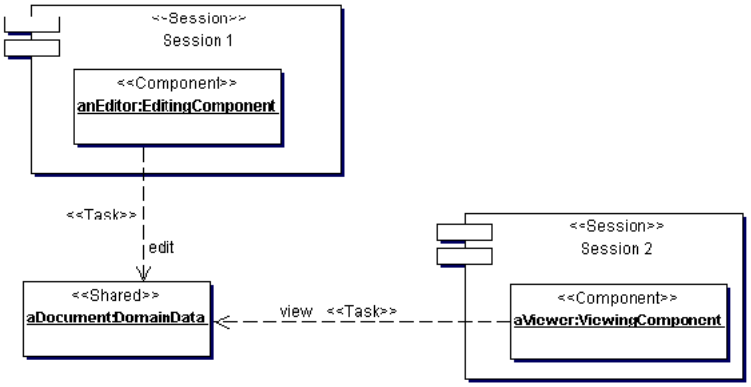

Figure B.5: UML extensions for modeling running sessions

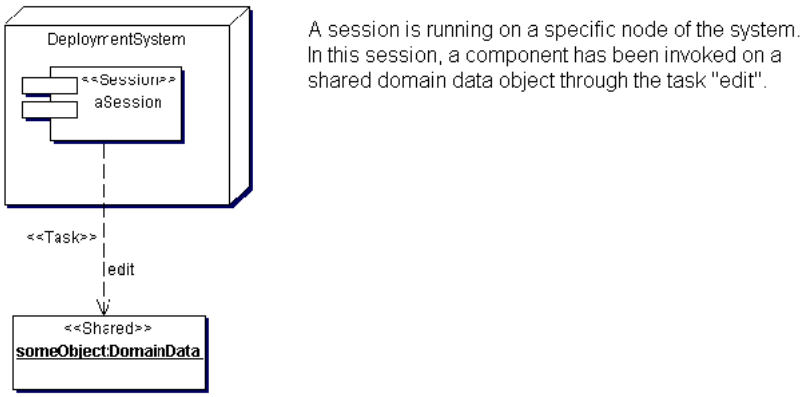

Figure B.6: UML extensions for modeling running sessions on a specific node

Appendix C

List of publications

The following publications have appeared so far (Dec. 2000) or have been accepted for publication:

Weigang Wang, Jörg M. Haake, Jessica Rubart and Daniel A. Tietze: Hypermedia-based support for cooperative learning of process knowledge. In: Journal of Network and Computer Applications (2000), 23. pages 357-379. Dec 2000.

Daniel A. Tietze, Till Schümmer: Kooperative Softwareentwicklung. In: Gerhard Schwabe, Norbert Streitz, Rainer Unland (Hrsg.): CSCW-Kompendium. Lehr- und Handbuch zum computerunterstützten kooperativen Arbeiten. Springer Verlag. 2001 (to appear)

Daniel A. Tietze, Ralf Steinmetz: Ein Framework zur Entwicklung komponentenbasierter Groupware. In: R.Reichwald, J.Schlichter (Ed.): Verteiltes Arbeiten - Arbeit der Zukunft (Proceedings der Fachtagung D-CSCW 2000), S.49-62, German Chapter of the ACM, Berichte, 54, Stuttgart, B.G.Teubner, September, 2000, ISBN 3-519-02695-3

Weigang Wang, Jörg M. Haake, Jessica Rubart, Daniel A. Tietze: Supporting Cooperative Learning of Process Knowledge on the World Wide Web. Proceedings of 26th EUROMICRO CONFERENCE, Maastricht, the Netherlands, 2000

Jan Schümmer, Thomas Tesch, Daniel A. Tietze, Ajit Bapat: Introducing Groupware in Administrative Environments - Experiences from the POLIWORK Project. In: Bullinger, H.-J., Ziegler, J. (Ed.): Human-Computer Interaction: Communication, Cooperation, and Application Design. Proceedings of HCI International '99, Munich, Germany, August 22-26, 1999, pp. 492-496, vol. 2, Mahwah, New Jersey, Lawrence Erlbaum Associates, 1999, ISBN 0-8058-3392-7

Jörg M. Haake, Daniel A. Tietze: User-Interface Erfahrungen im Informationsverbund Berlin-Bonn: Ein Bericht aus dem POLIWORK-Projekt. In: Ralf Reichwald, Manfred Lang (Ed.): Tagungsband: Kongress "Anwenderfreundliche Kommunikationssysteme", S. 61-77, Munich, Hüthig-Verlag, June 16-17, 1999, ISBN 3-7785-3937-X

Rolf Reinema, Daniel A. Tietze, Ralf Steinmetz: IMCE - An Integrated Multimedia Conferencing and Collaboration Environment. In: Proceedings of the First National CSCW Workshop (CCSCW98), pp. 95-100, Beijing, China, Publishing House of Electronics Industry, Dec., 1998, ISBN 7-5053-5066-8

Daniel A. Tietze, Ajit Bapat, Rolf Reinema: Document-Centric Groupware for Distributed Governmental Agencies. In: P. Fankhauser, M. Ockenfeld (Ed.): Integrated Publication and Information Systems: 10 Years of Research and Development. GMD-IPSI, pp. 75-90, GMD, Sankt Augustin, Selbstverlag GMD - Forschungszentrum Informationstechnik GmbH, 1998, ISBN 3-88457-968-1, (reprint of CAiSE paper)

Daniel A. Tietze, Ajit Bapat, Rolf Reinema: Document-Centric Groupware for Distributed Governmental Agencies. In: Pernici, B., Thanos, C. (Ed.): Proceedings of the 10th Conference on Advanced Information Systems Engineering (CAiSE'98), pp. 173-190, Pisa, Italy, June 8-12, 1998

Rolf Reinema, Daniel A. Tietze, Ralf Steinmetz: IMCE - An Integrated Multimedia Collaboration Environment. In: Poster Proceedings of the Sixth ACM InternationalMultimedia Conference (MULTIMEDIA'98), Bristol, 1998

Thomas Knopik, Daniel A. Tietze, Marc Volz, B. Paul, H. Speichermann, C. Wittinger: Towards a Collaborative Document Archive for Distributed Governmental Agencies. In: Lehner, F. and Dustdar, S. (Hrsg.) (Ed.): Telekooperation in Unternehmen, pp. 65-78, Gabler Edition Wissenschaft: Information Engineering und IV-Controlling, Dt. Univ.-Verl, Wiesbaden, 1997, ISBN 3-8244-6433-0

Ajit Bapat, Jörg Geißler, David Hicks, N.A. Streitz, Daniel A. Tietze: From Electronic Whiteboards to Distributed Meetings: Extending the Scope of DOLPHIN. In: Conference Video of the ACM 1996 Conference on Computer Supported Cooperative Work (CSCW '96), Boston, Massachusetts, November 16-20, 1996

Jörg M. Haake, Jörg Geißler, Daniel A. Tietze, Ajit Bapat: Hypermedia-based Collaboration Support . In: Wolf, M., Reimer, U. (Ed.): Proceedings of the First International Conference on Practical Aspects of Knowledge Management (PAKM '96), Vol. 2, Basel, Switzerland, October 30-31, 1996

Daniel A. Tietze, Marc Volz, Thomas Knopik: Kooperatives Dokumentenmanagement. In: GMD Jahresbericht 1995/96, pp. 51-57, Sankt Augustin, Selbstverlag GMD - Forschungszentrum Informationstechnik GmbH, 1996, ISSN 0949-2283

Bibliography

[AW96] Abdel-Wahab. Using Java for Multimedia Collaborative Applications. *Proc. 3rd International Workshop on Protocols for Multimedia Systems (PROMS'96)*, October 1996.

[BAB+97] R. Bentley, W. Appelt, U. Busbach, E. Hinrichs, D. Kerr, K. Sikkel, J. Trevor, and G. Woetzel. Basic Support for Cooperative Work on the World-Wide Web. In *International Journal of Human-Computer Studies: Special issue on Innovative Applications of the World-Wide Web*. Academic Press, 1997.

[BBD+91] R. Balter, J. Bernadat, D. Decouchant, A. Duda, A. Freyssinet, S. Krakowiak, P. Ledot, M. Meysembourg, H. Nguyen Van, E. Paire, M. Riveill, C. Roisin, X. Rousset de Pina, R. Scioville, and G. Vandome. Architecture and Implementation of Guide, an Object-Oriented Distributed System. *Computing Systems*, vol. 4, num. 1:31–67, 1991.

[BCM95] Georges Brun-Cottan and Mesaac Makpangou. Adaptable Replicated Objects in Distributed Environments. Rapport de Recherche 2593, Institut National de la Recherche en Informatique et Automatique, Rocquencourt (France), May 1995.

[BDMM99] Guruduth Banavar, Sri Doddapaneni, Kevan Miller, and Bodhi Mukherjee. Rapidly Building Synchronous Collaborative Applications By Direct Manipulation. In *Proceedings of ACM 1998 Conference on Computer Supported Cooperative Work (CSCW98)*, pages 139–148. ACM Press, 1999.

[BRS99] James Begole, Mary B. Rosson, and Clifford A. Shaffer. Flexible Collaboration Transparency: Supporting Worker Independence in Replicated Application-Sharing Systems. *ACM Transactions on Computer-Human Interaction*, Vol. 6, No. 2, June 1999:95–132, 1999.

[BSSS97] James Begole, Craig A. Struble, Clifford A. Shaffer, and Randall B. Smith. Transparent Sharing of Java Applets: A Replicated Approach. In *Proceedings of the 1997 Symposium on User Interface Software and Technology (UIST'97)*, pages 55–64, 1997.

[CGJP98] Annie Chabert, Ed Grossman, Larry Jackson, and Stephen Pietrovicz. NCSA Habanero (r) - Synchronous collaborative framework

and environment. `http://havefun.ncsa.uiuc.edu/habanero/Whitepapers/index.html`, 1998.

[Cha96] David Chappell. *Understanding ActiveX and OLE.* Microsoft Press, 1996. ISBN: 1-57231-216-5.

[Col97] David Coleman. *Groupware - Collaborative Strategies for Corporate LANs and Intranets.* Prentice-Hall, 1997. ISBN: 0-13-727728-8.

[DCS94] Prasun Dewan, Rajiv Choudhary, and Honghai Shen. An Editing-Based Characterization of the Design Space of Collaborative Applications. *Journal of Organizational Computing*, pages 219–239, 1994.

[DeS97] Alden DeSoto. Using the Beans Development Kit 1.0, September 1997.

[DGO+94] G. Dermler, T. Gutekunst, E. Ostrowski, N. Pires, T. Schmidt, M. Weber, and H. Wolf. JVTOS - A Multimedia Telecooperation Service Bridging Heterogeneous Platforms. *International Conference on Broadband Islands*, 1994.

[Dou95a] Paul Dourish. Developing a Reflective Model of Collaborative Systems. *ACM Transactions on Computer-Human Interaction, 2(1):40–63, March 1995.*, 1995.

[Dou95b] Paul Dourish. The Parting of the Ways: Divergence, Data Management and Collaborative Work. *Proc. Fourth European Conference on Computer-Supported Cooperative Work ECSCW'95*, 1995.

[EG89] C. A. Ellis and S. J. Gibbs. Concurrency control in group systems. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 18(2):399–407, June 1989.

[Eng97] Robert Englander. *Developing Java Beans.* O'Reilly and Associates, 1997. ISBN: 1565922891.

[ESSGS00] Abdulmotaleb El Saddik, Shervin Shirmohammadi, Nicolas D. Georganas, and Ralf Steinmetz. JASMINE: Java Application Sharing in Multiuser INteractive Environments. *International Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services 2000 (IDMS 2000)*, October 2000.

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616: Hypertext Transfer Protocol – HTTP/1.1. `ftp://ftp.isi.edu/in-notes/rfc2616.txt`, 1999.

[Fow99] Martin Fowler. *Refactoring - Improving the Design of Existing Code.* The Addison-Wesley object technology series. Addison-Wesley, 1999. ISBN: 0-201-485676-2.

[Fow00] Martin Fowler. *UML Distilled, Second Edition - A Brief Guide to the Standard Object Modeling Language.* Addison-Wesley, Reading, MA, 2000. ISBN: 020165783X.

[GM94] S. Greenberg and D. Marwood. Real Time Groupware as a Distributed System: Concurrency Control and its Effect on the Interface. *Research Report 94/534/03*, February 1994.

[Gor99] H.T. Goranson. *The Agile Virtual Enterprise : Cases, Metrics, Tools*. Quorum Books, October 1999. ISBN: 1567202640.

[Gre91] Saul Greenberg. *Computer-supported Cooperative Work and Groupware*. Academic Press, London, 1991.

[Gre98] S Greenberg. *Real Time Distributed Collaboration*. Kluwer Academic Publishers, 1998.

[Gro00] Groove. Peer Computing Comes to the Internet - Introducing Groove. `http://www.groove.net`, 2000.

[Haa99] Jörg M. Haake. Facilitating Orientation in Shared Hypermedia Workspaces. In *Proceedings of ACM Group'99*, Phoenix, Arizona, November 1999.

[HHB96] A. Helal, A. Heddaya, and B. Bhargava. *Replication Techniques in Distributed Systems*. Kluwer Academic Publishers, 1996.

[HK97] Markus Horstmann and Mary Kirtland. DCOM Architecture. `http://msdn.microsoft.com/library/backgrnd/html/msdn_dcomarch.htm`, July 1997.

[HKM98] J. Hummes, A. Kohrs, and B. Merialdo. Software Components for Cooperation: a Solution to the "Get Help" Problem. In *Proceedings of COOP'98: Third International Conference on the Design of Cooperative Systems*, May 1998.

[HM98] J. Hummes and B. Merialdo. Design of Extensible Component-based Groupware. *Computer Supported Cooperative Work - An International Journal*, 1998.

[HSH99] Merlin Hughes, Michael Shoffner, and Derek Hamner. *Java Network Programming, second edition*. Manning Publications Co., Greenwich, CT, 1999. ISBN: 1-884777-49-X.

[IBHS97] Philip Isenhour, James Begole, Winfield S. Heagy, and Clifford A. Shaffer. Sieve: A Java-Based Collaborative Visualization Environment. In *IEEE Visualization '97 Late Breaking Hot Topics Proceedings*, pages 13–16, October 1997.

[Ise98] Philip L. Isenhour. *Sieve: A Java-Based Framework for Collaborative Component Composition*. MSc Thesis, Virginia Polytechnic Institute and State University, February 1998.

[jav97] Java Beans API Specification. `http://java.sun.com/products/javabeans/docs/spec.html`, July 1997.

[JL96] Richard Jones and Rafael D Lins. *Garbage Collection : Algorithms for Automatic Dynamic Memory Management*. John Wiley & Son Ltd., 1996. ISBN: 0471941484.

[Joh91] R. Johansen. *Leading Business Teams*. Addison-Wesley, Reading, Mass, 1991.

[Joh97] Ralph E. Johnson. Frameworks = Components + Patterns. *Communications of the ACM*, Vol.40, No. 10:39–42, October 1997.

[KFCO99] K. E. Kerry Falkner, P. D. Coddington, and M. J. Oudshoorn. Implementing Asynchronous Remote Method Invocation in Java. In *Proc. Parallel and Real-Time Systems (PART'99)*, December 1999.

[KT99] Michael Koch and Gunnar Teege. Support for Tailoring CSCW Systems: Adaptation by Composition. In *Proc. 7th Euromicro Workshop on Parallel and Distributed Processing*, pages 146–152. IEEE Press, 1999.

[LJLR90] J. Chris Lauwers, T. A. Joseph, Keith A. Lantz, and A. L. Romanow. Replicated Architectures for Shared Window Systems: A Critique. In *Proceedings of the conference on Office Information Systems*, pages 249–260, 1990.

[LL90] J. Chris Lauwers and Keith A. Lantz. Collaboration awareness in support of collaboration transparency: requirements for the next generation of shared window systems. In *Conference proceedings on Empowering people: Human factors in computing system: special issue of the SIGCHI Bulletin*, pages 303–311, 1990.

[LLSG92] Rivka Ladin, Barbabra Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing High Avilability Using Lazy Replication. *ACM Transactions on Computer Systems*, pages 360–391, November 1992.

[LP00a] Radu Litiu and Atul Prakash. DACIA: A Mobile Component Framework for Building Adaptive Distributed Applications. In *Proceedings of Principles of Distributed Computing (PODC) 2000 Middleware Symposium*, July 2000.

[LP00b] Radu Litiu and Atul Prakash. Developing Adaptive Groupware Applications using a Mobile Component Framework. In *Proceedings of CSCW'2000*, December 2000.

[LWM99] W. Li, W. Wang, and I. Marsic. Collaboration Transparency in the DISCIPLE Framework. In *Proceedings of the ACM International Conference on Supporting Group Work (GROUP'99)*, Phoenix, AZ, November 1999.

[Mar99] I. Marsic. DISCIPLE: A Framework for Multimodal Collaboration in Heterogeneous Environments. *ACM Computing Surveys*, 1999.

[Mey89] Betrand Meyer. *Object-oriented Software Construction*. Prentice-Hall International, 4th edition, 1989. ISBN: 0-13-629031-0.

[Min75] M. Minsky. A framework for representing knowledge. *The Psychology of Computer Vision*, pages 211–277, 1975.

[MN98] T.D. Meijler and O. Nierstrasz. *Beyond Objects: Components.* Academic Press, 1998.

[Mor97] A.I. Morch. Three Levels of End-user Tailoring: Customization,Integration, and Extension. *Journal: Computers and Design in Context*, 1997.

[NPH99] Christian Nester, Michael Philippsen, and Bernhard Haumacher. A More Efficient RMI for Java. In *Proc. ACM 1999 Java Grande Conference*, 1999.

[omg00] CORBA Basics (CORBA Frequently Asked Questions). `http://www.omg.org/gettingstarted/corbafaq.htm`, 2000.

[ÖV91] M.T. Öszu and P. Valduriez. *Principles of Distributed Database Systems.* Prentice-Hall International, Inc., Englewood Cliffs, New Jersey, 1991.

[Phi99] W.Greg Philips. Architectures for Synchronous Groupware. tech. report 1999-425, Department of Computing and Information Science, Queen's University, Kingston, Ontario, 1999.

[Pre99] Wolfgang Pree. *Komponentenbasierte Softwareentwicklung mit Frameworks.* dpunkt Verlag, Heidelberg, 1999.

[PS94] Atul Prakash and Hyong Sop Shim. DistView: Support for Building Efficient Collaborative Applications using Replicated Objects. In *Proceedings of CSCW'94*, 1994.

[PS95] David Plainfosse and Marc Shapiro. A Survey of Distributed Garbage Collection Techniques. *International Workhop on Memory Management*, September 1995.

[RG96] M. Roseman and S. Greenberg. Building real time groupware with GroupKit, a groupware toolkit. *ACM Transactions on Computer-Human Interaction*, pages 66–106, March 1996.

[RU98] Jörg Roth and Claus Unger. DreamTeam - A Platform for Synchronous Collaborative Applications. *Groupware und organisatorische Innovation (D-CSCW'98)*, pages 153–165, 1998.

[RU00] Jörg Roth and Claus Unger. Developing synchronous collaborative applications with TeamComponents. In R.Dieng et al., editor, *Designing Cooperative Systems - Proceedings of coop'2000.* IOS Press, 2000.

[Sai98] Adib Saikali. The Hitchiker's Guide to the Microsoft Component Object Model. `http://www.csclub.uwaterloo.ca/\~asaikali/HitchGuideToCom.html`, September 1998.

[SC00] Oliver Stiemerling and Armin B. Cremers. The EVOLVE Project: Component-Based Tailorability for CSCW Applications. *AI and Society*, issue 14:120–141, 2000.

[SKFS99] Abdulmotaleb El Saddik, Oguzhan Karaduman, Stephan Fischer, and Ralf Steinmetz. Collaborative Working with Stand-Alone Applets. In *In Proc. of the 12th International Symposium on Intelligent Multimedia and Distance Education(ISIMADE'99).*, pages 203–209, August 1999. ISBN 0-921836-80-5.

[SKSH96] C. Schuckmann, L. Kirchner, J. Schümmer, and J.M. Haake. Designing Object-Oriented synchronous groupware with COAST. In *Proceedings of the ACM 1996 Conference on Computer Supported Cooperative Work (CSCW'96)*, pages 30–38. ACM Press, New York, 1996.

[SSS99] Christian Schuckmann, Jan Schümmer, and Peter Seitz. Modeling Collaboration using Shared Objects. In *Proceedings of ACM GROUP99, Intrnational Conference on Supporting Group Work.* ACM Press, 1999.

[STBT99] Jan Schümmer, Thomas Tesch, Ajit Bapat, and Daniel Tietze. Introducing Groupware in Administrative Environments - Experiences from the POLIWORK Project. *Bullinger, H.-J., Ziegler, J. (Ed.): Human-Computer Interaction: Communication, Cooperation, and Application Design. Proceedings of HCI International '99*, pages 492–496, 1999.

[Sti99] Oliver Stiemerling. Komponentenbasierte Anpassbarkeit von Groupware. In *Proceedings of DCSCW'98*, pages 225–236, Dortmund, 1999.

[Sti00] Oliver Stiemerling. *Component-Based Tailorability.* Ph.d. thesis, Rheinische Friedrich-Wilhelms-Universitt, July 2000.

[Sum98] Robert Summers. *Official Microsoft(c) NetMeeting(tm) Book.* Microsoft Press, 1998. ISBN: 1-57231-816-3.

[Szy97] Clemens Szypersky. *Component Software - Beyond Object-oriented Programming.* Addison-Wesley, 1997.

[Tan96] Andrew Tanenbaum. *Computer Networks.* Prentice Hall, 1996. ISBN: 0133499456.

[tH98] Henri ter Hofte. *Working Apart Together - Foundations for Component Groupware*, volume No. 001 of *Telematica Institute Fundamental Research Series.* Telematica Instituut, Enschede, the Netherlands, 1998. ISBN: 90-75176-14-7.

[The01] The Yankee Group. Communication, Collaboration, Coordination: The "Three Cs" of Workgroup Computing. `www.yankeegroup.com`, 2001.

[WDM99] W. Wang, B. Dorohonceanu, and I. Marsic. Design of the DISCIPLE Synchronous Collaboration Framework. In *Proceedings of the 3rd IASTED International Conference on Internet, Multimedia Systems and Applications (IMSA'99)*, pages 316–324, Nassau, Grand Bahamas, October 1999.

[Wei93] William E. Weihl. Transaction-Processign Techniques. In Sape Mullender (ed.), editor, *Distributed Systems, Second Edition*, chapter chapt.13, pages 329–351. 1993. ISBN: 0-201-62427-3.

[WHRT00a] Weigang Wang, Jörg M. Haake, Jessica Rubart, and Daniel A. Tietze. Hypermedia-Based Suppport for Cooperative Learning of Process Knowledge. *special issue on Support for Open and Distance Learning on the WWW in Journal of Network and Computer Applications*, pages 357–379, December 2000.

[WHRT00b] Weigang Wang, Jörg M. Haake, Jessica Rubart, and Daniel A. Tietze. Supporting Cooperative Learning of Process Knowledge on the World Wide Web. In *Proceedings of 26th EUROMICRO Conference, Vol.2*, pages 20–27, Maastricht, the Netherlands, September 2000.

[WJH97] O. Wolfson, S. Jajodia, and Y. Huang. An Adaptive Data Replication Algorithm. *ACM Transactions on Database Systems*, Vol. 22, No. 2:255–314, June 1997.

[Wol98] Ouri Wolfson. Adaptive Replication. `http://www.eecs.uic.edu/~wolfson/html/replic.html`, 1998.

[WP00] Martin Wessner and Hans-Rüdiger Pfister. Points of Cooperation: Integrating Cooperative Learning into Web-Based Courses. In *Proceedings of the NTCL2000, The International Workshop for New Technologies for Collaborative Learning*, Hyogo, Japan, 2000.

[WPS+00] Matthias Wiesmann, Fernando Pedone, Andr Schiper, Bettina Kemme, and Gustavo Alonso. Understanding Replication in Databases and Distributed Systems. *Proc. of the 20th International Conference on Distributed Computing Systems (ICDCS), Taipei, Taiwan*, 2000.

[XFT+01] Bo Xiao, Stephan Fischer, Daniel Tietze, Jörg M. Haake, and Ralf Steinmetz. Integratiing Multimedia support into Web-based Shared Workspaces. *Submitted to: WWW'10 Conference*, 2001.